Sven Overhage,
Klaus Turowski
(Eds.)

# Proceedings
# 1st Int. Workshop Component Engineering Methodology

September 24, 2003

Erfurt, Germany

German Computer Science Society
Working Group WI-KobAS (5.10.3)
Component-Based Business Applications

# Program Committee

# Preface

These proceedings contain the written position statements that have been accepted for presentation at the First International Workshop on Component Engineering Methodology (WCEM ’03), which was held in conjunction with the International Conference on Generative Programming and Component Engineering (GPCE’03) in Erfurt, Germany.

Having the long-term goal of establishing a methodology for industrial component-based application development, this first workshop focused on component specifications, which are an important prerequisite and integration instrument for the development of compatible (development) methods and tools. In particular, position statements referring to the following topics have been invited:

- Aspects of component specifications (e.g. assertions, behavior, architecture, method coordination, variability, and non-functional/quality characteristics)
- Applications for component specifications (e.g. component search, assessment, dynamic configuration, formal models to predict configuration behavior)
- Formal validation and certification of component specifications (e.g. certification methods)
- Standardization of component specifications (e.g. existing approaches and specification frameworks)

The submitted position statements have been reviewed by the program committee (using a double-blind refereeing procedure), which selected six contributions that contain up-to-date results of research projects or document experiences from industrial practice.

We would like to thank the authors of the position statements for contributing to the success of this workshop and the members of the program committee for carefully reviewing the submissions. Finally, we owe a thank to the organizing committee of the International Conference on Generative Programming and Component Engineering, which accepted our workshop proposal, helped us with the announcements, and integrated it into the conference program.

Augsburg, Germany
(September 2003)

*Sven Overhage,*
*Klaus Turowski*

# Table of Contents

# Ontological evaluation of the specification framework proposed by the "Standardized Specification of Business Components" memorandum – some preliminary results

Peter Fettke, Peter Loos

Johannes Gutenberg-Universität Mainz
Information Systems & Management
Lehrstuhl Wirtschaftsinformatik und Betriebswirtschaftslehre
D-55099 Mainz, Germany
E-Mail: {fettke|loos}@isym.bwl.uni-mainz.de

**Abstract:** The specification framework for business components proposed by the research group "Component Oriented Business Application System" defines seven specification levels of business components. According to this framework several notations are used to describe a business component, e. g. the OMG Interface Definition Language (OMG IDL) and the Object Constraint Language (OCL). The specification framework implicitly claims to allow a complete specification of a business component. However, this proposition is not justified by the authors of the specification framework. In this paper, we use the Bunge-Wand-Weber-model (BWW-model) to evaluate the specification framework's completeness. The BWW-model has already been used for the evaluation of several other modeling grammars. This study demonstrates that the proposed approach is feasible. Based on the preliminary findings, we suggest some directions for further developments of the specification framework.

## 1 Introduction

Component-based software development is a potential reuse paradigm for the future (D'SOUZA, WILLS 1998; SZYPERSKI 2002). While the required technologies for component-style system development are widely available, e.g. Sun's Enterprise Java Beans, a problem inhibits the breakthrough of the component paradigm in business application domains: In practice, there is a lack of a standardized, well-known approach to describe the business component's functionality and its quality characteristics. To overcome this situation, the research group "Component Oriented Business Application System" – a subgroup of the "Gesellschaft für Informatik" (German Informatics Society) – proposed a specification framework for business components (ACKERMANN et al. 2002) (see section 2.1, in the following, the term "specification framework" always refers to this proposal).

The specification framework implicitly claims to allow a *complete* specification of business components (ACKERMANN et al. 2002, 3-5). However, this proposition is not justified by the authors. The objective of this paper is to analyze the specification framework's completeness. We define the completeness of the specification framework in terms of ontology – a well-known branch of philosophy. Our analysis is based on the Bunge-Wand-Weber-model (BWW-model, section 2.2). The BWW-model has already been used for the evaluation of several other modeling grammars (see section 2.4).

The main contributions of this work is to show some ontological deficiencies of the specification framework. In doing so, we will demonstrate the usefulness of the BWW-model for the evaluation of business component specifications too. However, a critical discussion of limitations of an ontological evaluation is out of the scope of this paper.

The paper is structured as follows: The theoretical background of this study is described in the next section. In section three some preliminary results of the ontological evaluation of the specification framework are presented. Section four concludes our findings and gives directions for further research.

## 2 Theoretical background

In this section we describe the theoretical background of this study. We present:

- an outline of the specification framework (section 2.1),
- an outline of the BWW-model (section 2.2),
- the method for the ontological evaluation of modeling grammars (section 2.3), and
- an overview of prior work on ontological evaluations (section 2.4).

### 2.1 Standardized Specification of Business Components

The specification framework uses seven specification levels (cf. figure 1) (ACKERMANN et al. 2002). Each level focuses on a specific aspect of a business component specification and addresses the needs of different development roles. Various notations are used on all specification layers. For an in-depth discussion of the various specification aspects and notations used on each specification level see the work cited above. An exemplar specification of a business component based on this specification framework is given in (FETTKE, LOOS 2003b; FETTKE, LOOS 2003c).

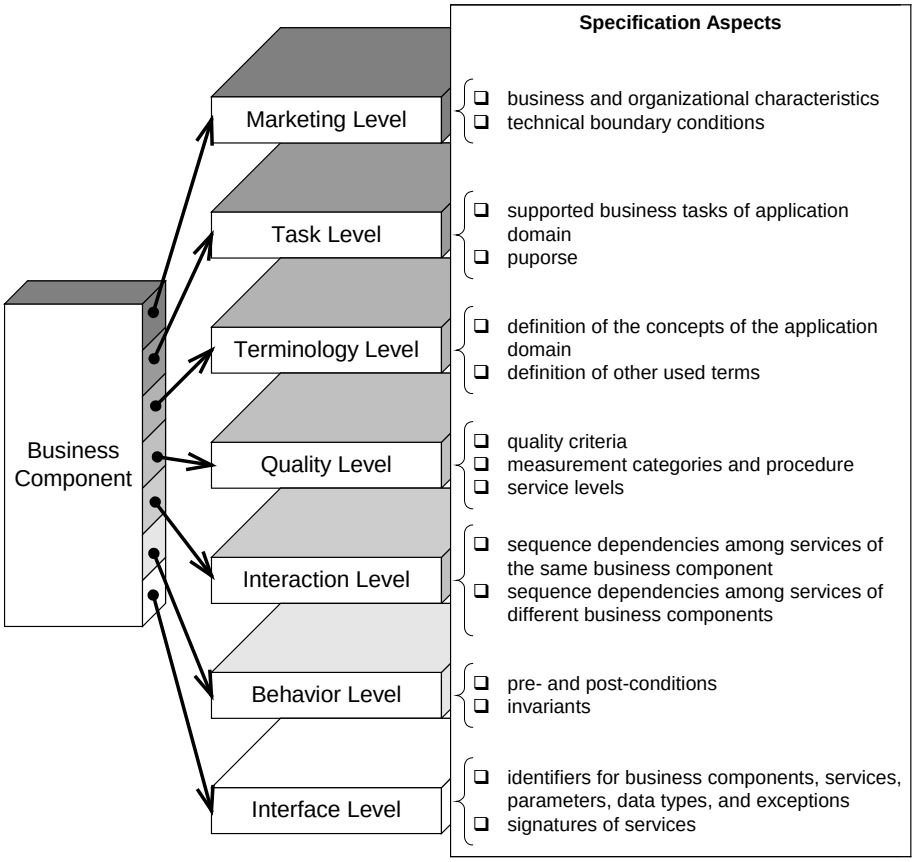


Figure 1: Specification levels and specification aspects (ACKERMANN et al. 2002)

## 2.2 Bunge-Wand-Weber-model

The philosophical discipline of ontology studies "the most pervasive features of reality, such as real existence, change, time, causation, chance, life, mind, and society"(BUNGE 2003, p. 201). This discipline provides a foundation for conceptual modeling, if the assumption is followed that conceptual models represent reality (WAND et al. 1995). Note, that our definition of the term "ontology" may not be confused with the common interpretation of this term as a more or less formalized "concept directory" (FENSEL 2001; GRUBER 1995).

Up to the present, there does not exist a generally accepted set of ontological principles and assumptions. Our analysis is based on the ontology initially introduced by Bunge (BUNGE 1977; BUNGE 1979) and adapted by Wand & Weber for the information systems field (WAND, WEBER 1989b; WAND, WEBER 1995; WEBER 1997). In the following, the term "ontology" refers to the Bunge-Wand-Weber-model (BWW-model). To improve the BWW-model's clarity, Rosemann and Green developed a meta-model of the BWW-model (ROSEMANN, GREEN 2002). For brevity, we do not introduce the BWW-model in detail. Instead, table 1 summarizes it's main constructs.

| **Ontological Construct** | **Explanation** |
|---|---|
| Thing | "The elementary unit in our ontological model. The real world is made up of things. A composite thing may be made up of other things (composite or primitive)" |
| Property | "Things possess properties. A property is modeled via a function that maps the thing into some value. A property of a composite thing that belongs to a component thing is called a *hereditary* property. Otherwise it is called an *emergent* property. A property that is inherently a property of an individual thing is called an *intrinsic* property. A property that is meaningful only in the context of two or more things is called a *mutual* or *relational* property" |
| State | "The vector of values for all property functions of a thing" |
| Conceivable state space | "The set of all states that the thing might ever assume" |
| State law | "Restricts the values of the property functions of a thing to a subset that is deemed lawful because of natural laws or human laws" |
| Lawful state space | "The set of states of a thing that comply with the state laws of the thing. It is usually a proper subset of the conceivable state space" |
| Event | "A change of state of a thing. It is effected via a transformation (see below)" |
| Event space | "The set of all possible events that con occur in the thing" |
| Transformation | "A mapping from a domain comprising states to a codomain comprising states" |
| Lawful transformation | "Defines which events in a thing are lawful" |
| Lawful event space | "The set of all events in a thing that are lawful" |
| History | "The chronologically ordered states that a thing traverses" |

Table 1: Constructs of the BWW-model (source: (WAND, WEBER 1993; WEBER, ZHANG 1996), part 1/2)

| | |
|---|---|
| Coupling | "A thing acts on another thing if its existence affects the history of the other thing. The two things are said to be coupled or interact" |
| System | "A set of things is a system if, for any bi-partitioning of the set, couplings exist among things in the two subsets" |
| System composition | "The things in the system" |
| System environment | "Things that are not in the system but interact with things in the system" |
| System structure | "The set of couplings that exist among things in the system and things in the environment of the system" |
| Subsystem | "A system whose composition and structure are subsets of the composition and structure of another system" |
| System decomposition | "A set of subsystems such that every component in the system is either one of the subsystems in the decomposition or is included in the composition of one of the subsystems in the decomposition" |
| Level structure | "Defines a partial order over the subsystems in a decomposition to show which subsystems are components of other subsystems or the system itself" |
| Stable state | "A state in which a thing, subsystem or system will remain unless forced to change by virtue of the action of a thing in the environment (an external event)" |
| Unstable state | "A state that will be changed into another state by virtue of the action of transformation in the system." |
| External event | "An event that arises in a thing, subsystem or system by virtue of the action of some thing in the environment on the thing, subsystem or system. The before-state of an external event is always stable. The after-state may be stable or unstable." |
| Internal event | "An event that arises in a thing, subsystem, or system by virtue of lawful transformations in the thing, subsystem, or system. The before-state of an internal event is always unstable. The after-state may be stable or unstable." |
| Well-defined event | "An event in which the subsequent state can always be predicted given the prior state is known" |
| Poorly defined event | "An event in which the subsequent state cannot be predicted given the prior state is known" |
| Class | "A set of things that possess a common property" |
| Kind | "A set of things that possess two or more common properties" |

Table 1: Constructs of the BWW-model (source: (WAND, WEBER 1993; WEBER, ZHANG 1996), part 2/2)

The BWW-model has been successfully used for several application areas: for instance, definition of an object model (WAND 1989), formalization of audit-procedures (WAND, WEBER 1989a), foundation of model quality (WAND, WANG 1996), proposal for modeling rules (WAND, STOREY, WEBER 1999) and evaluation of modeling grammars.

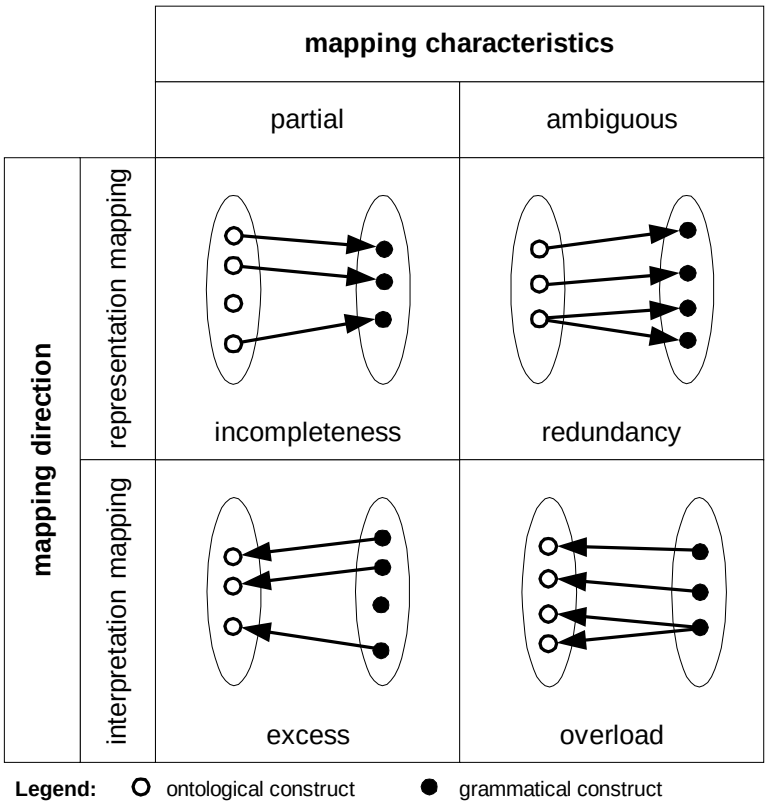


Figure 2. Ontological deficiencies of a grammar

## 2.3 Ontological evaluation of modeling grammars

The objective of an ontological evaluation is to develop a transformation mapping between the constructs of the BWW-model and the constructs of the modeling grammar being assessed (WAND, WEBER 1993; WAND, WEBER 1995; WAND, WEBER 2002; WEBER 2002; WAND et al. 1995). The transformation mapping consists of two mathematical mappings: First, a representation mapping describes how the constructs of the BWW-model are mapped onto the grammatical constructs. Second, the interpretation mapping describes how the grammatical constructs are mapped onto the constructs of the BWW-model.

With respect to both mappings, four ontological deficiencies can be distinguished (figure 2):

- Incompleteness: Can each ontological construct be mapped onto a construct of the grammar? A grammar is incomplete if the representation mapping is not defined in total. Otherwise a grammar is complete.
- Redundancy: Can each ontological construct be mapped onto exactly one or more than one grammatical constructs? A grammar is redundant if the representation mapping is ambiguous.
- Excess: Can each grammatical construct be mapped onto an ontological construct? A grammatical construct is excessive if it can not be mapped onto an ontological construct. A grammar is excessive if at least one of its constructs is excessive.
- Overload: Can each grammatical construct be mapped onto exactly one or on more than one ontological constructs? A grammatical construct is overloaded if it can be

mapped onto more than one ontological construct. A grammar is overloaded if at least one of its constructs is overloaded.

We refer to the term "grammar" as "ontologically clear" if it is neither incomplete nor redundant. A grammatical construct is adequate if it is neither excessive nor overloaded, so that it is defined unambiguously with respect to the interpretation mapping. A grammar is adequate if each of its grammatical constructs is adequate.

## 2.4 Prior research

We conclude the theoretical background of our study with an overview of the prior research. Table 2 depicts prior research, modeling grammars evaluated by the studies and used empirical research methods – if applicable. The primary intention of this overview is to demonstrate that an ontological evaluation is already used for several other applications areas. However, we are not aware of an ontological evaluation approach in the area of business component specifications.

| Authors | NIAM | DFD | ERM | UML | ARIS | OML | SOM | Jackson | Others | Empirical Inquiry |
|---|---|---|---|---|---|---|---|---|---|---|
| (WAND, WEBER 1989b) | | ● | ● | | | | | | | - |
| (WAND, WEBER 1990) | | | ● | | | | | | | - |
| (WEBER, ZHANG 1991) | ● | | | | | | | | | - |
| (WAND, WEBER 1993) | ● | ● | | | | | | | ● | - |
| (WAND, WEBER 1995) | | | ● | | | | | | | - |
| (ROHDE 1995) | | | | | | | | ● | | - |
| (WEBER, ZHANG 1996) | ● | | | | | | | | | - |
| (WEBER 1996) | | | ● | | | | | | | laboratory experiment |
| (GREEN 1996) | | ● | ● | | | | | | ● | survey |
| (WAND, STOREY, WEBER 1999) | | | ● | | | | | | | - |
| (GREEN, ROSEMANN 2000) | | | | | ● | | | | | - |
| (OPDAHL, HENDERSON-SELLERS 2001) | | | | | | ● | | | | - |
| (EVERMANN, WAND 2001a) | | | | ● | | | | | | - |
| (EVERMANN, WAND 2001b) | | | | ● | | | | | | - |
| (GREEN, ROSEMANN 2001) | | | | | ● | | | | | survey |
| (BODART et al. 2001) | | | ● | | | | | | | laboratory experiment |
| (OPDAHL, HENDERSON-SELLERS 2002) | | | | ● | | | | | | - |
| (FETTKE, LOOS 2003a) | | | | | | | ● | | | - |

Table 2: Overview of prior work on the ontological evaluation of modeling grammars

## 3 Results

We evaluate the specification framework with respect to its ontological characteristics. For reasons of brevity, we focus our analysis on the representation mapping. See table 3 for an overview of the representation mapping introduced. We see this mapping as a first idea how to ontologically interpret the specification framework.

A BWW-thing can be represented by a whole business component (e.g. a business component "article"), indirectly by a data type on the interface level (e. g. data type "article"), and by a concept definition on the terminology level (e.g. concept "article"). So, these constructs are ontological partially redundant and may lead to confusions. The BWW-class and BWW-kind constructs are not represented by the specification framework.

There are several ways to represent properties of a BWW-thing. First of all, attributes of data types on the interface level may represent a property, e.g. article's names may be represented by strings. Furthermore, BWW-properties can be described by quality criteria, concept definitions on the terminology level, and characteristics on the marketing level. The BWW-state of a BWW-thing is represented by concrete values of the properties described before. Restrictions of states can be formulated by expressions on the behavior level. For example, the warehouse stock of an article may not be less than 100 units. State values represented by quality criteria or characteristics on the marketing level cannot be restricted. BWW-state laws can be represented by expressions on the behavior and interaction level. However, the BWW-lawful state space is not represented explicitly.

Pre- and post-conditions of expressions on the behavior and interaction level partially represent BWW-events. The pre-condition can define the pre-state of an event (e.g. delivery received) and the after-state of the event is represented by the post-condition (e.g. warehouse stock is incremented). The BWW-event space is not represented explicitly. Furthermore, the specification framework does not allow to explicitly represent external, internal, well-defined and poorly defined events. BWW-transformations are represented by services and the corresponding specifications on the behavior and interaction level. All BWW-transformations are deemed to be lawful. As the BWW-event space, the BWW-lawful event space is not represented explicitly. The BWW-history cannot be represented at all.

A BWW-coupling between BWW-things can be partially represented by the "interface extern" specification on the interface level. A BWW-system is defined by a whole business component. Furthermore, the BWW-system environment is partially represented by the "interface extern" specification on the interface level. However, it is not possible to represent system composition, system structure, subsystem, system decomposition, and level structure within the specification framework.

| Ontological constructs | Constructs of the specification framework |
|---|---|
| Thing | Whole business component, data types, concept |
| Property | Data types, concept, quality criteria, characteristics on the marketing level |
| State | Values of data types are not directly represented, values of quality criteria, values of the characteristics on the marketing level |
| Conceivable state space | Restrictions of values of data types can be represented by specifications on the behavior level; quality criteria and characteristics on the marketing level cannot be restricted |
| State law | Can be represented by invariants specifications on the behavior and interaction level |
| Lawful state space | Is not represented directly |
| Event | Events are represented by pre- and post-conditions of expression on the behavior and interaction level. |
| Event space | Is not represented explicitly |
| Transformation | Transformations are represented by services and the specifications on the behavior and interaction level. |
| Lawful transformation | See transformations |
| Lawful event space | Is not represented explicitly |
| History | - |
| Coupling | Some couplings between components are represented by the “interface extern” on the interface level. |
| System | Whole component |
| System composition | - |
| System environment | Partially represented by the “interface extern” on the interface level. |
| System structure | - |
| Subsystem | - |
| System decomposition | - |
| Level structure | - |
| Stable state | - |
| Unstable state | - |
| External event | - |
| Internal event | - |
| Well-defined event | - |
| Poorly defined event | - |
| Class | - |
| Kind | - |

Table 3: Ontological analysis of specification framework

## 4 Conclusions, limitations and further work

The preliminary findings of our ontological analysis allows to draw the following conclusions:

- First, the main constructs of the BWW-model (thing, property, transformation, state) can be represented by the specification framework. So, the specification framework can be called ontological complete with respect to these main constructs.
- Second, the BWW-constructs representing the structure of a system (BWW-system structure, BWW-subsystem etc.) cannot be represented by the specification framework. So we conclude, that the specification framework should be enhanced by constructs which allow to represent the system's composition. On the one hand, it may be argued that such constructs will violate the black-box-principle of a component specification. On the other hand, the composition of simple business components to larger assemblies or modules is a well-known engineering principle. Furthermore, the developer of a business component assembly will have the need to specify a business component structure. In summary, we believe that the advantage of an explicit specification of the business component's structure will outperform its flaws.
- Third, BWW-events may be represented by the specification framework. However, it is not possible to explicitly define external and internal events. We argue that this ontological deficiency may cause problems when using the business component in different application environments. For example, it cannot straightforward be assessed if a business component fulfils the state requirements of given business environment. Such information is always attached to a specific service of a business component and not to a business component as a whole.
- Fourth, the ontological evaluation shows that some constructs of the specification framework are ontological redundant. For example, BWW-things, BWW-properties and BWW-states can be represented by several constructs. These deficiencies may lead to problems when using the specification framework. For example, should the characteristic "real-time accounting" be described on the quality level or on the task-level? On the one hand, it can be argued that this characteristic is a business task, so it should be described on the task level. On the other hand, "real time accounting" can be clearly viewed as a quality characteristic of book keeping components.

Next, we discuss some limitations of our work. An ontological evaluation is not an objective procedure or an algorithm which can be codified in some program language and processed automatically. Instead, it relies on the evaluator's interpretation of modeling constructs. This interpretation is to a certain degree subjective. So, it is necessary that the interpretation's rationale is made explicit and justified by specific reasons. We have to admit that our analysis is not based on an in-detail discussion of possible representation and interpretation mappings. This is a clear limitation of our study. However, the intention of our work is to point out that the implicitly stated claim of the specification framework to allow a *complete* specification of business components is not justified yet. Hence, we propose the BWW-model as a mean to analyze the completeness of the specification framework. The obtained results of our preliminary study are promising and demonstrate that this approach is feasible. On the other hand, we believe that other criteria can be developed to describe and to measure the quality of business component specification frameworks. For example, from a user oriented point of view, proposed specification techniques should be easily to use .

We see several directions for further work: First, our study focuses the representation mapping. It is possible to investigate ontological interpretations of the constructs of the specification framework. Furthermore, our study examined the correspondence between the BWW-model and the constructs of the specification framework on a rather rough level. Such an on-

tological evaluation should be conducted in a deeper way to exactly identify partial correspondences. Second, the ontological deficiencies identified in this study give some insights to improve and to develop the specification framework. Third, the specification framework should not only evaluated analytical, but also empirical. The BWW-model provides for such investigations a sound theoretical foundation.

## References


ACKERMANN, J.; BRINKOP, F.; CONRAD, S.; FETTKE, P.; FRICK, A.; GLISTAU, E.; JAEKEL, H.; KOTLAR, O.; LOOS, P.; MRECH, H.; RAAPE, U.; ORTNER, E.; OVERHAGE, S.; SAHM, S.; SCHMIETENDORF, A.; TESCHKE, T.; TUROWSKI, K.: Standardized Specification of Business Components. http://wi2.wiwi.uni-augsburg.de/gi-memorandum.php.htm, accessed: 2003-03-01.

BODART, F.; PATEL, A.; SIM, M.; WEBER, R.: Should Optional Properties Be Used in Conceptual Modelling? A Theory and Three Empirical Tests. In: Information Systems Research 12 (2001) 4, pp. 384-405.

BUNGE, M.: Ontology I: The Furniture of the World. Dordrecht, Holland 1977.

BUNGE, M.: Ontology II: A World of Systems. Dordrecht, Holland 1979.

BUNGE, M.: Philosophical Dictionary. 2. ed., Amherst, New York 2003.

D'SOUZA, D. F.; WILLS, A. C.: Objects, Components, and Frameworks with UML - The Catalysis Approach. Reading, MA, et al. 1998.

EVERMANN, J.; WAND, Y.: An Ontological Examination of Object Interaction in Conceptual Modeling. Proceedings of the 11th Workshop on Information Technologies and Systems (WITS 2001). New Orleans, Louisiana 2001a

EVERMANN, J.; WAND, Y.: Towards Ontologically Based Semantics for UML Constructs. In: H. S. Kunii; S. Jajodia; A. Sølvberg (Ed.): Conceptual Modeling - ER 2001 - 20th International Conference on Conceptual Modeling, Yokohama, Japan, November 27-30, 2001, Proceedings. Berlin, Heidelberg 2001b, pp. 354-367.

FENSEL, D.: Ontologies - A Silver Bullet for Knowledge Management and Electronic Commerce. Berlin et al. 2001.

FETTKE, P.; LOOS, P.: Ontologische Analyse des Semantischen Objektmodells. 11. Fachtagung Modellierung betrieblicher Informationssysteme (forthcoming). Bamberg 2003a

FETTKE, P.; LOOS, P.: Specification of Business Components. In: M. Aksit; M. Mezini; R. Unland (Ed.): Objects, Components, Architectures, Services, and Applications for a Networked World - International Conference NetObjectDays, NODe 2002, Erfurt, Germany, October 7-10, 2002, Revised Papers. Berlin et al. 2003b, pp. 62-75.

FETTKE, P.; LOOS, P.: Specifying Business Components in Virtual Engineering Communities. Ninth Americas Conference on Information Systems 2003. Tampa, FL, USA 2003c, pp. 1937-1947.

GREEN, P.: An Ontological Analysis of Information Systems Analysis and Design (ISAD) Grammars in Upper Case Tools. PhD Thesis, 1996.

GREEN, P.; ROSEMANN, M.: Integrated Process Modeling: An Ontological Evaluation. In: Information Systems 25 (2000) 2, pp. 73-87.

GREEN, P.; ROSEMANN, M.: Ontological Analysis of Integrated Process Models: Testing Hypotheses. In: Australian Journal on Information Systems 9 (2001) 1, pp. 30-38.

GRUBER, T. R.: Toward principles for the design of ontologies used for knowledge sharing. In: International Journal of Human-Computer Studies 43 (1995), pp. 907-928.

OPDAHL, A. L.; HENDERSON-SELLERS, B.: Grounding the OML metamodel in ontology. In: The Journal of Systems and Software 57 (2001) 2, pp. 119-143.

OPDAHL, A. L.; HENDERSON-SELLERS, B.: Ontological Evaluation of the UML Using the Bunge-Wand-Weber Model. In: Software and Systems Modeling 1 (2002) 1, pp. 43-67.

ROHDE, F.: An Ontological Evaluation of Jackson's System Development Model. In: Australian Journal of Information Systems 2 (1995) 2, pp. 77-87.

ROSEMANN, M.; GREEN, P.: Developing a meta model for the Bunge-Wand-Weber ontological constructs. In: Information Systems 27 (2002) 2, pp. 75-91.

SZYPERSKI, C.: Component Software - Beyond Object-Oriented Programming. 2. ed., London et al. 2002.

WAND, Y.: A Proposal for a Formal Model ob Objects. In: W. Kim; F. Lochovsky (Ed.): Object-Oriented Concepts, Applications, and Databases. Reading, MA 1989, pp. 537-559.

WAND, Y.; MONARCHI, D. E.; PARSONS, J.; WOO, C. C.: Theoretical foundations for conceptual modelling in information systems development. In: Decision Support Systems 15 (1995), pp. 285-304.

WAND, Y.; STOREY, V. C.; WEBER, R.: An Ontological Analysis of the Relationship Construct in Conceptual Modeling. In: ACM Transactions on Database Systems 24 (1999) 4, pp. 494-528.

WAND, Y.; WANG, R. Y.: Anchoring Data Quality Dimensions in Ontological Foundations. In: Communications of the ACM 39 (1996) 11, pp. 86-95.

WAND, Y.; WEBER, R.: A Model of Control and Audit Procedure Change in Evolving Data Processing Systems. In: The Accounting Review 64 (1989a) 1, pp. 87-107.

WAND, Y.; WEBER, R.: An Ontological Evaluation of Systems Analysis and Design Methods. In: E. D. Falkenberg; P. Lindgreen (Ed.): Information Systems Concepts: An In-Depth Analysis. North-Holland 1989b, pp. 79-107.

WAND, Y.; WEBER, R.: Toward a Theory of the Deep Structure of Information Systems. In: J. I. DeGross; M. Alavi; H. Oppelland (Ed.): International Conference on Information Systems. Copenhagen, Denmark 1990

WAND, Y.; WEBER, R.: On the ontological expressiveness of information systems analysis and design grammars. In: Journal of Information Systems 3 (1993) 4, pp. 217-237.

WAND, Y.; WEBER, R.: On the deep structure of information systems. In: Information Systems Journal 5 (1995), pp. 203-223.

WAND, Y.; WEBER, R.: Research Commentary: Information Systems and Conceptual Modelling - A Research Agenda. In: Information Systems Research 13 (2002), pp. 363-377.

WEBER, R.: Are Attributes Entities? A Study of Database Designer's Memory Structures. In: Information Systems Research 7 (1996) 2, pp. 137-162.

WEBER, R.: Ontological Foundations of Information Systems. Melbourne 1997.

WEBER, R.: Conceptual Modelling and Ontology: Possibilities and Pitfalls. http://er2002.cs.uta.fi/info/weberspeech.rtf, accessed: 2003-01-27.

WEBER, R.; ZHANG, Y.: An Ontological Evaluation of NIAM's Grammar for Conceptual Schema Diagrams. International Conference on Information Systems. New York, New York 1991

WEBER, R.; ZHANG, Y.: An analytical evaluation of NIAM's grammar for conceptual schema diagrams. In: Information Systems Journal 6 (1996), pp. 147-170.

# Specifying Contractual Use, Protocols and Quality Attributes for Software Components

Steffen Becker, Ralf H. Reussner, Viktoria Firus
Software Engineering Group, Department of Computing Science
University of Oldenburg, Germany
becker|reussner|firus@informatik.uni-oldenburg.de

August 28, 2003

**Abstract**

We discuss the specification of signatures, protocols (behaviour) and quality of service within software component specification frameworks. In particular we focus on (a) contractually used components, (b) the specification of components with variable contracts and interfaces, and (c) of quality of service. Interface descriptions including these aspects allow powerful static interoperability checks. Unfortunately, the specification of constant component interfaces hinders the specification of quality attributes and impedes automated component adaptation. This is because, especially quality attributes heavily depend on the components context. To enable the specification of quality attributes, we demonstrate the inclusion of *parameterised contracts* within a component specification framework. These parameterised contracts compute adapted, context-dependent component interfaces (including protocols and quality attributes). This allows to take context dependencies into account while allowing powerful static interoperability checks.

## 1 Introduction

A specification framework for components has to provide information for several purposes, such as component retrieval and assessment, component deployment, interoperability checks, automated component adaptation, etc.

Current specification frameworks include signatures of the services offered by a component (e.g., UDDI [1]) or comprise additional metadata to classify components (easing the retrieval) and to specify additional component attributes (such as quality) [2]. In any case, specification frameworks for components include the "classic" interface models (signature-list based interface models), stemming from object based middleware, such as the CORBA-IDL [3].

However, using object based interface models is not appropriate for software components for (at least) three reasons:

1. Object interfaces model only provided services, not the required services. As argued in section 3, the contractual use of components is only possible, if a component not only specifies the services offered, but also the services required for proper operation. Interoperability checks between two components $A$ and $B$ depend on the specification of both: the services $A$ requires from $B$ and the services $B$ offers (to $A$, or to any other component).

2. Object interfaces include only signatures. Adding behavioural specifications and quality attributes significantly increases the power of interoperability checks (i.e. the class of detectable errors). Errors due to wrong service call sequences or insufficient quality of service can be detected (and hence excluded) before using the software.

3. Objects have fixed interfaces. An object interface corresponds to the functionality implemented by the object. This also holds for component interfaces and components. However, the deployment context of a component is variable. It heavily influences (a) the functionality offered (or effectively required) by the component, (b) the protocol and (c), most obviously quality attributes such as performance or reliability [4, 5].

The contribution of this paper is a syntax for including *parameterised contracts* into the specification framework of the working group 5.10.3 of the German Informatics society (G.I. e.V.) [2]. Parameterised contracts (as introduced in [6] and formally discussed in [4, 7]) compute context dependent contracts (i.e., provides- and requires-interfaces) and have been deployed for predicting the component reliability in dependency of its context [7]. Further on this paper discusses the importance of tool support for generating parts of the specifications proposed in this paper.

The structure of this paper is as follows. In the following section some prerequisites are mentioned. The term "contractual use" of software components and its relation to interoperability checks is clearified in section 3. Parameterised contracts for signatures, protocols and quality attributes are introduced afterwards. The importance of tool support for the specification of parameterised contracts is discussed in section 5. After the presentation of related work (section 6), we conclude with a summary and the discussion of open issues and future work in the last section.

## 2 Prerequisite

In contrary to the specification framework which we choose to base our specification tasks on, we differentiate strictly between a component and its interfaces because of the possibility to differentiate between the specification (interface) and the actual implementation. Those interfaces can be further divided into two categories: provided interfaces and required interfaces.

The concepts presented in this paper rely on the separation of a component on the one hand side, represented by the actual implementation of the component, and on the other hand on one or more interfaces specifying the services offered by the component to potential external users. As we will show in section 4 the provides-interface is calculated dynamically during composition time. Therefore it should be seen not tiedly coupled to the component as the same implementation may expose different interfaces depending on the reuse context in which the component is deployed.

Further on it is important for our work to distinguish between provided and required interfaces. The first describe services offered by a component, the later services needed by the component (i.e., services from other components). Components connected to any interface of the component form the *environment* of the component.

Although the need of requires-interfaces is obvious for static interoperability and substitutability check (and well-known in literature [8, 9]), current component models like Sun's EJB or Microsoft's .NET only contain provides-interfaces (one notable exception is CORBA 3.0). As we aim at providing tools to support those interoperability checks during component configuration we focus our work on supplying the necessary specifications needed to perform this task.

A sub task of checking the interoperability of components is the matching of signature names. Imagine one component producer calling a provided function `StartFundsTransfer` and an other one simply calling it `TransferFunds`. We assume that the matching of those names can be done by the means of the normative language specified on the terminological layer of the components specification (e.g. if all the components use the same normative language) or it has to be done manually during composition time by the configurator. Therefore we assume for the rest of this paper that signature names have been matched already.

The specification framework we utilize in this paper already fulfils the stated requirements except that a component may not have multiple interfaces. Interface specifications are separated from other specification attributes by the use of a dedicated layer for this technical information. The provides-interface is modelled on this layer as well as required services of external components. Parameterised contracts specify a link between the required and provided interfaces.

Finally it is worth mentioning that we concentrate solely on the technical aspects of the component specification. The impact of this work on the remaining layers of the specification framework is outside the scope of this paper.

# 3 Contractual Use of Components and Interoperability Checks

Much of the confusion about the term ”contractual use” of a component comes from the double meaning of the term ”use” of a component. The ”use” of a component refers often to the following:

1. the usage of a component during run-time. This is, calling services of the component, like calling `TransferFunds` on a payment component.

2. the usage of a component during composition time. This is, placing a component in a new reuse-context, like it happens when architecting systems, or reconfiguring existing systems (e.g., updating the component).

Depending on the above case, contracts play a different role. Contracts are assumed to be known, as they are well known in software engineering literature [10] and are also included in the regarded specification framework on the behavioural layer. Instead of explaining the design by contract paradigm again the essence is summarized here by the following sentence in a general form:

> If the client fulfils the precondition of the supplier, the supplier will fulfil its postcondition.

Let’s get back to the two types of the ”use” of a component. It is clear, that the component plays the role of a supplier in both cases. In order to specify contracts the pre- and postconditions as well as the clients additionally need to be identified. The use of a component during run-time is obviously simply calling the components services. Hence, the clients of the component are all the components calling a service of the supplying component, which are all those components connected to the provides-interface of the supplier. The pre- and postconditions involved in this case are simply those specified for the affected service itself. Therefore it should be evident that this type of use is nothing different as using a method. Thus this case should be called the use of a *component service* instead of the use of a *component* and is being disregarded in the following.

The other case of component usage (usage at composition time) is the actual important case, when talking about the contractual use of components. This is the case, when architecting systems out of components or deploying components within existing systems for reconfigurations. Consider a component $C$ which is acting as a supplier, and the environment acting as client. The component offers services to the environment (i.e., the components connected to $C$’s provides-interface(s)). According to the above discussion of contracts, these offered services are the postcondition of the component, because it is that, what the client can expect from a working component. Also according to Meyers above description of contracts, the precondition is that, what the component expects from its environment (actually all components connected to $C$’s requires-interface(s)) to be provided by the environment, in order to enable $C$ to offer its services (as stated in its postcondition). Hence, the precondition of a component is stated in its requires-interfaces.

Analogously to the above single sentence formulation of a contract, we can state:

> If the user of a component fulfils the components’ required interface (offers the right environment) the component will offer its services as described in the provided interface.

Note that checking the satisfaction of a requires-interface includes checking whether the contracts of required services (the service contracts specified in the requires-interface(s)) are sub-contracts of the service contracts stated in the provides-interfaces of the required components. A detailed description of subcontracts can be found in [11, p. 573]. The contractual use of components enables interoperability checks that can be performed when architecting new systems or replacing components during system maintanance [4].

There is a range of formalisms used for specifying pre- and postconditions, defining a range of interface models for components (see for extensive discussions and various models e.g., [12, 13, 14]). This leads naturally to different kinds of contracts for components [15].

# 4 Parameterised Contracts

In daily life of component reuse, a component rarely fits directly in a new reuse context. For a component developer it is hard to foresee all possible reuse contexts of a component in advance (i.e., during design-time). One of the severe consequences for component oriented programming is that one cannot provide the component with all the configuration possibilities which will be required for making the component fit into future reuse contexts. Coming back to our discussion about component contracts, this means, that in practice one single pre- and postcondition of a component will not be sufficient. Consider the following two cases:

1. the precondition of a component is not satisfied by a specific environment while the component itself would be able to provide a meaningful subset of its functionality.

2. a weaker postcondition of a component is sufficient in a specific reuse context (i.e., not the full functionality of a component will be used). Due to that, the component will itself require less functionality at its requires-interface(s), i.e., will be satisfied by a weaker precondition.

Hence, what we need are not static pre- and postconditions, but *parameterised contracts* [4, 14]. In the first case a parameterised contract computes the postcondition which is computed in dependency of the strongest precondition guaranteed by a specific reuse context (hence the postcondition is parameterised with the precondition). In the second case the parameterised contract computes the precondition in dependency of the postcondition (which acts as a parameter of the precondition). For components this means, that provides- and requires-interfaces are not fixed, but a provides-interface if computed in dependency of the actual functionality a component receives at its requires-interface and a requires-interface is computed in dependency of the functionality actually requested from a component in a specific reuse context. Hence, opposed to classical contracts, one can say:

> Parameterised contracts link the provides- and requires-interface(s) of the same component. They have a range of possible results (i.e., new interfaces).

Interoperability is a special case now: if a component is interoperable with its environment, its provides-interface will not change. If the interoperability check fails, a new provides-interface will be computed.

Like classical contracts, parameterised contracts depend on the actual interface model and should be statically computable. In any case, there's no need for the software developer to foresee possible reuse contexts. Only the specification of a bidirectional mapping between provides- and requires-interfaces is necessary.

Resulting from this there is a need for the component supplier to specify the parameterised contract in the components specification to enable anyone trying to reuse the component to determine the components capabilities in a certain environment or to learn about the environment one has to provide to get the needed functionality. To achieve this we propose in the following syntactical notations and the according semantics which allows the specification of parameterised contracts in the initially mentioned specification framework.

## 4.1 Signatures

The specification framework utilizes CORBA-IDL to specify interfaces as primary notation. As mentioned earlier the provided interface and a list of external services (required interface) are already included in the specification. Therefore there is only the need to add information about the parameterised contract of the component.

CORBA-IDL uses signature lists to specify the services of a component. In addition to the list of provided services and the list of required services we need a mapping between every provided service and the respective required services. This means that for each provided service a list of required external services must be provided by the component developer or has to be extracted by code analysis tools. When computing the actual provides-interface a service is only included in the provides-interface, if all its required services are provided by the environment.

Hence, the specification on the interface layer of the utilized specification framework should be expanded by the specification of a parameterised contract using the following syntax. The contract specification can simply be appended to the IDL definition of the respective interface.

The syntax of the proposed specification can be taken from the following extended BNF:

```
ParameterisedContract ::= "parametrised contract {"
                          (ServiceEffectSpecification)+ "}"
ServiceEffectSpecification ::= SerivceID "{"
           ((ServiceIDExtern ", ")* ServiceIDExtern | "") "}"
ServiceID ::= Identifier
ServiceIDExtern ::= Identifier "::" Identifier
```

For the rest of this paper a payment component should be regarded which makes use of parameterised contracts. The example component is capable of performing funds transactions either by requesting a bank transfer or by the use of the customers credit card information. As a matter of fact, the supplied credit card information needs to be validated before a transaction will be accepted. For this, a remote component hosted by the credit card company is being called. As the usage of the validation component is dependent on the payment of a monthly fee the component will not be available in every environment for economic reasons. Further on the component needs a database connection for caching purposes. The database connection is also needed for performing bank transactions because it contains a mapping between the bank codes and the names of the respective banks. Nevertheless bank transactions can be offered without the credit card validation services so that the component might still be useful in environments not providing these services. Hence, the component producer decided to use a parameterised contract to reflect this scenario in the components implementation in order to offer the component to a larger group of customers.

Using this example the payment component and its parameterised contract may be specified as follows on the interface layer of the specification framework.

```
interface Payment
{
  void CreditCardPayment (in double amount, in CreditCardInformation info);
  void BankTransferPayment (in double amount, in BankAccountInformation info);
}
interface extern
{
  void CreditCardValidator::ValidateInformation (in CreditCardInformation info);
  DBConnection DB::GetConnection ();
}
parameterised contract
{
  CreditCardPayment { CreditCardValidator::ValidateInformation,
                      DB::GetConnection }
  BankTransferPayment { DB::GetConnection }
}
```

It can be seen that the component only offers the service `CreditCardPayment` if the required service `CreditCardValidator::ValidateCardInformation` is available as it is described in the use case above. If no database is available the component is unable to offer any services any more.

## 4.2 Protocols

If the component producer specifies the interface of the component by the protocol the component provides, one has to specify (supported by existing tools) for each offered service which call *sequences* are

required for its correct execution [14]. The component specification therefore has to include the so called service effect specification which is a description of every possible control flow of a specific service call. The requires-protocol is not stated explicitly. It is being calculated dynamically out of the service effect information at configuration time as it is depending on the actual part of the provides-protocol being used by the component's clients [7]. Notice that the service effect needs to be a sequence of calls leading from a defined start state to a defined end state, e.g. if there is the need for housekeeping functions they have to be requested as well.

In general, protocols can be specified using two different approaches. One possibility is to specify all permitted sequences of service calls, the other one is to describe which sequences are prohibited. Using the first approach a description of all eligible call sequences has to be specified. The set of allowed call sequences describes the provides-protocol of the component. If the second approach is being favoured the specification needs to state under which conditions a preceding service call is prohibited.

As stated before we focus on the support of static interoperability checking. Hence, the used notation for the component protocol has to enable the deployment of efficient algorithms for checking if a given protocol is included in an other protocol. Although finite state machines are limited regarding their expression power they enable inclusion checks to be evaluated efficiently. For this reason we use finite state machines to express permitted call sequences for the rest of this paper.

There are additional reasons for preferring the specification of allowed call sequences over the specification of prohibited sequences. The reasons are summarized in the following enumeration.

1. Its easier for the producer of the component to specify the allowed order of service calls. In case of incomplete specifications, a missing allowed order is not causing harm (although it restricts the usability of the component). However, missing a prohibited sequence in a specification can lead to unexpected behaviour.

2. Tools can be used to perform automatic analysis of message sequence charts (MSCs) or workflows specified in similar languages. Further on it is possible to perform control flow analysis on the provided byte code to extract the service effect specification as mentioned below.

3. As we aim at predicting Quality of Service properties of component configurations there is the need to determine the call sequences the component performs on external component services. This information is needed to estimate the probability of a call to a specific external service.

The following is a proposal for a syntax which may be used to specify the needed FSMs. An example is depicted and specified in figure 1 where a required and a provided protocol was specified by the use of UML state charts. States are specified by their identifier and the additional information if the state is a start or a final state. Only one state is allowed to be a start state but any number of final states may be used. For complexity reasons the automaton we expect to be supplied should be deterministic. Transitions are described by their start and final state and the operation triggering the given transition.

```
FSM ::= "fsm { states { " (STATE ", ")* STATE
          " } transitions { " TRANS* " } }"
STATE ::= IDENTIFIER STARTSTATE FINALSTATE
TRANS ::= "(" IDENTIFIER "," IDENTIFIER "," IDENTIFIER")"
STARTSTATE ::= " ISSTARTSTATE" | ""
FINALSTATE ::= " ISFINALSTATE" | ""
```

A component supplier has to specify the provided component protocol as given in figure 1. Additionally one has to specify the service effect specification. We propose to use a FSM according to the given grammar for these specifications as well.

### 4.3 Quality of Service

For extra-functional properties, the application of parameterised contracts is crucial. For example, one cannot specify the timing behaviour of a software component as a fixed number. Much more, the tim-

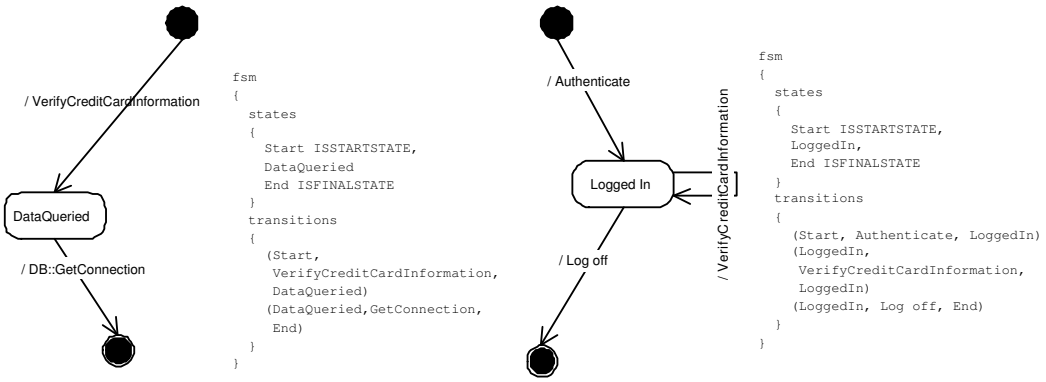


Figure 1: An example required and provided protocol

ing properties of a component offered in its provides-interface is always a function of the environment's timing behaviour, as received at its requires-interfaces. The same argument holds for reliability as empirically validated in [16]. By sequencing parameterised contracts of single components which form a component architecture (without cyclic dependencies) one can compute the overall architectural properties by sequencing the single parameterised contracts and applying them (as a function) to the properties of the underlying environmental where the system is deployed. (How to deal with multiple provides- or requires-interfaces is discussed in detail in [14, 17].)

For Quality of Service specifications we assume the usage of QML [18] at interface level. QML adds quality attributes to single services. There are two possible cases:

1. A component is depending on the services of other components. In this case the provides-interface needs to be calculated from the actual selection of the components providing the needed services. The specification should therefore provide enough information to allow the estimation of the quality attributes at the provides-interface if all required services are known.

2. A component is independent of other services or at least they are unknown at the time the system is assembled (e.g. consider a WebService, internally it might consist of further components but that is unknown or unchangeable by the configurator). In this case the needed quality attribute values need to be determined by monitoring or other techniques. Remember that some kind of reference architecture needs to be specified in this case as well to get comparable figures.

The specification in the second case is clear. It is simply an interface description in CORBA-IDL with additional QML constructs specifying the needed quality attributes. In the first case we need a specification of the service effect interface specified as FSM as described in section 4.2. Additionally we now need probabilities describing - given a certain state - which transition might be performed next. As a result the specification schema given in the previous section needs to be expanded by those probabilities which results in the following modification to the BNF given before:

```
TRANS ::= "(" IDENTIFIER "," IDENTIFIER "," IDENTIFIER
              "," PROBABILITY ")"
PROBABILITY ::= [0..1] AS DOUBLE
```

This information is sufficient for predicting accurately the components reliability [16].

# 5 Tool Support

As one can image the needed specifications for parameterised contracts might become quite large and complex but fortunately a lot of information which needs to be specified can be generated by the support of tools [19, 16].

As shown in figure 2 there are several possible sources for the specification data to result from. An important aspect of any specification is to assure the consistency between the specification and the specified object or component in our context. If the specification is generated from the used modelling diagrams or even from the code of the component it's easier to keep the specification and the actual implementation in sync which gives an important cost benefit.

It also raises the question who is responsible for adding the specification data. The utilized specification framework assumes that the component producer/vendor supplies the complete specification. In this case the component is used as black-box. Tools which are able to analyse the code of a component enable the user of the component to generate missing specifications. In this case we speak of grey-box reuse as the client uses additional information (e.g. the code) to determine needed information.

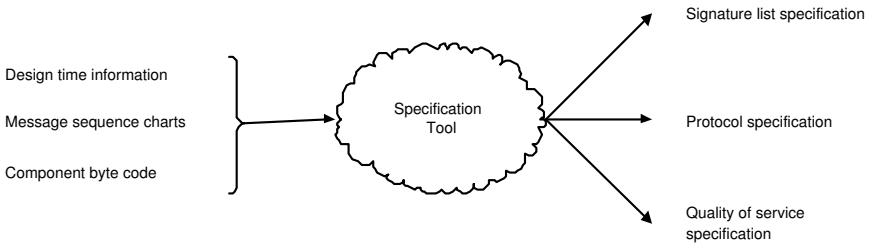


Figure 2: Tool support for the generation of parameterised contract specifications

# 6 Related Work

We propose the use of FSMs to specify protocol and QoS information. Nevertheless as this approach is limited in expression power, research has been done to find other ways of describing component behaviour. These descriptions have been expressed in different formalisms, each having specific advantages and drawbacks [20, 21, 22, 23], such as linear-timed logic (LTL) ([24, 25]) or Petri-nets [26, 27]. When considering finite call sequences, the use of (QP)LTL is equivalent to the FSM approach in term of power of specification and the analysis one can perform [28, p. 1024]. However, when transforming LTL expressions into finite state machines one has to consider the possible state explosion. The Petri-net approach is in general more powerful in modelling and the set of feasible analyses differs from finite state machines. Successful research was also undertaken to make the state-machine approach more powerful without loosing its possibilities of efficient analyses [29]. However, in general there is a trade-off between a formalism's expression power and the set of feasible analyses which can be performed within this formalism.

# 7 Conclusions

In this paper we focused on explaining what needs to be specified to add parameterised contracts to an existing specification framework. We distinguished between three levels of specification data whereby higher levels contain lower ones. On the lowest level we introduced signature list based specifications. In a second step we added behavioural information by specifying protocols with finite state machines.

On the highest level Quality of Service attributes were added to the component specification. Finally, the importance of tools in the highlighted context was emphasised.

Future work will be directed to the specification of additional Quality of Service attributes like safety, fairness or liveness. The specification data will then also be used to predict the corresponding properties of complete component architectures.

# References


[1] The UDDI standardization consortium, "The UDDI homepage." http://www.uddi.org.

[2] J. Ackermann, F. Brinkop, S. Conrad, P. Fettke, A. Frick, E. Glistau, H. Jaekel, O. Kotlar, P. Loos, H. Mrech, E. Ortner, U. Raape, S. Overhage, S. Sahm, A. Schmieten, T. Teschke, and K. Turowski, "Standardized specification of business components," *Memorandum of the working group 5.10.3, Component Oriented Business Application Systems*, 2002.

[3] Object Management Group (OMG), "The CORBA homepage." http://www.corba.org.

[4] R. H. Reussner and H. W. Schmidt, "Using Parameterised Contracts to Predict Properties of Component Based Software Architectures," in *Workshop On Component-Based Software Engineering (in association with 9th IEEE Conference and Workshops on Engineering of Computer-Based Systems), Lund, Sweden, 2002* (I. Crnkovic, S. Larsson, and J. Stafford, eds.), Apr. 2002.

[5] R. H. Reussner, "Contracts and quality attributes of software components," in *Proceedings of the Eigth International Workshop on Component-Oriented Programming (WCOP'03)* (W. Weck, J. Bosch, and C. Szyperski, eds.), June 2003.

[6] R. H. Reussner, "Parameterised Contracts for Software-Component Protocols." Presentation given at Oberon Microsystems, Zürich, http://liinwww.ira.uka.de/∼reussner/zuerich00.ps.gz, Dec. 2000.

[7] R. H. Reussner, I. H. Poernomo, and H. W. Schmidt, "Reasoning on software architectures with contractually specified components," in *Component-Based Software Quality: Methods and Techniques* (A. Cechich, M. Piattini, and A. Vallecillo, eds.), no. 2693 in LNCS, pp. 287–325, Springer-Verlag, Berlin, Germany, 2003.

[8] N. Wirth, *Programming in MODULA-2*. Springer-Verlag, 3rd Edition, 1985.

[9] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer, "Specifying distributed software architectures," in *Proceedings of ESEC '95 - 5th European Software Engineering Conference*, vol. 989 of *Lecture Notes in Computer Science*, (Sitges, Spain), pp. 137–153, Springer-Verlag, Berlin, Germany, 25–28 Sept. 1995.

[10] B. Meyer, "Applying "design by contract"," *IEEE Computer*, vol. 25, pp. 40–51, Oct. 1992.

[11] B. Meyer, *Object-Oriented Software Construction*. Prentice Hall, Englewood Cliffs, NJ, USA, 2 ed., 1997.

[12] B. Krämer, "Synchronization constraints in object interfaces," in *Information Systems Interoperability* (B. Krämer, M. P. Papazoglou, and H. W. Schnmidt, eds.), pp. 111–141, Taunton, England: Research Studies Press, 1998.

[13] A. Vallecillo, J. Hernández, and J. Troya, "Object interoperability," in *Object Oriented Technology – ECOOP '99 Workshop Reader* (A. Moreira and S. Demeyer, eds.), no. 1743 in LNCS, pp. 1–21, Springer-Verlag, Berlin, Germany, 1999.

[14] R. H. Reussner, *Parametrisierte Verträge zur Protokolladaption bei Software-Komponenten*. Logos Verlag, Berlin, 2001.

[15] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins, "Making components contract aware," *Computer*, vol. 32, pp. 38–45, July 1999.

[16] R. H. Reussner, H. W. Schmidt, and I. Poernomo, "Reliability prediction for component-based software architectures," *accepted at Journal of Systems and Software – Special Issue of Software Architecture - Engineering Quality Attributes*, 2002.

[17] H. W. Schmidt and R. H. Reussner, "Generating Adapters for Concurrent Component Protocol Synchronisation," in *Proceedings of the Fifth IFIP International conference on Formal Methods for Open Object-based Distributed Systems*, Mar. 2002.

[18] S. Frolund and J. Koistinen, "Quality-of-service specification in distributed object systems," Tech. Rep. HPL-98-159, Hewlett Packard, Software Technology Laboratory, Sept. 1998.

[19] G. Hunzelmann, "Generierung von Protokollinformation für Softwarekomponentenschnittstellen aus annotiertem Java-Code," diplomarbeit, Fakultät für Informatik, Universität Karlsruhe (TH), Germany, Apr. 2001.

[20] R. Campbell and N. Habermann, "The Specification of Process Synchronization by Path Expressions," in *Proc. Int. Symp. on Operating Systems*, vol. 16 of *Lecture Notes in Computer Science*, pp. 89–102, Springer-Verlag, Berlin, Germany, 1974.

[21] O. Nierstrasz, "Regular types for active objects," in *Proceedings of the 8th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-93)*, vol. 28, 10 of *ACM SIGPLAN Notices*, pp. 1–15, Oct. 1993.

[22] D. Yellin and R. Strom, "Protocol Specifications and Component Adaptors," *ACM Transactions on Programming Languages and Systems*, vol. 19, no. 2, pp. 292–333, 1997.

[23] R. H. Reussner, "Enhanced component interfaces to support dynamic adaption and extension," in *34th Hawaiin International Conference on System Sciences*, IEEE, Jan. 3–5 2001.

[24] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, New York, USA, 1992.

[25] J. Han, "Temporal logic based specification of component interaction protocols," in *Proccedings of the 2nd Workshop of Object Interoperability at ECOOP 2000*, (Cannes, France), June 12.–16. 2000.

[26] C. A. Petri, "Fundamentals of a theory of asynchronous information flow," in *Information Processing 62*, pp. 386–391, IFIP, North-Holland, 1962.

[27] C. Ling and H. W. Schmidt, "A concept of time in workflow modelling and analysis.," Tech. Rep. 2000/72, School of Computer Science and Software Engineering, Monash University, VIC 3168 Australia, June 2000.

[28] J. van Leeuwen, *Formal Models and Semantics, Handbook of Theoretical Computer Science*, vol. 2. Amsterdam, The Netherlands: Elsevier Science Publishers, 1990.

[29] R. H. Reussner, *Parametrisierte Verträge zur Protokolladaption bei Software-Komponenten*. Dissertation, Fakultät für Informatik, Universität Karlsruhe (TH), Germany, July 2001.

*Position Statement*

# On Practical Component Acceptance Testing

Hans-Ludwig Hausen

Fraunhofer, Schloss Birlinghoven, Sankt Augustin, Germany
hausen@gmd.de

**Abstract.** Component evaluation and acceptance testing is considered as a critical task comprising: analysis of condition and constrains (result: evaluation requirements), specification of the evaluation (result: identification of relevant component items and associated component attributes), design of the evaluation (result: evaluation plan), conducting the evaluation (result: test and measurement reports) and finally reporting on the evaluation (result: compilation of all intermediate results). Complementary, the concept of evaluation module is introduced to allow a well-structured description of mature evaluation techniques (e.g. inspection, testing, reliability modelling) as well as the definition of their interaction, the latter being necessary to instrument an acceptance testing method by a set of coherent evaluation techniques.

## 1 Introduction

For many applications, most code is not devoted to implementing the primary input-output functionality but instead addresses other concerns, such as reliability, availability, responsiveness, performance, security, and manageability. Conventional programming practice requires the programmer to keep all these other ilities in mind while coding and to explicitly invoke behavior at exactly the right places to achieve them. Therefore, component quality evaluation and acceptance testing is identified by industry as an important issue for component development, distribution and application [Raea95].

A component evaluation scheme needs to be capable of dealing with any type of components. Such components range from *off-the-shelf* component developed for a general customer base, through projects commissioned by a single customer, to embedded components in systems. The FIG. 1 illustrates this characterisation of components and shows the parties having a direct interest in the acceptance testing.

Such a scheme must be of value both to the producers, sellers, and users of a componet and to the community-at-large. It must have stability and must be trusted by all. Therefore, it must be regulated, consistent, understandable, cost effective and respected. Any scheme must be flexible, evolutionary and capable of rapid response to change. There will also be the need to harmonise the scheme with any changes in the law or of standards or regulations which impinge on components and their use.

A practical approach should comprise: analysis of condition and constrains (result: evaluation requirements), specification of the evaluation (result: identification of relevant component items and associated component attribu-tes), design of the evaluation (result: evaluation plan), conducting the evaluation (result: test and measurement reports) and finally reporting on the evalua-tion (result: compilation of all intermediate results). Complementary, the concept of evaluation module is adopted which allows a well-structured description of evaluation techniques (e.g. inspection, testing, re-liability modelling) as well as the definition of their interaction, the latter being necessary to instrument the method by a set of coherent evaluation techniques.

In the following sections the methods and tools for the evaluation and assessment of components and software processes are discussed in detail. Particular emphasis is given to the identification and selection of component characteristics and metrics as well as to the handling of evaluation methods and tools. In a situation where we have a huge amount of software metrics, the problem of identifying the right one and applying it correctly is an important issue.

## 2 Specified Versus Actual Service

The basic task of a component evaluation process is to check for evidence that the actual service exhibited by a component is a trusted instantiation of the specified service. In other words, *Has the component been made correctly?* has to be answered. on the other hand, a component evaluation cannot ensure a correspondence between expected service (from the user's perception) and actual service - i.e.*Is it the correct component?*

A *formal* procedure like acceptance testing, therefore, must be based upon the assurance of a formalized description of the component. Usually, this will be contained in documents such as a requirements specification, user's manuals and instructions, design specifications etc. The main point is that the description is in a published format which does not change (at least for the period of the acceptance certificate) and can be used as a reference. It does not matter that the component does not represent the intentions of the producer correctly. The acceptance tester's job is to assure that the component is, and behaves, as it is described.

This view does, of course, present problems for users who may be expecting that a certificate will be an assurance that a component will meet their expectations. But this can only be true when the component fully meets its specifications and the specifications correspond to the expectations. Unfortunately, user expectations frequently are

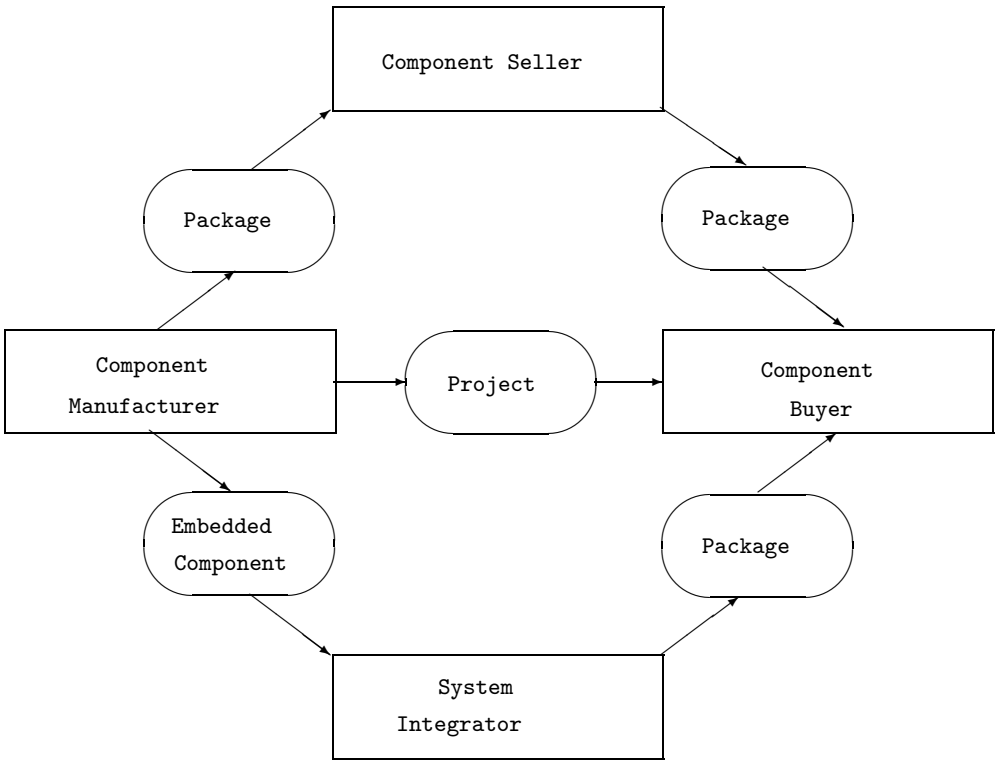


**Fig. 1.** Component Views and Interests

informal, subjective and go beyond the written descriptions (specifications) of the component (see respective discussin in [Raea95]).

The key relationships are that it must be the user's responsibility to ensure that the choice of component corresponds to his or her needs and the producer's responsibility to ensure that a component corresponds to its description.

There is one exception to the principle that the certificate should only represent correspondence between actual and specified service. That exception is where some omission or ambiguity in specification results in a component which has a potential failure mode which may be hazardous to the user or third parties. In this case, it might be expected that it was a responsibility of the certifier to identify such anomalies during the analysis of the component.

These considerations lead to two views of acceptance testing. There must exist a fundamental or BASIC acceptance testing which can assure conformance of the actual service of a (software) component with its specified service. Where it is necessary also to assure that the component behaves both reasonably and that its use carries no unacceptable risk, a BASIC acceptance testing can be extended. EXTENDED acceptance testing would include a more rigorous analysis of the component and the additional tests, evaluations or audits which were found necessary to assure, for instance, safety in use.

It should be noted that there is no difference in the method between BASIC and EXTENDED acceptance testing. One is simply an extension of the other. However, some of the issues raised by extended acceptance testing (such as *is the component fit for its intended purpose?*) pose considerable difficulties.

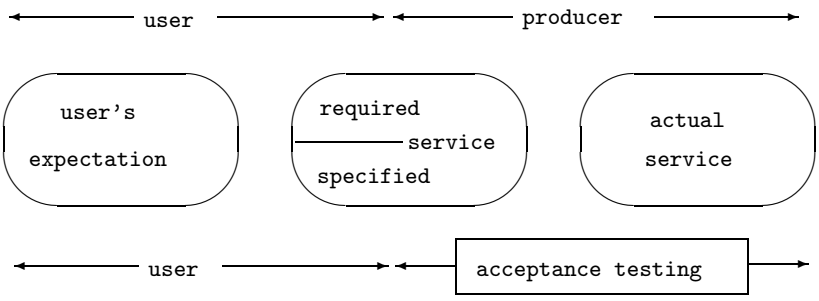


**Fig. 2.** expectations, required, specified, and actual service

## 3 The Certification Approach

It is assumed that the technology of software validation, verification, testing, and meassurementdeveloped by the research community is ready to use for software acceptance testing. In consequence, our strategy is to use existing techniques to build a consistent framework and to rationalise their application. The acceptance testing framework is in two parts: the evaluation and acceptance testing model that precisely defines the notions used, and the evaluation method that describes the various steps leading to the acceptance testing of a component.

The evaluation and acceptance testing model is based on four formal sub-models which define notions introduced by the evaluation method: a component model, a software development process model, a software characteristics model and a measurement model. These models are under continuous review as the project proceeds and empirical results are fed back from the Case Studies. The component model is aimed at defining a component which is submitted for an evaluation process. The definition consists of two steps, the identification of software parts and the classification of software parts. The software development process model identifies items of process evidence that may be useful to facilitate component measurements. In a sense, this approach tries to reconcile the two ideas of *process* acceptance testing and *component* acceptance testing.

The software characteristics model is the kernel of the evaluation and acceptance testing model. It defines the characteristics that will be assessed in a component, i.e. the acceptance testing attributes. To be of any value, the characteristics composing this model must correspond to the public notion of software quality. The major issue in producing this model is to be able to define the characteristics unambiguously. The measurement model is more difficult. One problem is to deal with the complexity of measurement and evaluation techniques. The number of applicable measures proposed in the literature is extremely large and their conditions of application vary enormously. The approach taken by the measurement model is modular; a small set of evaluation techniques and tools, that can be mastered by a specialist, is encapsulated in what is called an *evaluation module*.

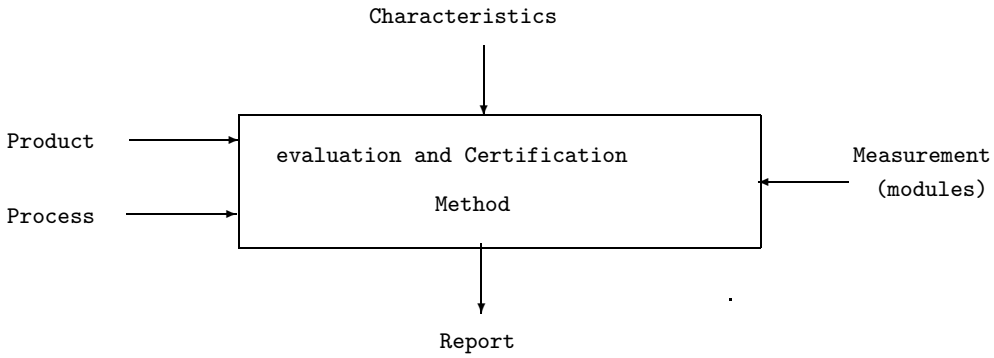


**Fig. 3.** Input and Output of the evaluation and Certification

The evaluation and acceptance testing method is composed of the following steps:

Identification, in the component submitted for acceptance testing, of the recognised component parts and elements of development process evidence.

Specification of the software characteristics that are to be assessed and the conformance evaluation level. The specification is based on an analysis of the descriptive component parts and will include the pass/fail criteria for the attributes of interest.

Selection and application of evaluation modules. The choice depends on the characteristics to be assessed, the available component parts and process evidence and the applicable techniques. The result of applying the evaluation modules will be a set of measurements which can be judged for conformance with the evaluation specification.

Reporting the evaluation results and evaluation of the results against the evaluation specification. This report will be the basis of the award (or refusal) of a Certificate.

## 4 The Evaluation Procedure

What follows is a description of one possible procedure which might be refined into a scheme for assessing components. The primary reference for acceptance testing should be a acceptance testing norm, or better an internationally aggreed standard, providing details of the basis for acceptance testing, level of achievement and generally what must be done to secure a certificate. The norm should be supplemented by authoritative guides which will explain, in precise terms, exactly how the evaluation method should be applied, the attributes and characteristics that should be examined, the evaluation modules available for use and the evaluation criteria. (see respective discussion in [Raea95])

Evaluation might be undertaken by accredited test laboratories spread throughout the community. Then a certificate that is awarded will be valid anywhere in the community, irrespective of where the evaluation was done. It is

crucial to the success of the scheme that application of the evaluation method by any test laboratory to a component produces consistent results.

There are a number of distinct stages in the evaluation of a component. In Fig. 4 we provide a block diagram of the evaluation process.

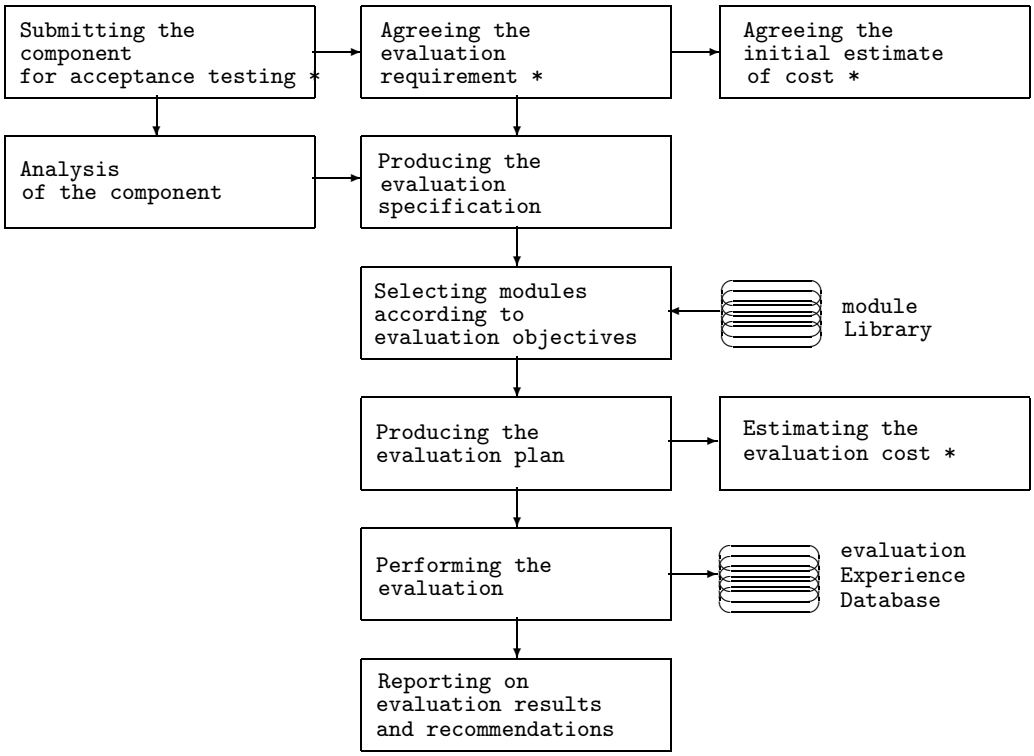


**Fig. 4.** Stages of an evaluation Procedure

### 4.1 Submitting the component for acceptance testing *

The decision as to whether a component should be submitted for acceptance testing should be taken as early as possible in the life cycle of the component. If this is done right at the beginning of the design and development stages, it should be possible to 'build in' to the development, the checks and tests which the component will have to pass while it is being assessed by the Certification Agency. This should ensure the maximum chance of the component passing the evaluation, as well as minimising the chance of extra, unexpected costs being incurred.

Early contact with the Test Laboratory to discuss the intention of submitting a component for evaluation will also help the producer to anticipate any special needs (such as particular documents or evidence which might be required) which the assessors may have. It may be that some (or even all) of the evaluation tests will have to be done 'on site', rather than at the Test Laboratory. In these cases, the tests will still be controlled by the Test Laboratory staff to ensure that the results are unbiassed. For very large, complex software projects, it may be beneficial for the producer to have a continuous, detailed association with the assessors during the whole of the development to minimise the duration and cost of the evaluation process.

When the intention to submit a component for acceptance testing is declared, (usually the decision of the producer) it will be the first task of the assessors to ensure that a minimum set of component parts (documents, code and test results) will be available to allow an evaluation of conformance to be made.

The size of the 'minimum' list will vary according to the particular circumstances of the proposed evaluation. It may be very short, such as the program object code and the user manual. These two component parts would enable an evaluation to be made as to whether the program did the things that were stated in the manual.

On the other hand, a large, complex component may need to be supported by a substantial amount of documentation, ranging from user requirements, through functional and design specification, code listings, quality and test plans, test results - to name but a few!

Extended evaluation will also require additional component parts such as risk and safety analyses and the results from beta testing.

### 4.2 Agreeing the evaluation requirement *

In the majority of cases, it is expected that the producer will bear the cost of acceptance testing. This will give the producer some rights as to how extensive the coverage of the certificate should be and how much the process should cost. It will be necessary, therefore, for the test laboratory and the producer to agree which software characteristics are to be assessed, the level to which they are to be assured and whether basic or extended acceptance testing is sought.

The formal record of this agreement of what will be covered in the evaluation process will be known as the evaluation REQUIREMENT. It will provide a nominal list of attributes (features, characteristics) which are to be assessed and identify the sources of data and evidence which can be used in the evaluation process.

### 4.3 Agreeing an initial estimate of cost *

Once the evaluation requirement is agreed, an initial cost estimate can be constructed from a knowledge of the list of items in the requirement and the work to be done.

The input to this step is the characteristics of the component which are to be assessed and the agreement on the type and level of acceptance testing. At this stage, the component has not been analysed so a detailed knowledge of the content and quality of the component (documentation, manuals, source code etc) is not available. Only the application area and a few rough measures such as the number of documentation pages and the number of codelines and the programming language are known. Therefore, the cost estimate can only be based on the agreed acceptance testing level and the size of the component code and any previous experience of assessing similar components. However, it should be possible to provide a reasonably accurate cost estimate for the work needed to progressing to the componention of the evaluation Plan (stage 6).

### 4.4 Analysis of the component

It is necessary to perform an analysis of the component submitted to evaluation in order to identify the various component parts and elements of process evidence it consists of. This information is needed in order to identify which evaluation can be performed. This will be used, together with the evaluation requirements, to build the evaluation specification.

The analysis of the component consists of two phases: (i) identification of available documents, and (ii) classification of the information contained on the component and process models.

**(i) Identification of submitted documents** The component submitted to evaluation consists of documents, which includes code. The first step of the component analysis consists of making a list of these documents, together with the identification of their claimed characteristics. For each document, the following information should be provided:

- title
- formalism (natural language, programming language, ...)
- claimed conformity to standard (optional, reference should be provided - language standards or development method standards should be considered)
- size (to be used for costing process)

**(iia) Classification of submitted component information** The information contained in component documents belongs to the following categories:

- required service information
- specified service information
- actual service information

For each of the services, the information can be sub-classified in:

- code; data-flow, control-flow, states trace
- annotations

Of course, it is clear that most of the components under evaluation are nor composed of documents falling strictly in the categories identified above. Some component documents contain information belonging to several classes, while the same type of information may be spread amongst several documents.

**(iib) Classification of submitted process information** In order to support the evaluation of a component, the sponsor may submit documents concerning the component development process. When evaluation of these documents is to be performed, the information contained in them must be identified, so that it can be used. The information concerning the process might be classified into:

- project handbook
- quality plan
- quality reports

Some other process information might be required depending on the objectives of the evaluation and acceptance testing.

### 4.5 Producing the evaluation specification

A list of attributes which need to be assessed will have been derived from the evaluation Requirement, and possibly modified by the findings of the component analysis. This list represents the necessary compromise between the characteristics of the component the producer feels that it is important to assess and the more comprehensive list of characteristics that an assessor might feel to be the appropriate set of attributes which should be assessed.

The list is the basis for the evaluation SPECIFICATION which should cover:

(i) characteristics which are to be assessed
(ii) sub-characteristics, which can be decomposed from the primary list (of characteristics), which will provide a link to actual measures
(iii) a list of measurements which can be used to assess the conformance of sub-characteristics (and ultimately characteristics) with requirements, specifications, standards and legal needs
(iv) target values for the measurements being made which will indicate whether conformance criteria are being met (pass / fail)
(v) class of documentation required (process or component)

The evaluation specification need not be concerned as to whether any attribute can or cannot be measured, ie - whether particular modules are available.

Neither should the specification be totally influenced by the wishes of the producer expressed in the evaluation requirement. If, at the component analysis stage, some aspect of the component is suspect and is deemed necessary for investigation, then this should be included in the evaluation specification. It will, of course, still be the right of the producer to withdraw the component from acceptance testing if the additional costs of the more extensive evaluation are not agreed.

### 4.6 Selecting evaluation modules according to evaluation objectives

The input to this stage will be the evaluation Specification. The objective is to attach to each of the bottom level acceptance testing attributes one or more measurement techniques or 'modules'. The output from this stage will, therefore, be a list of modules which are to be applied to perform the evaluation of each attribute.

The selection process itself requires that the module library be searched for modules which will be useful in the evaluation of each attribute for the target component. This directly implies two criteria to be applied in the selection of modules: First, the module must be known (and proven) to be useful in the evaluation of the attribute it is to be used for. Secondly, the module must be applicable to the component part it is to be used on. For example, many 3GL software metrics cannot be applied to object oriented or rule based software.

### 4.7 Producing the evaluation plan

In the previous stage, possible evaluation modules have been identified and associated with component parts. However, this set of modules may not be optimal for carrying out the evaluation. Some modules may be redundant and some modules may be missing. It must be decided whether new modules must be developed or whether missing modules can be substituted by a combination of existing modules. The purpose of this step is to make the final planning of modules for the evaluation. The planning will be done in order to optimize the coverage of evaluation and the cost of carrying through the evaluation.

### 4.8 Estimating the evaluation cost *

The evaluation plan includes the list of modules to be applied. Each module includes information from which the cost of its application can be derived. Hence, it is easy to calculate the total cost of the evaluation. However, the test laboratories act in a competitive market and therefore the actual price of the evaluation may differ substantially from the calculated cost. Furthermore, in some cases, the cost may also include the whole, or part of the development cost of a new module.

### 4.9 Performing the evaluation

The implementation of the evaluation plan means applying the modules on the related component parts and collecting for each of them the application results. The output will be a collection of measurement reports resulting from the application of the modules.

This step consists in:

**(i) planning and managing the evaluation project**

These are the usual activities to be performed at the beginning and during a project, namely: identifying the evaluation activities, identifying resources (human, tools, ...), allocating resources ro activities and scheduling, reporting progress.

**(ii) performing the measurements on the target component parts**

Measurements can be manual, computer aided (eg, using a check list manager for applying check lists), or automatic (eg, measuring the cyclomatic number in a source code component using a static analyser).

The main task is to collect the measurement result and also to keep any information (measurement data) about the measured component part, that could be helpful for pass/fail decision to be taken.

These data can be figures, diagrams, parts of documentation etc.

**(iii) producing the measurement reports**

This consists, for all modules, in collecting and synthesising measurement data and results in order to produce the report resulting from the application of the module. The structure of this report is pre-defined in the module document.

### 4.10 Reporting on evaluation results and recommendations

**(i) Making the decision**

The actual module application results contained in the measurement reports, are to be compared to expected results specified in the evaluation specifications. Technical expertise might be required to make the pass/fail decision. This expertise has been gained from previous experience in component evaluation and from experience in the industrial sector the component belongs to.

**(ii) General reporting and recommendations**

The general report is a synthesis of all measurement reports. If the decision is *pass* for all modules, this report should only recapitulate the evaluation specifications, and the actual results.

If the decision is *fail*, the measurement result that drove this decision should be highlighted. Recommendations can be made to the sponsor of the evaluation in the sense of improvement of the submitted component.

In any case, all documents produced during the evaluation, measurement results and data, should be referenced in this document in order to be able to control the correct application of evaluation procedures and the suitability of decisions taken.

**(iii) Capitalising of the experience**

Experience gained from the current evaluation process must be stored. Running a component evaluation provides information that can help to precise the cost of module application, to improve module documents, to identify needs for new evaluation techniques.

This step only concerns the acceptance testing laboratory.

## 5 The Evaluation Characteristics

The evaluation characteristics represent a selection of properties or attributes of a component. The evaluation process measures and assesses these attributes and the certificate is a statement of the extent to which the attributes are present in the component.

The choice of evaluation attributes is important for the acceptance of the idea of software acceptance testing. Many attributes can be suggested and many pros and cons can be given for each. However, a number of basic requirements can be formulated and will be helpful for the selection process. (see respective discussion in [Raea95])

The evaluation attributes must be relevant for the user (buyer) of the component, i.e. they must tell something important about the software. The evaluation attributes must be unambiguosly defined and intuitively easy to understand; i.e. they must be meaningful to the users. The evaluation attributes must be measurable and measures must be reproducible, i.e. the evaluation must be based on scientific principles. Workmanship, Correctness, Reliability and Efficiency might be considered as core criteria, but also the whole Model or McCall[McCa77] might be necessary.

Internationally relevant is the quality standard ISO/IEC 9126 [ISO9126]. Here we get following six characteristics:

FUNCTIONALITY: a set of attributes that bear on the existence of a set of functions and their specified properties. The functions are those that satisfy stated or implied needs.

RELIABILITY: a set of attributes that bear on the capability of software to maintain its level of performance under stated conditions for a stated period of time.

USABILITY a set of attributes that bear on the effort for use and on the individual evaluation of such use by a stated or implied set of users.

EFFICIENCY a set of attributes that bear on the relationship between the level of performance of the software and the amount of resources used, under stated conditions.

MAINTAINABILITY a set of attributes that bear on the effort needed to make specified modifications.

PORTABILITY a set of attributes that bear on the ability of software to be transferred from one environment to another.

A further decomposition of these six characteristics to sub-characteristics and metrics is given in the quality model section of [Raea95]

## 6 The Evaluation Modules

The quality model describes how measurement and evaluation techniques are organised. The goal is to allow an effective selection and application of these techniques. The state of the art relative to them is extremely large. In order to cope with this complexity, the proposed Measurement Model is modular: A set of complementary measurement and evaluation techniques are encapsulated in an element that is called an *evaluation module*.

An evaluation module can be considered as a black-box that when suitable inputs are applied, returns one or more measurements which can be used to assess the conformance of a component attribute with its specification. An

evaluation module should be independent of component type, unambiguously defined and produce consistent results. The format and detail necessary for describing a module is given in appendix A.

The implementation of a module, in a concrete case, is a procedure that takes as input elements of the component or evidence from the process and produces a value on a specified scale (nominal, ordinal, interval, ratio or absolute) and a report providing information on the practical module application.

The evaluation of a component (in order to certify it) is performed by applying several evaluation modules. The global pass/fail decision is obtained by evaluating the measurements from the modules with respect to the component and evaluation specifications.

A worthwhile way of considering a module is that it primarily measures a property of a given component. This can be compared to measuring physical characteristics such as length, humidity, electric current etc.

The evaluation criteria, however, are not only physical. They are derived also from technical, legal or commercial considerations. The selected evaluation attributes strongly influence the set of necessary evaluation modules. Obviously, each module must be associated with at least one attribute and conversely each attribute must be assessed by modules. The choice of modules for assessing a particular piece of software will be based on the type of available information. For example, the programming language used may restrict the number of available tools for static analysis.

On the other hand, a module might not be explicitly related to any higher level quality characteristics. Its result may be used in the evaluation of various characteristics. However, the definition of modules using other module results will provide some structuring elements.

It may be that no complete set of modules exists that would completely assess to the required level for a particular component. In this situation there may be a case for constructing a new module or rejecting acceptance testing of the component.

## 7 Summary and Conclusions

In summary the Evaluation Procedure presented is

o adaptable to circumstances of any testing laboratory (available personnel, evaluation methods and tools), and

o flexible to relevant standards of component type dependent quality requirements and of software engineering processes, and to legal or contract issues.

One prime objective for proposing the acceptance testing procedure is to ensure that an evaluation process is pracmatic and effective. In order to achieve this goal it is necessary to reflect

- experience with evaluation modules and the module library,
- appropriateness of levels and software characteristics,
- appropriateness of component representation,
- appropriateness of process representation,
- calculation of actual costs in order to improve cost estimates,
- appropriateness of the evaluation method.

Running a component evaluation provides information that might help to estimate the cost of an evaluation module application, to improve module documents, to identify needs for new evaluation techniques. The experience gained should be stored in a data base in order to make them available for further investigations which could lead to an improvement of evaluation procedures or particular techniques.

## References


[—] References and further Readings.

[Abr00] Pekka Abrahamsson and Timo Jokela *Development of Management Commitment to Software Process Improvement* Proceedings of IRIS 23. Laboratorium for Interaction Technology, University of Trollhttan Uddevalla, 2000. L. Svensson, U. Snis, C. Srensen, H. Fgerlind, T. Lindroth, M. Magnusson, C. stlund (eds.)

[conr00] Maria Letizia Jaccheri and Reidar Conradi and Bard H. Dyrnes, *Software Process Technology and Software Organisations*, "European Workshop on Software Process Technology", p96-108,2000,

[Cang01] Joao W. Cangussu, *Modeling and Controlling the Software Test Process*, IEEE, International Conference on Software Engineering 2001, p 787-788

[Ghez98] Gianpaolo Cugola and Carlo Ghezzi, *Software processes: a retrospective and a path to the future* journal Software Process: Improvement and Practice, vol4 no3 p.1001-123 1998

[ISO9126] Draft international standard ISO/IEC 9126, *Information technology - Software component evaluation - Quality characteristics and guidelines for their use*, ISO International Organization for Standardization, International Electrotechnical Commission, 1996

[HLH92a] H.L.Hausen, A Software Assessment an Certification Advisor, in: J.P. Agrawal, V. Kumar, V. Wallentine (eds.), ACM, CSC'92 20th Annual Computer Science Conference, Kansas City, March 3-5, 1992

[McCa77] McCall, J.A.; Richards, P.K.; Walters, G.F. *Concepts and Definitions of Software Quality* Factors in Software Quality, Vol. 1, Springfield, Va.: NTIS, November 1977.

[Raea95] A. Rae, P. Robert, H.L. Hausen *Software Evaluation for Certification* McGraw Hill, London 1995

[RoMa89] Rombach, H. Dieter; Mark, Leo *Software Process & Product Specifications: A Basis for Generating Customized Software Engineering Information Bases*, Proceedings of the 22nd Annual Hawaii International Conference on System Sciences, Vol. II - Software Track, pp. 165-174, IEEE, 1989.

# Annotation of Component Specifications with Modular Analysis Models for Safety Properties

Lars Grunske[1]

[1]Department of Software Engineering and Quality Management
Hasso-Plattner-Institute for Software Systems Engineering at the University of Potsdam
Prof.-Dr.-Helmert Strasse 2-3, D-14482 Potsdam (Germany)
+49(0)3315509152
lars.grunske@hpi.uni-potsdam.de

**Abstract.** The application of component based software engineering techniques in safety critical technical systems has increased due to economic reasons. This leads to the problem how to analyze the safety properties, because the failure types and their probabilities of especially COTS-components are potentially unknown. We propose to annotate components with encapsulated fault trees and basic failure probabilities. Based on this information and the structure specification an automated safety analysis is possible.

## 1 Introduction

In order to analyze the safety properties of a system the probability of a caused hazard must be determined. Therefore, general fault trees are used [2,7,10,12,14,15]. These fault trees are logic formulas [10] that model the interrelationships between a potential system hazard and the basic faults that cause this hazard. This paper deals with the problem that basic faults and their probabilities are often unknown, especially for COTS-components. Thus, we propose to annotate each component in safety critical systems with a fault tree that models the failure behaviour of this component. Based on these fault trees a model-based evaluation of the safety properties is possible [6,14,15].

The remaining part of this paper is organized as follows: section 2 introduces notations for the structure and interface specification of component-based software architectures. These notations serve as the basis for the construction and evaluation of encapsulated fault trees. In section 3 we present encapsulated fault trees and a methodology to annotate COTS-components. Based on these annotations an automated safety analysis is described. In section 4 we illustrate the feasibility of the methodology through an example that models a level crossing control system. Finally, we outline our conclusions and point out the directions for future work in section 5.

## 2 Architecture Specification

The architecture specification is the first simplified model of a system under development [1]. It consists of the structure specification and a set of interface specifications.

### 2.1 Structure Specification

The structure specification is the basic construction plan of a software system. It describes how the system is decomposed into smaller components and which of them interact during the runtime of the system. These structure specifications are basically divided into component-models and component-connector-models [7,9]
Due to the popularity in industrial projects we use simple component-models for the structure specification [1,17,18]. They allow for the description of the software structure in terms of communicating components, which are also referred to as capsules [17] or as actors [9]. These components are concurrent objects specified by component-classes. A component-class specification models either a flat software component that cannot be refined further or a composite of finer, more granular components. This leads to a recursive definition of component-classes that are modeled by a composition hierarchy, in which the top-level component describes the entire system.
For the communication with its environment a component utilizes interface objects called ports. Between these ports point-to-point connections can be established that are used to send messages. If a message is sent directly to a component, the receiving port is called an end-port. To communicate with a component inside a hierarchical component special ports are used to forward a message from the outside of a composite component to an inner component. These ports are called relay ports.
The simplified meta-model of a structure specification with a component-model is presented in figure 1.

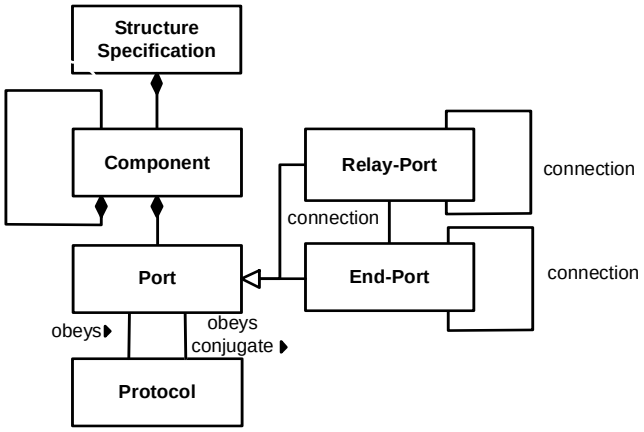


**Fig. 1.** Meta-model for a structure specification

### 2.2 Interface Specification

Interface specifications are used to model the black-box behaviour of components. More precisely, they specify a protocol with a set of valid message sequences [13]. Interface automata [5] may serve as a notation for an interface specification. They describe the causal order of messages or actions that are sent to or by the component. Interface automata are formally defined as follows [5]:

**Definition 1: Interface automata**
An interface automaton is a 6-tuple $\langle S, S^{init}, E^I, E^O, E^H, T \rangle$, where:

- $S$ is a set of states
- $S^{init} \subseteq S$ is a set of initial states, with $S^{init} \neq \emptyset$
- $E^I$, $E^O$ and $E^H$ are mutually disjoint sets of input, output or internal actions.
- $T \subseteq S \times E \times S$ is a set of steps, where an action $a \in E^I$, $a \in E^O$ or $a \in E^H$ is enabled in the state $v$ if $\langle v, a, v' \rangle \in T$. If $a$ occurs the next state is $v'$. Based on the action type the step $\langle v, a, v' \rangle$ is called input, output, or internal step.

For the specification of interface automata a typical graphical automaton notation [8] is used, where input, output, or internal actions are denoted with the postfix symbols ?,! or ;. An example for an interface specification is given in figure 2. This component can send a message *y?* to the environment and waits to receive the message *x!*.

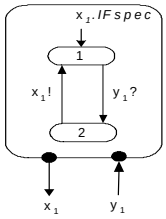


**Fig. 2.** Interface specification with interface automata

If two components interact via a point-to-point connection, they must be compatible. Therefore, a proof algorithm to check the compatibility of two interface automata is presented in [3].
This proof algorithm determines first the set of shared actions between the two automata. Based on this set the two interface automata are compatible if none of the automata may produce an output action that is not accepted as an input action of the other automaton. To prove this the product automaton is constructed and it is checked that it does not contain illegal states as described above.

## 3 Evaluation of Safety Properties with Encapsulated Fault Trees

The basic idea to allow the evaluation of safety properties is to annotate each component in an architecture specification with an encapsulated fault tree. Such an encapsulated fault tree describes the failure behaviour of the component [11]. It contains a set of outputs called output failure ports which define all concrete failure types that can be caused by the component. The output failures can be further caused either by an internal fault or by an external failure of the environment or another component. The external failures are specified with a set of input failure ports. The internal structure of an encapsulated fault tree is specified similar to normal fault trees [10] as a Boolean function.

To evaluate the safety properties of the developed system the encapsulated fault trees of the contained components must be embedded in the encapsulated fault tree of the system [11]. Further, if the components exchange messages that can cause a

failure the input and output failure ports of the two encapsulated fault trees must be connected.

In the following section, we first introduce the formal concept of an encapsulated fault tree. Then we propose an algorithm for the construction of an encapsulated fault tree based on the interface specification of hierarchical components. This algorithm connects the input and output failure ports based on the architecture specification.

### 3.1 Formal Definition of an Encapsulated Fault Tree

An encapsulated fault tree is a hierarchical directed acyclic graph [19] that can be defined as follows:

**Definition 2: Encapsulated Fault Tree**

An encapsulated fault tree $EFT$ is a hierarchical directed acyclic graph (Digraph) that is described with the tuple $\langle N, P, E \rangle$, where:

- $N$ is a set of nodes partitioned into internal failure events $N_{intern}$, input failure ports $N_{in}$, and output failure ports $N_{out}$
- $P$ is a set of proxies that are partitioned into gate proxies $P_G$ and sub-component proxies $P_{SC}$. These fault tree components are specified with a tuple $\langle N_{in}^{extern}, N_{out}^{extern}, CTS \rangle$, where:
  - $N_{in}^{extern}$ and $N_{out}^{extern}$ are a set of external input and external output failure ports
  - $CTS$ is an assignment function which assigns to a proxy a logic formula if $P \in P_G$ or an encapsulated fault tree if $P \in P_{SC}$
- $E$ is a set of directed edges which are associated to a source and a target node, where:
  - A source node can be an intern failure event $n \in N_{intern}$, an input failure port $n \in N_{in}$, or an output failure port of a contained proxy $n \in P.N_{out}^{extern}$
  - A target node can be an output failure port $n \in N_{out}$ or an input failure port of a contained proxy $n \in P.N_{in}^{extern}$
  - Further only one edge $e \in E$ can have an output failure port $n \in N_{out}$ or an input failure port of a contained proxy $n \in P.N_{in}^{extern}$ as target

### 3.2 Annotation of Components with Encapsulated Fault Trees and Model Based Analysis

To annotate a component with an encapsulated fault tree we utilize the corresponding interface specification. Therefore, we assume that each input and each output action could be faulty. Thus, an encapsulated fault tree contains exactly as many outgoing failure ports as the number of output actions. The number of incoming failure ports depends on the number of input actions and the number of failures of the hardware platform or operating system, because these failures can also lead to a failure of the software component. Because an internal action or an internal fault in the software component may cause a failure, we specify them as internal failure

events in the encapsulated fault trees. Based on this methodology to each flat component an encapsulated fault tree can be assigned modeling the causes of an outgoing failure. These encapsulated fault trees of the flat components serve as the basis for the construction of encapsulated fault trees for hierarchical components. Thus, we extend the meta-class of a component with an attribute `FaultTree` and an operation `GenerateFaultTree` that construct an encapsulated fault tree for a hierarchical component:

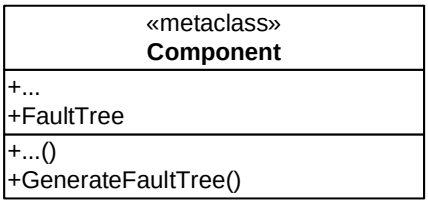


**Fig. 3.** Extended meta-class for a component specification with encapsulated fault tress

The operation `GenerateFaultTree` must be defined as follows:

```
void GenerateFaultTree(){
   if (this.ContainsSubComponents?()){
      Component active, insert;
      set<Component> open, close, neighbors;
      set<Action> sharedaction;
      Action activeaction;
      this.FaultTree.Clear();
      open.Add(GetFirst(this.SubComponents));
      do{
         active=open.GetFirst();
         active.GenerateFaultTree():
         this.FaultTree.AddFaultTreeComponent(active.FaultFree);
         neighbors=active.ExpandNeighbors ();
         do{
            insert=neighbors.GetFirst(); neighbors.Remove(insert);
            if (close.Contains?(insert)){
               sharedaction=GetSharedAction(insert,active);
               do{
                 activeaction = sharedaction.GetFirst();
                 sharedaction.Remove(activaction);
                 qport= this.FaultTree.GetOutPort (activeaction,insert);
                 dport= this.FaultTree.GetInPort (activeaction,active);
                 this.FaultTree.Connect(qport,dport);
               }
               while(!sharedaction.isEmpty())
            }
            else if (!open.Contains?(insert)){
               open.add(insert);
            }
         }
         while(!neighbors.isEmpty())
         close.Add(activ);
         open.Remove(activ);
      }
      while(!open.isEmpty())
   }
}
```

This algorithm is basically a recursive graph search algorithm that systematically explores the contained components of a hierarchical component. Therefore, it uses the two component sets `open` and `closed`. The set `closed` contains all components that have been already explored. The second set `open` contains the components that are a neighbor of one component in the set `closed` and that must be explored in the future.

To construct the fault tree a component from the set `open` is selected. It is denoted as the `active` component. For this component the operation `GenerateFaultTree` is called. The constructed fault tree of the selected component is added to a current fault tree. Then the neighbors of the selected component are explored. If they are not in the set `closed` they are added to this set or otherwise the input or output failure ports of encapsulated fault trees must be connected with the input or output failure ports of the `active` components fault tree. Therefore, the set of shared action between the two components is determined and based on this set the corresponding input and output failure ports are connected. Finally, the `active` component is added to the `closed` set and thus all encapsulated fault trees of the components in the `closed` set are connected.

## 4 Example

To present the feasibility of our approach we chose to model a simplified version of a control system for a level crossing. This system is constructed with a COTS-component that controls train signal actuators and gate actuators. To get the information from the environment the control component utilizes a sensor that determines the state of the gates and a sensor that detects an arriving train and its progress through the level crossing. The following structure diagram illustrates the structure specification of the level crossing control system.

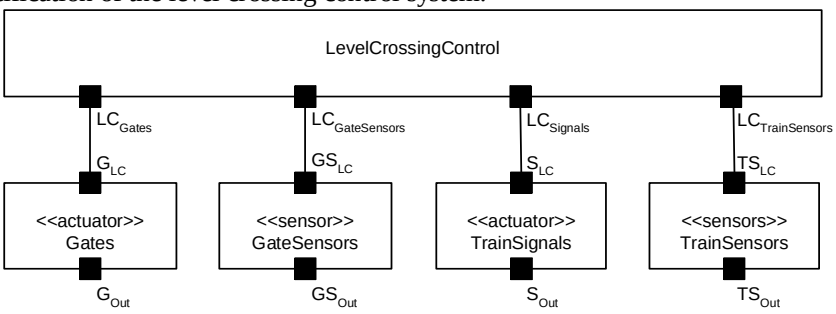


**Fig. 4.** Structure specification of the level crossing example

For each component of the structure specification the behaviour is characterized with an interface automaton in Figure 5 and 6. In addition to this, the corresponding ports in the structure specification are annotated for each input and output message.

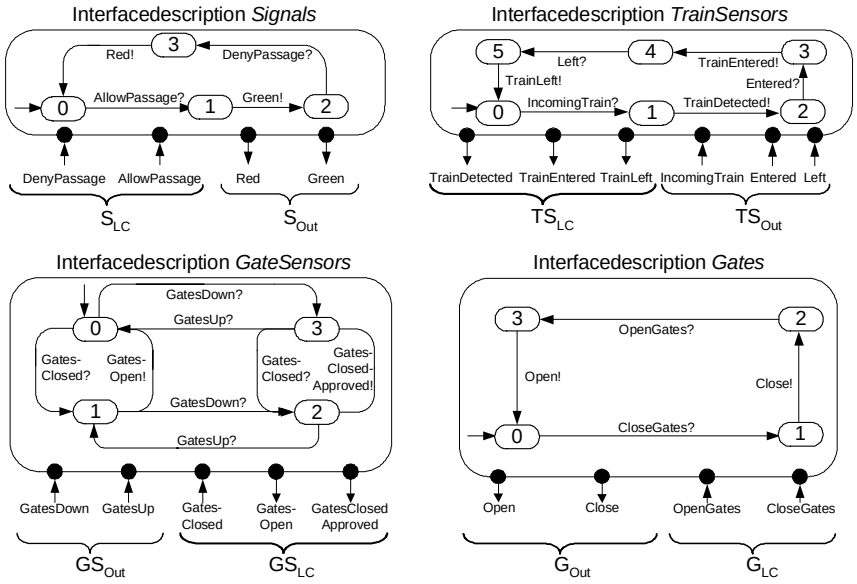


**Fig. 5.** Interface specifications of the sensors and actuators in the level crossing example

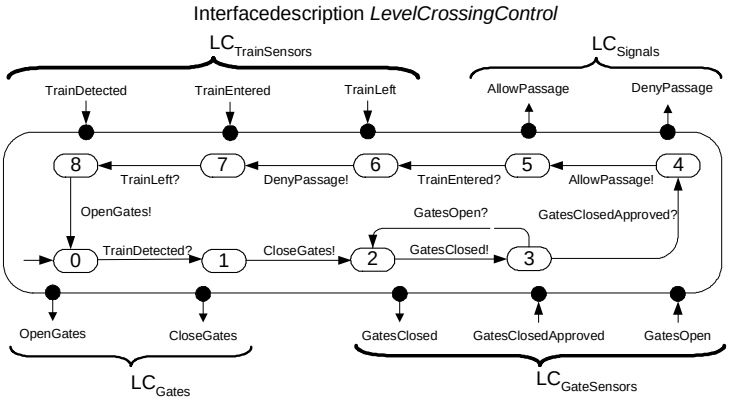


**Fig. 6.** Interface specification of the *LevelCrossingControl* component

Based on the interface automata and our domain knowledge it is clear that the hazard condition of the system is to signal green to the arriving train when the gates are open. This can be the case if either the system gives a “green” signal in the wrong situation or the system omits to set the signal “red” after the train has entered the level crossing section. To evaluate the probabilities of these events for every component of our level crossing example an encapsulated fault tree is assigned. These fault trees are depicted for the sensors and actuators in figure 7. Notice that the encapsulated fault tree contains an input or output failure port for each input or output action

in the corresponding interface specification. As an example, the *Gates* component contains an output failure port for the *Open* and *Close* action, which can be caused either by a failure of the hardware or by a failure of the *OpenGates* and *CloseGates* action. All other sensors and actuators are straightforward. They send faulty signals in case the corresponding external signal is faulty or a hardware failure occurs.

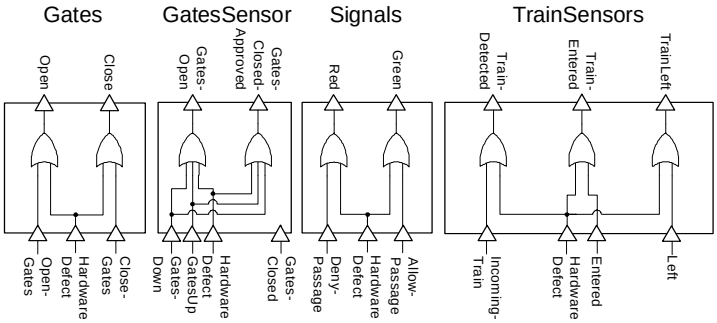


**Fig. 7.** Encapsulated fault trees for the sensors and actuators in the level crossing example

The control component implies a much more complex fault tree that also contains internal failure events. The reason for these internal failure events can be a programming error.

In the example we identified three internal failure events that can lead to a system hazard. The first is the sending of the *AllowPassage* message in a wrong situation. This can be caused by a fault in the control flow. The second internal failure event is that the system reacts too late to a *TrainEntered* signal due to a missing of a performance requirement. In this case, the *DenyPassage* signal would be sent too late and a second train could enter the level crossing control area. In case of the third internal failure event an omission of the *DenyPassage* signal may occur, too. For this event we assume that the *TrainEntered* signal is ignored.

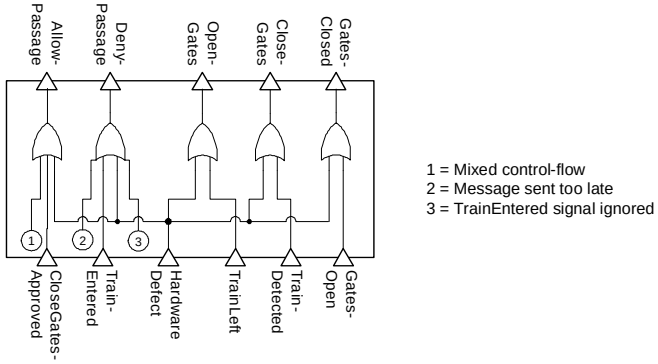


**Fig. 8.** Encapsulated fault tree of the *LevelCrossingControl* component

Based on the encapsulated fault trees of the components the fault tree of the level crossing control system can be constructed with the `GenerateFaultTree` algorithm.

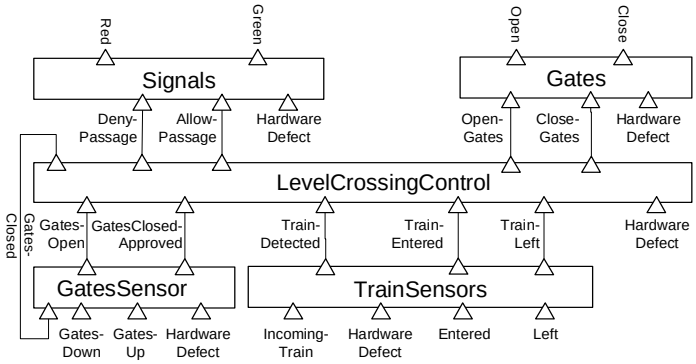


**Fig. 9.** Composed fault tree for the level crossing example

Based on these complete fault trees and the probability of the input failure ports the probability of the hazard can be calculated as for a normal fault tree.

## 5 Conclusion and Future Work

In this paper we proposed a technique for the annotation of COTS-Components with encapsulated fault trees. This technique utilizes interface specifications with interface automata for the construction of fault trees for flat components. To allow the automatic and systematic construction of fault trees for a complete system an algorithm is introduced. This algorithm constructs a fault tree for an encapsulated hierarchic component based on the structure specification and the encapsulated fault trees of the utilized components. The constructed fault trees allow evaluating the probabilities of system hazards or components output failures. In addition to this, software architects are enabled to systematically select appropriate components that fulfill the safety requirements.

In view of our contribution and the overall problem we conclude with some items that remain for future work. First of all, the prediction of the probability of an internal fault must be improved. Up to now, predictions depend mostly on expert knowledge.

Second, there is a need for special failure types which model different aspects of failures that can be propagated between two components. In [6,16] the following failure modes (failure types) are suggested:

- tl timing failures (reaction too late)
- te timing failures (reaction too early)
- v value failures
- c failures of commission
- o failures of omission

The usage of these failure types would extend the expressiveness of an encapsulated fault tree. Nevertheless, it would also increase the complexity of encapsulated fault trees, which must be handled in an appropriate way.

## References


1. Bass L., Clements P., Kazman R. Software Architecture in Practice, Addison-Wesley 1998.
2. Birolini A.: Reliability engineering: theory and practice, New York, Springer, 1999.
3. Chakrabarti A., de Alfaro L., Henzinger T. A., Jurdzinski M., and Mang F. Y.C., Interface compatibility checking for software modules. Proceedings of the 14th International Conference on Computer-Aided Verification (CAV), Lecture Notes in Computer Science 2404, Springer-Verlag, 2002, pp. 428-441.
4. Clements P., Kazman R., Klein M., Evaluating Software Architectures: Methods and Case Studies, Addison-Wesley, 2001.
5. de Alfaro L., Henzinger T.A., Interface automata, in: 9th Symp. Foundations of Software Engineering, ACM Press 2001
6. Fenelon P., McDermid J.A., Nicholson M., Pumfrey D. J., Towards Integrated Safety Analysis and Design, ACM Applied Computing Review, 1994.
7. Grunske L., Neumann R., Quality Improvement by Integrating Non-Functional Properties in a Software Architecture Specification, in Proceedings of the Second Workshop on Evaluating and Architecting System dependability, San Jose, October 3-6, 2002, pp 23-33
8. Harel D., Statecharts: A visual formalism for complex systems. Science of Computer Programming, 8, 1987,pp 231-274
9. Hofmeister C., Nord R., Soni D.: Applied Software Architecture, Reading, MA: Addison Wesley Longman, 1999
10. IEC 61025, Fault Tree Analysis (FTA), International Electrotechnical Commission.
11. Kaiser, B., Liggesmeyer, P., Mäckel, O., A New Component Concept for Fault Trees. in Proceedings of the 8th Australian Workshop on Safety Critical Systems and Software (SCS'03), Adelaide, to appear 2003
12. Liggesmeyer P., Qualitätssicherung softwareintensiver technischer Systeme, Spektrum-Akademischer-Verlag, Heidelberg, 2000
13. Luckham D. C., Kenney J. J., Augustin L. M., Vera J., Bryan D., Mann W., Specification and Analysis of System Architecture Using Rapide, in: IEEE Transactions on Software Engineering, Vol. 21, No. 4, April 1995, pp. 336-355
14. Papadopoulos Y., McDermid J. A., Sasse R., Heiner G., Analysis and Synthesis of the Behaviour of Complex Programmable Electronic Systems in Conditions of Failure, Reliability Engineering and System Safety, 71(3), Elsevier Science, 2001, pp 229-247.
15. Papadopoulos Y., McDermid J. A., (1999) Hierarchically Performed Hazard Origin and Propagation Studies, SAFECOMP '99, 18th Int. Conf. on Computer Safety, Reliability and Security, Toulouse, 1999, LNCS, 1698:139-152
16. Pumfrey D. J., The Principled Design of Computer System Safety Analyses, Dissertation, University of York, 1999.
17. Selic B., Gullekson G., Ward P. T., Real-Time Object-Oriented Modeling. Wiley, New York, 1994.
18. Szyperski C.: Component Software. Beyond Object-Oriented Programming. ACM Press/ Addison Wesley, 1998
19. Tapken J., Implementing hierarchical graph structures. In J. P. Finance, editor, Proc. Formal Aspects of Software Engineering (FASE'99), Lecture Notes in Computer Science, 1577, Springer, 1999.

# A Generic Ontology for the Specification of Domain Models

*Rosario Girardi and Carla Gomes de Faria*

Department of Computer Science, Federal University of Maranhão,
Av. Dos Portugueses s/n - Campus do Bacanga, São Luis - MA , Brazil
*rgirardi@deinf.ufma.br; carlaslz@yahoo.com.br*

**Abstract**. Ontologies are becoming an important field in Software Engineering especially useful for the specification of high-level reusable software, like domain models and frameworks. This work proposes a generic ontology to guide the construction and specification of domain models in the agent-based development paradigm. ONTODM represents the knowledge of techniques for the specification of the requirements of a family of multi-agent systems in an application domain. It is being used as a CASE tool to aid the elicitation and specification of domain models.

## 1. Introduction

In traditional Software Engineering, for each new application to be built, a new conceptualization is developed. For each new application, an elicitation phase is accomplished almost always from scratch, focusing on all particularities of the system in hand. This approach is extremely expensive since elicitation is an activity of great effort in software development. Therefore, it is important to share and reuse the captured knowledge.

Ontologies are becoming an important field in Software Engineering especially useful for the specification of high-level reusable software, like domain models and frameworks.

This work introduces the definition and design of ONTODM, an ontology for the construction of domain models in the agent-based development paradigm.

The article is organized as follows. Section 2 introduces main concepts about Domain Engineering, and ontologies. Section 3 describes the current approach of the MaAE research project to Domain Engineering. Section 4 describes the knowledge to be represented and shows the definition and design of ONTODM.

## 2. Ontology based domain analysis

### 2.1 Domain Engineering

Domain Engineering is a systematic approach for the construction of reusable software products [1] [25] [27] [35]. The process of Domain Engineering is composed of the phases of analysis, design and implementation of a software application domain.

Domain analysis activities identify reuse opportunities and determine the common requirements of a family of systems in a domain. The product of this phase is a domain model. Domain design activities look for a documented solution to the problem specified in a domain model. The product of this phase is a framework - a reusable software architecture - and a collection of design patterns, documenting good solutions in that domain. Reusable components integrating the framework are constructed during the phase of domain implementation. This is the compositional approach of Domain Engineering.

In a generative approach, Domain Engineering produces Domain Specific Languages (DSLs), which can be used as application generators to construct a family of applications in such a domain. Knowledge of the domain and design patterns are encoded in DSLs.

### 2.2 Ontologies in software development

For the purpose of this paper, an ontology is a representation vocabulary, often specialized to some domain or subject matter. More precisely, it is not the vocabulary as such that qualifies

as an ontology, but the conceptualizations that the terms in the vocabulary are intended to capture [1].

Domain ontologies are formal descriptions of the classes of concepts and the relationships between those concepts that describe an application domain. Frame-based knowledge representation structures are usually used to represent domain ontologies where concepts are represented in frame-based classes and attributes and relationships between concepts as slots of the frames. The classes in the ontology along with a set of class instances constitute a knowledge base.

Ontologies are especially useful for the development of high-level reusable software, like domain models and frameworks. They provide a not ambiguous terminology that can be shared by all involved in the development process. Also, an ontology can be as generic as needed allowing its reuse and easy extension.

In an ontology-based domain analysis approach, the elicitation and modeling of software requirements can be accomplished in two stages. First, the general domain knowledge should be elicited and specified as ontologies. These ontologies, in turn, are used to guide the development of specific applications in that domain, when the particularities of a specific application are considered. In this way, the same ontology can be used to guide the development of several applications, diluting the costs of the first stage and allowing knowledge sharing and reuse.

An ontology cannot being considered just a building block going to be adapted and reused but rather a tool - analogous to any other CASE (Computer-Aided Software Engineering) tool - that can increase the quality and productivity of the analysis process.

An important reason for using ontologies in the agent-based development process is that they are essential for enabling the communication between software agents. Software agents communicate with each other via messages that contain expressions formulated in terms of an ontology. In order for a software agent to understand the meaning of these expressions, the agent needs access to the ontology they commit to.

## 3. Multi-Agent Domain Engineering

In MaAE [17] [18], we are working on a model for Multi-Agent Domain Engineering. Domain Engineering, also known as Development FOR Reuse, is the process of creating reusable software abstractions, and Application Engineering or Development WITH Reuse, the one of constructing an specific application using reusable software abstractions.

We expect to construct a software development environment composed by a set of development tools and libraries of high-level reusable software abstractions for agent-based application development. Experiments are being conducted on the legal, tourism, and information access application domains.

### 3.1 Developing multi-agent specific applications

In MaAE, a multi-agent specific application is obtained through the composition of a set of reusable agent frameworks available in the library of the development environment (Figure 1). These frameworks are realizations of high-level software abstractions in the library. Particular requirements of a multi-agent specific application are used to map the specification level of domain and user models into a realization satisfying such needs. The realization should have associated a set of frameworks, which are agent-based solutions to those requirements.

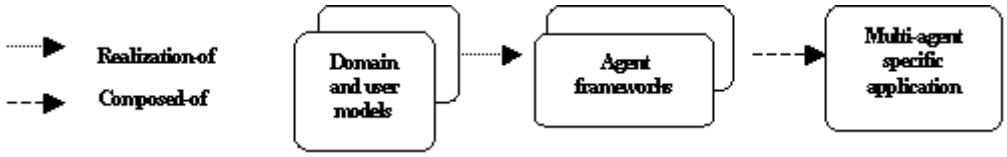


**Fig. 1.** A model for Agent-based Application Engineering

### 3.2 Developing high-level software abstractions

Figure 2 illustrates a model for Domain Engineering in MaAE, showing their activities - Domain Analysis, Domain Design and Domain Implementation - and generated products. Ontologies [1] [20] [21] are being used to represent both the knowledge of techniques for Domain Engineering and generated products [13] [15][26].

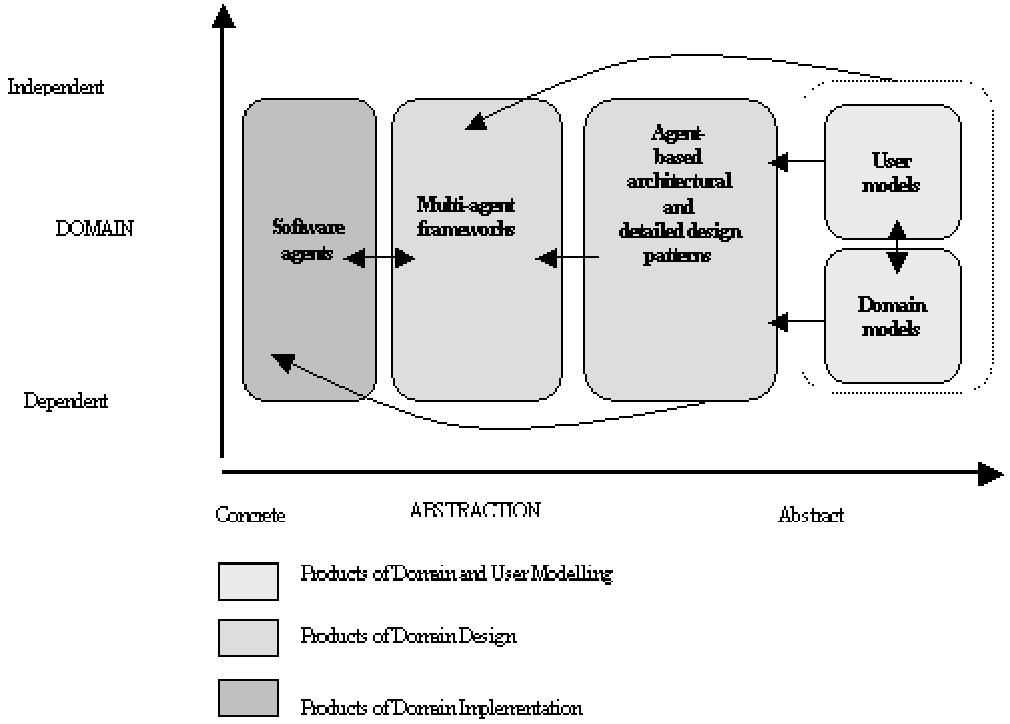


**Fig. 2.** Software abstractions and processes of Multi-Agent Domain Engineering

The reusable agent-based software abstractions are illustrated considering their abstraction level (from abstract to concrete) and their dependence level from the application domain (from domain dependent to domain independent): Domain models, User models, Agent-based architectural and detailed design patterns, Multi-agent frameworks, and Software agents.

Domain and user modelling produces the requirement specification of a family of similar systems in an application domain. The specification includes both the required functionality of the domain and features of end users as well. Domain and user models are the reusable software products generated by the Domain and User modelling phase of Agent-based Domain Engineering.

The domain model - application domain dependent and specified at a high level of abstraction - represents the formulation of a problem, knowledge or activity of the real world. The formulation is generic enough to represent a family of similar systems. Ontologies are being used to represent domain models establishing the vocabulary and semantics for the elements, processes and relationships in the systems. A generic ontology, ONTODM, guides

the construction of domain models, which are created by instantiating the hierarchy of meta-classes of ONTODM. Thus, a domain model is represented in a frame-based ontology where concepts, activities and relationships in the domain are represented in frame-based classes according to the representation criteria of ONTODM (Figure 3).

During Application Engineering, the requirement specification of a specific application is obtained through the composition of a set of ontology-based domain models, specialized according to the particular requirements of the specific application (Figure 3).

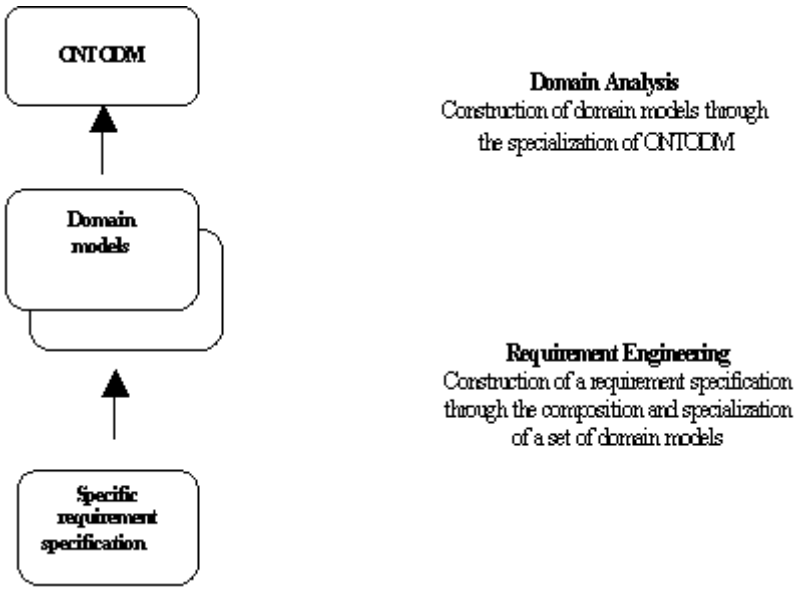


**Fig. 3.** Ontology-based domain analysis in MaAE

The user model specifies features, needs, preferences and goals of end users [14]. Ontologies are also being used to represent user models.

Domain design produces a reusable design specification for a family of similar systems in an application domain. The design specification is composed by a set of agent-based architectural and detailed design patterns and frameworks [13][16].

Domain implementation transforms the solutions identified during domain design in a set of reusable components.

## 4. ONTODM: A generic ontology for the construction of domain models

ONTODM is a generic ontology representing the knowledge of methods for constructing domain models. The ontology specifies the commonalities and differences of the software requirements of a family of applications to be developed according to the agent paradigm. It is a knowledge-based tool, which guides the elicitation and specification of the concepts and tasks to be accomplished in the domain.

The development process of ONTODM (Figure 4) consists of two phases: the definition and the design of the ontology. The techniques used in each phase are inspired in several proposals for ontology development [20] [23] [30].

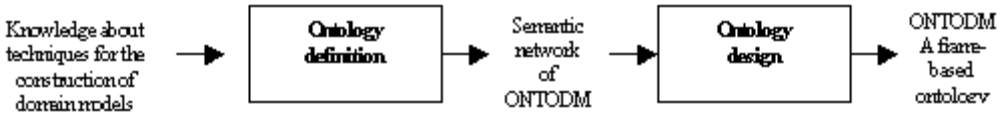


**Fig. 4.** The development process of ONTODM

In the definition phase, the knowledge of methods for constructing domain models is represented in a semantic network. In the design phase, concepts and relationships in the semantic network are mapped to a frame-based ontology.

### 4.1 Knowledge about techniques for the construction of domain models

Main techniques of Domain Analysis and Requirement Engineering of multi-agent systems were considered for the extraction of the knowledge to be used in the construction of the domain models.

Domain Analysis models the concepts and functionalities required by a family of systems by consulting domain specialists, and sources of information in the domain. Commonalities and differences of software applications in the domain are also analyzed. Several methods for Domain Analysis have been proposed as the Prieto Diaz's approach [35], Object Oriented Domain Modeling - ODM [38] and features-based approaches [7] [24]. The current version of ONTODM just represents the knowledge of Domain Analysis techniques for modelling main concepts of the domain and the relationships between these concepts.

Methods for Requirement Engineering of multi-agent systems [22] [42], like GAIA [41], MaSE [10] [11] [40], MADS [19], TROPOS [5] [29], SODA [32], PASSI [8], AUML [31], and MESSAGE [4], usually focus on the modelling of goals, roles, activities and interactions of individuals of an organization.

Unfortunately, there is not still a common definition of those modeling concepts. Therefore, in the MaAE project [18], we are considering the following concepts. An organization is composed by individuals with general and specific goals that establish what the organization intends to reach. The achievement of specific goals allows reaching the general goal of the organization. For instance, an information system can have the general goal of "satisfying the information needs of an organization" and the specific goals of "satisfying dynamic or long term information needs". Specific goals are reached through the exercise of responsibilities that individuals have. Individuals play roles with a certain degree of autonomy and exercise their responsibilities through the execution of activities. For that, they dispose of a set of resources. For instance, an individual can play the role of "information retriever" with the responsibility of executing activities allowing satisfying the dynamic information needs of an organization. Another individual can play the role of "information filter" with the responsibility of execute activities allowing to satisfy the long-term information needs of the organization. Resources can be, for instance, the rules of the organization to access and structure its information sources. Sometimes, individuals have to communicate with other internal or external individuals to cooperate in the execution of an activity. For instance, the individuals playing the roles of information retriever and information filter may need to interact with an individual (e.g. Information source) having the responsibility of the storage and update of the information items of the organization.

According to these definitions, requirement modelling of multi-agent systems is based on the following modelling tasks:

- *Goal modelling*. Considering the problem that the system intends to solve, the general goal of the system can be identified. Specific goals are obtained through a refinement of the general goal.
- *Role modelling*. For each specific goal, the responsibilities that will be exercised by internal or external roles are identified. Then, the activities that will allow exercising each responsibility are defined. During this refinement process, it can be identified that the same activity or a set of related activities are executed by several roles. In this case, it should be appropriate the creation of an independent role having the responsibility of executing these activities on behalf of the other roles. The resources that an individual playing a role will need to execute his/her activities are also identified.

- *Interaction modelling*. Through an analysis of their respective activities, the interactions between internal and external roles are identified.

### 4.2 Ontology definition

In this phase, the knowledge for the construction of domain models is represented in a semantic network (Figure 5 and Figure 6). Figure 5 shows the part of the semantic network representing the modelling tasks and generated products. Domain modelling builds a domain model composed of a goal, role, interaction and domain concept models constructed through the subtasks Goal, Role, Interaction and Concept modelling, respectively. Goal modelling builds a goal model, representing the general and specific goals of the system. Role modelling builds a role model, representing the role. Interaction modelling builds a model of interactions, representing the interactions between the roles and the external entities. Concept modelling builds a model of domain concepts, representing the concepts of the domain and the relationships between them.

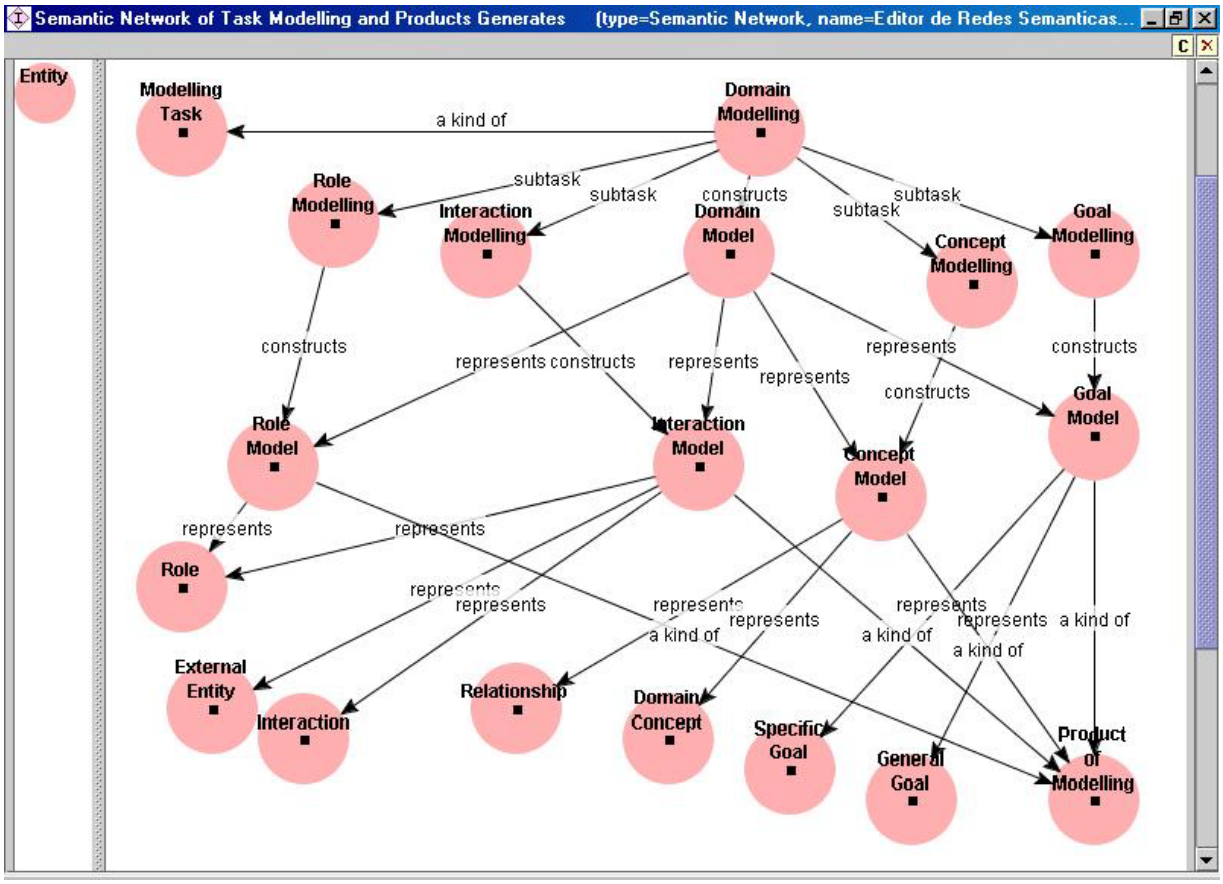


Fig. 5. Semantic network of modelling tasks and products of modeling

Figure 6 shows the part of the semantic network representing the concepts of modeling, their relationships and attributes: goal, role, interaction, domain concept, activity, responsibility, resource, relationship between domain concepts and external entities.

Goals are classified in general and specific ones. Specific goals lead to the general goal. Specific goals are reached through the exercise of the responsibility of a role. A responsibility is exercised through the accomplishment of activities. Roles use resources for executing their activities. Activities can manipulate domain concepts. Interactions have sources and destinations, which can be roles or external entities. Domain concepts can be related according to a particular relationship.

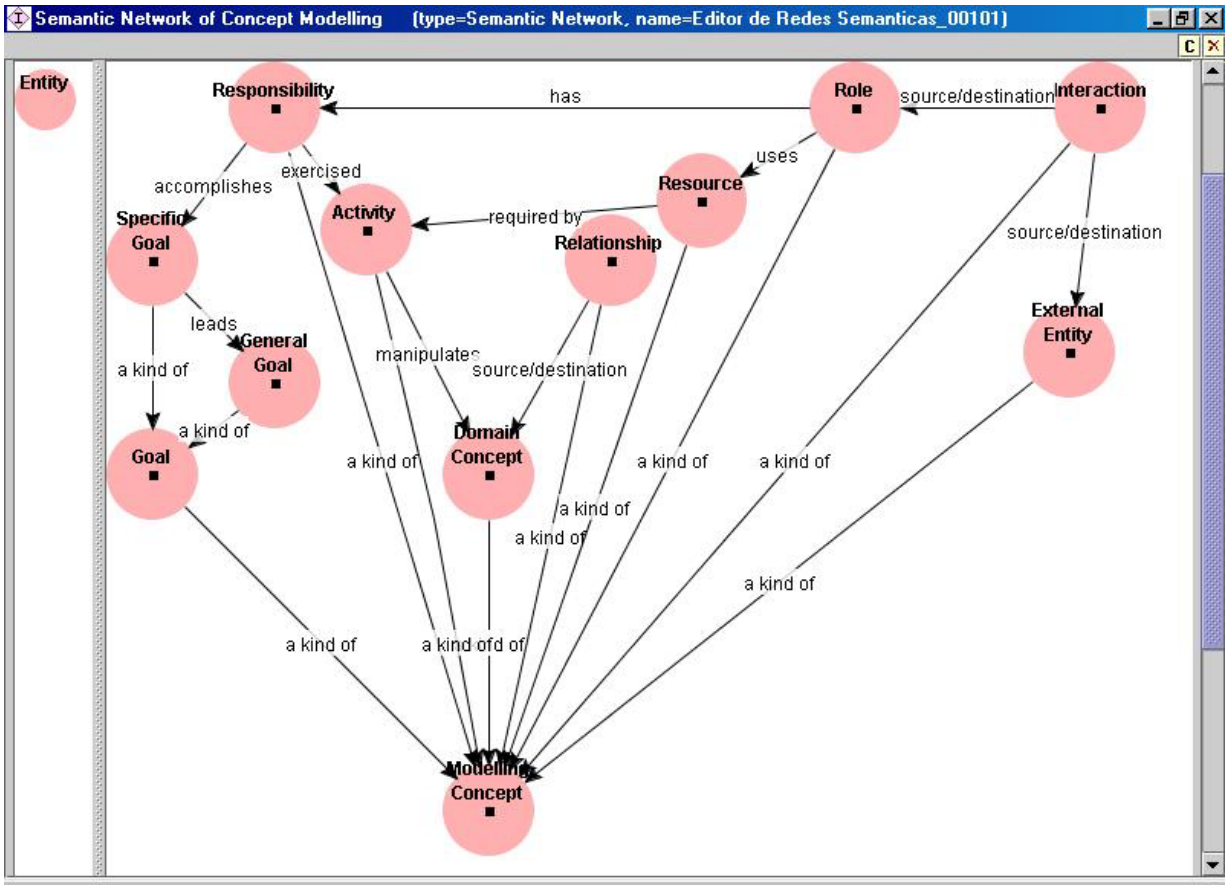


Fig. 6. Semantic network of modeling concepts

### 4.3 Ontology design

In the design phase, concepts and relationships in the semantic network are mapped to ONTODM, a frame-based ontology composed of a hierarchy of meta-classes, according to the following modelling rules. Nodes are mapped to meta-classes. Nodes related by a link of type *a kind of* are mapped in a hierarchy of subclasses and superclasses. Other links are mapped to slots of the corresponding meta-class. Appropriate facets are associated to each slot, like type and cardinality.

Figure 7 shows the hierarchy of meta-classes of ONTODM and the Role Model meta-class of ONTODM in the Protégé editor [37]. The slot represents roles of the meta-class Role Model is of type instance of the meta-class Role. The slot constructs of the meta-class Role Modelling is of type instance of the meta-class Role Model.

Figure 8 shows the Role meta-class of ONTODM, which has the slots *has responsibility* and *uses resources* of type instance of the meta-classes *Responsibility* and *Resources*, respectively.

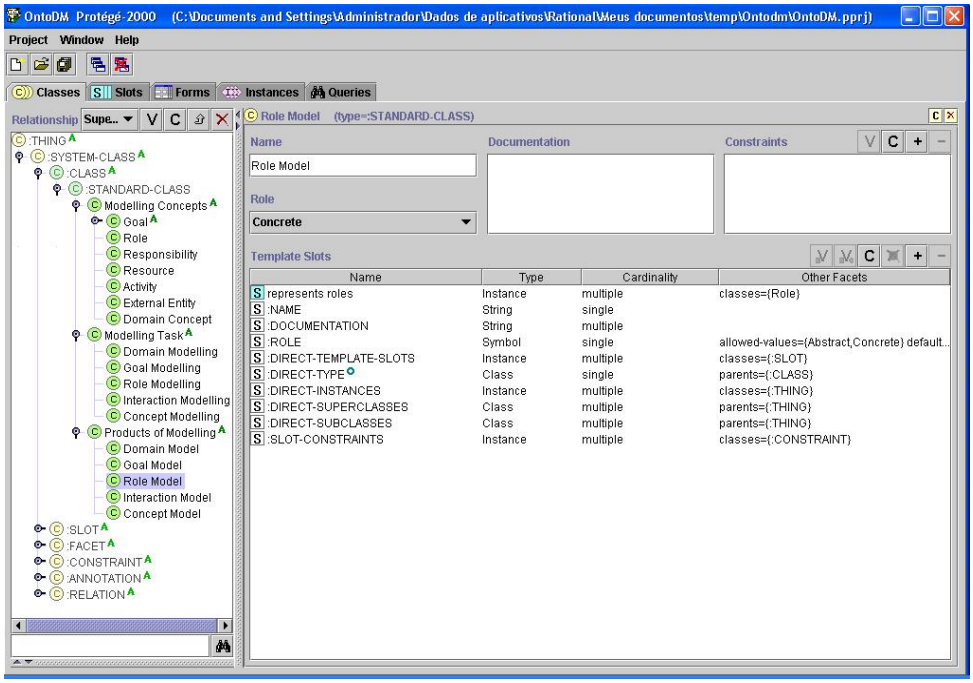


Fig. 7 Meta-class hierarchy of ONTODM and the *Role Model* meta-class

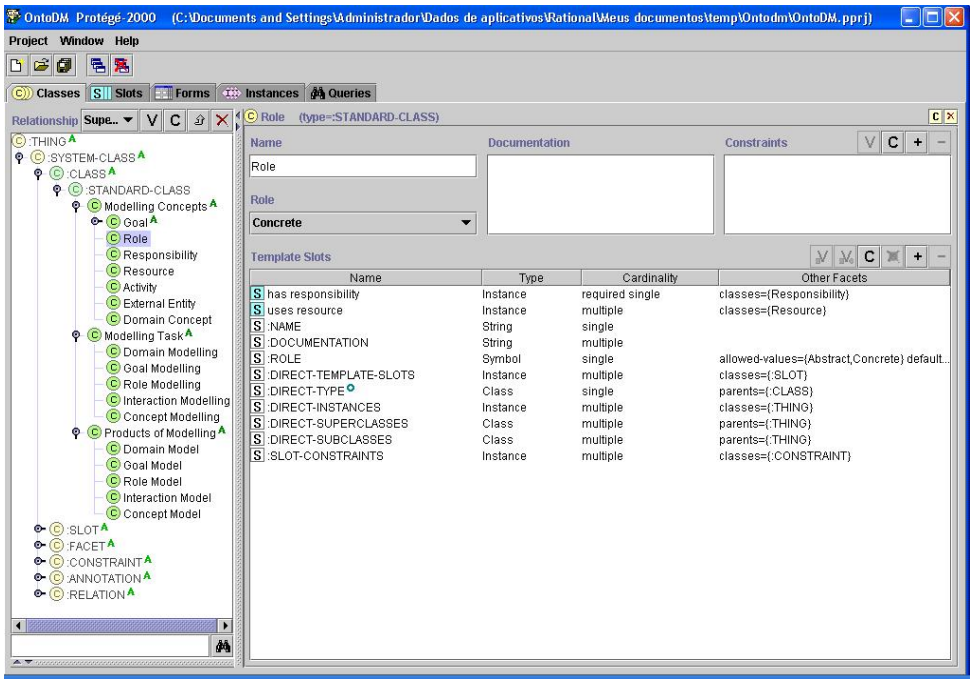


**Fig. 8** *Role* meta-class of ONTODM

## 5. Related Work

Considering their advantages, the use of ontologies for the representation of software products is increasing [9][28]. Particularly, some methodologies for the development of multi-agent specific applications are being extended for ontology support [8][11].

Some approaches for Ontology-based Domain Analysis have been proposed [9], which integrate methodologies for ontology building with techniques for Domain Analysis. Our approach differs from these proposals mainly in two aspects. First, we are proposing a meta-domain model - ONTODM – which represents the knowledge of both techniques for

requirement analysis of multi-agent systems and techniques for Domain Analysis. Second, ONTODM is a CASE tool, which guides the application of GRAMO [12], the technique we have developed for the construction of ontology-based domain models.

## 6. Concluding remarks and further work

This article proposes ONTODM, a generic ontology for the construction of domain models to be reused in the development of multi-agent applications. ONTODM represents the knowledge of techniques for the specification of the requirements of a family of multi-agent systems in an application domain. It is being used as a CASE tool to aid the elicitation and specification of domain models.

The ontology is being validated and improved through the specification of domain models in the legal, tourism and information access domains [6][26][38]. The ontology is also being extended to support the construction of user models.

ONTODM and the domain models we are constructing with it are being used in the definition of a technique based on patterns and ontologies for the construction of multi-agent frameworks [13][16][32][32]. The final goal is the specification of a methodology for Agent-based Domain Engineering, and a framework for ontology-based specification of agent-based reusable artefacts, exploring both compositional, like the one introduced in this paper, and generative approaches. In our generative approach, domain models are being used as the main resources for the construction of Domain Specific Languages.

Some of the advantages of using ontologies for the representation of reusable products have been shown in this article. Although ONTODM has been designed for its integration in a software development environment for Multi-agent Domain Engineering, the approach can be generalized to other development paradigms. For that, the ontology should be re-designed according to the particular knowledge of those development paradigms techniques.

## 7. Acknowledgments

This work is supported by CNPq, an institution of Brazilian Government for scientific and technologic development.

## 8. References

[1] Arango, G. "Domain Analysis Methods," 17-49. Software Reusability. Chichester, England: Ellis Horwood, (1994).

[2] Braga, R.; Werner, C.; Mattoso, M. “Odyssey: A Reuse Environment based on Domain Models”, IEEE Symposium on Application-Specific Systems and Software Engineering Technology (ASSET'99), 50-57, Texas, Mar 1999.

[3] Chandrasekaran, B. and Josehson, J. "What are Ontologies, and Why Do We Need Them?", IEEE Intelligent Systems, Vol 14, nº1, (1999)20 – 26.

[4] Caire, G., et alii. "Agent-Oriented Analysis using MESSAGE/UML", In: M. Woodridge, P. Ciancarini, and G. Weirs, editors. Second International Workshop on Agent-Oriented Software Engineering, AOSE 2001, (2001) 101-108.

[5] Castro, Jaelson, Kolp, Manuel, Mylopoulus, John. "A Requirement-Driven Software Development Methodology", 13th International Conference on Advanced Information Systems Engineering CAISE01, Interlaken, Switzerland, June (2001) 4-8.

[6] Cerveira, Núbia. Development of an Ontology-based Domain Model for the Touristical Area, CEAPS-DEINF-UFMA, Final degree work, (2003).

[7] Cohen, S., Kang, K., Hess, J., Nowak, W., Peterson, S. "Feature Oriented Domain Analysis (FODA) Feasibility Study", Technical Report CMU/SEI-90-TR-21. Software Engineering Institute, Cornegie Mellon University, Pittsburgh, PA, November (1990).

[8] Cossentino, M., Burrafato, P., Lombardo, S., Sabatucci, S., "Introducing Pattern Reuse in the Design of Multi-Agent Systems", AITA02, Workshop at NODE 02, Erfurt, Germany, October (2002), pp. 8-9.

[9] De Almeida Falbo, Ricardo; Guizzardi, Giancarlo and Duarte, Katia Cristina, “An Ontological Approach to Domain Engineering”, In: Proceedings of the International Conference on Software Engineering and Knowledge Engineering (SEKE´92), July 15-19,2002, Ischia, Italy.

[10] DeLoach, S.: "Multi-Agent Systems Engineering: A Methodology and Language for Designing Agent Systems" In: Proceeding of the 1st Bi-Conf. Workshop on Agent-Oriented Information Systems (AOIS'99), Heidelberg, Germany, (1999).

[11] Dileo, J., Jacobs, T., Deloach, S. "Integrating Ontologies into Multi-Agent Systems Engineering", Proceedings of 4th International Bi-Conference Workshop on Agent Oriented Information Systems (AOIS 2002), Bologna (Italy), July (2002), pp. 15-16.

[12] Faria, Carla and Girardi, Rosario. “An Ontology-based Technique for Domain Modeling in Multi-Agent Domain Engineering”, In: Proceedings of XXIX Latin American Conference of Informatics (CLEI 2003). La Paz, Bolívia, (2003). (to appear)
[13] Ferreira, S. and Girardi, R. "Specification of a Generic Ontology for Domain Design". Submitted paper, (2003).
[14] Fleming, M. and Cohen, R., "User Modeling in the Design of Interactive Interface Agents", Proceedings of UM99 (User Modeling), (1999).
[15] Girardi, R, Lindoso, A. "Specification of a Legal Domain Model Using Ontologies". Submitted paper, (2003).
[16] Girardi, R, Ribeiro, I., and Bezerra, G. "Towards a System of Patterns for the Design of Agent-based Systems". Proceedings of The Second Nordic Conference on Pattern Languages of Programs” (VikingPLoP 2003). Bergen, Norway (2003). (to appear)
[17] Girardi, R. "Agent-Based Application Engineering". In: 3RD International Conference on Enterprise Information Systems (ICEIS 2001), Setúbal, (2001).
[18] Girardi, R., "Reuse in Agent-based Application Development", 1st International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS´2002), ICSE´2002, May (2002).
[19] Girardi R. and Sodré A. "A Methodology for Multi-Agent Application Development", In: Proceedings of the ITS´2002 Workshops - Architectures and Methodologies for Building Agent-Based Learning Environments, 6th International Conference on Intelligent Tutoring Systems, www.fapeal.br/its2002workshop_agents, (2002).
[20] Gruber, T. R. "Towards Principles for the Design of Ontologies used for knowledge Sharing". International Journal Human-Computer Studies. Nº 43, (1995) 907-928.
[21] Guarino, Nicola. "Formal Ontology in Information Systems" Proceedings of FOIS'98, Trento, Italy, June (1998) 6-8. Amsterdan, IOS Press, 3-15.
[22] Iglesias, C. et al., "A Survey of Agent-Oriented Methodologies", In: Intelligent Agents V. Agents Theories, Architectures, and Languages, Lecture Notes in Computer Science, Springer-Verlag, (1998).
[23] Jones, Dean, Bench - Capon, Trevor and Visser, Pepjin. "Methodologies for Ontology Development." Proc. - It&Knows Conference of the 15th IFIP World Computer Congress, Budapest, Chapman-Hall, (1998).
[24] Krut, R. "Integrating 001 Tool Support into the Feature Oriented Domain Analysis Methodology", Technical Report CMU/SEI-93-TR-11. Software Engineering Institute, Carnegie Mellon University, (1993).
[25] Krut, R. & Zalman, N. Domain Analysis Workshop Report for the Automated Prompt & Response System Domain (CMU/SEI-96-SR-001). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, (1996).
[26] Lindoso, A. "Specification of a Domain Ontology for the Legal Domain", CGCC-UFMA, Final degree work, (2003). (In Portuguese)
[27] Moore, J. M., Bailin, S. C. "Domain Analysis: Framework for reuse". IEEE Computer Society Press. Tutorial, (1991) 179-202.
[28] Musen, Mark A. “Domain Ontologies in Software Engineering: Use of Protégé with the EON Architecture”, Methods of Information in Medicine. Edição 37. Número: 4-5. (1998) 540-550.
[29] Mylopoulos, John, Castro, Jaelson. "TROPOS: A Framework for Requirements-Driven Software Development", In J. Brinkkemper and A. Solvberg (edts), Information Systems Engineering: State of the Art and Research Themes, Lecture Notes in Computer Science, Springer, Verlag, June (2000) 261- 273.
[30] N. F. Noy & D. L. McGuinness. "Ontology Development 101: A Guide to Creating Your First Ontology". Knowledge Systems Laboratory, March (2001).
[31] Odell, J., Parunak, H.V.D., and Bauer, B. Extending UML for Agents. Proc. Of the Agent-Oriented Information Systems Workshop at the 17th National Conference on Artificial Intelligence, accepted role, AOIS Workshop at AAAI (2000) 3-17.
[32] Oliveira, Ismênia and Girardi, Rosario. Architectural and Design Patterns for Agent-based User Modelling, Third Latin American Conference on Pattern Languages of Programming. Porto de Galinhas, Pernambuco, Brazil. August 12th, 2003. (to appear) (In Portuguese)
[33] Oliveira, Ismênia and Girardi, Rosario. Agent-based Patterns for User Modelling, Proceedings of SEMINCO 2003. Blumenau, Santa Catarina, Brazil. August 5th (2003) (In Portuguese)
[34] Omicini, A. "SODA Societies and Infrastructures in the Analysis and Design of Agent-based Systems", First International Workshop, AOSE 2000 on Agent-Oriented Software Engineering, Limerick, Ireland, January (2001) 185-193.
[35] Prieto-Diaz, R. “Domain Analysis and Software Systems Modeling”. Los Alamitos, CA: IEEE Computer Society Press, (1991).
[36] Prieto-Díaz, R. "Classifying Software for Reuse", IEEE Software, Vol. 4, nº01, January (1987).
[37] The Protégé Project, http://protege.stanford.edu.
[38] Serra, Ivo, Lindoso, Alisson and Girardi, Rosario. Specification of an Ontology-based Domain Model for Information Access in the Juridical Area. Submitted paper, (2003).
[39] Shlaer, S. and Mellor, S. "An Object-Oriented Approach to Domain Analysis", ACM SIGSOFT Software Engineering Notes, Vol. 14, Nº 5, July (1989), 66-77.
[40] Wood, M., and DeLoach, S.A. : "An Overview of the Multi-Agent Systems Engineering Methodology" in: Proc. of the First Int. Workshop on Agent-Oriented Software Engineering (AOSE-2000), Limerick, Ireland, (2000).
[41] Wooldridge, M, . Jennings, N. R. and Kinny, D. The Gaia methodology for agent-oriented analysis and design, International Journal of Autonomous Agents and Multi-Agent Systems 3 (2000).
[42] Woodridge, M., Cicarini, P. "Agent-Oriented Software Engineering: the State of the Art", In P. Cicarini and M. Woodridge, editors, Agent Oriented Software Engineering, Springer-Verlag, (2001).

# Specification Proposals for Customizable Business Components

Jörg Ackermann

*University of Augsburg, Universitätsstraße 16, 86135 Augsburg, Germany, E-Mail: joerg.ackermann.hd@t-online.de*

**Abstract.** Compositional plug-and-play-like reuse of black box components requires sophisticated techniques to specify components. A solution how to specify business components formally was proposed in the memorandum „Standardized Specification of Business Components". So far not considered was the situation when a component can be tailored to user requirements by setting parameters (customizing). In this paper we discuss how the ideas of the memorandum can be extended to formally specify customizing options of business components.

**Keywords:** Component Based Software Engineering; Business Component; Business Application System; Formal Specification; Adaptation; Customizing

## 1 Introduction

Combining off-the-shelf software components offered by different vendors to customer-individual business application systems is a goal that is followed-up for a long time. One of the crucial prerequisites towards this goal is an appropriate specification of business components, since the specification might be the only available support for a composer who combines components to an application system. The memorandum „Standardized Specification of Business Components" proposes how to specify business components formally, cf. [Turo2002]. (For details see section 2).

The construction of business applications from components promises solutions that are flexible and well tailored to user requirements. Nevertheless there will still be a need for adaptation of single components. One prominent technique for planned adaptation is customizing, that means setting of predefined parameters (for details see section 3). Is a business component customizable, its customizing options and consequences must be apparent for a component consumer and therefore need to be specified.

The memorandum did so far not consider the situation when a business component allows adaptation by customizing. In this paper we address the problem how customizable business components can be specified.

We identify which aspects of customizing need to be specified and then propose how they can be specified. We will transform customizing data to an interface information model and enable to specify customizing options within the frame of the memorandum. By this approach the basic structure and the notation mix of the memorandum will be preserved and only some enhancements will be necessary.

## 2 Memorandum „Standardized Specification of Business Components"

To enable a common understanding component specifications need to be standardized. This is the main goal of the memorandum „Standardized Specification of Business Components". The content of the memorandum is a recommendation made by the business components

working group of the German Informatics Society (GI). The English version of the memorandum is accessible via Internet, cf. [Turo2002]. Documentation about practical experiences can also be found there ([Acke2001] and [FeLT2001]).

The aim of the memorandum is to set a *methodical* standard for the specification of business components. This is achieved by identifying the objects to be specified and by defining a notation mix that is standardized, accepted and agreed upon by all participating parties. The term *specification* of a business component is defined as a complete, unequivocal and precise description of its *external* view, that is, it describes which services a business component provides under which conditions.

Based on the ideas of [BJP+1999], [Turo1999] and [Turo2001b], the memorandum defines different contract levels for the specification of components. Besides arranging the specifications' contents according to contract levels, for all of these levels a specific notation language is proposed (see fig. 1).

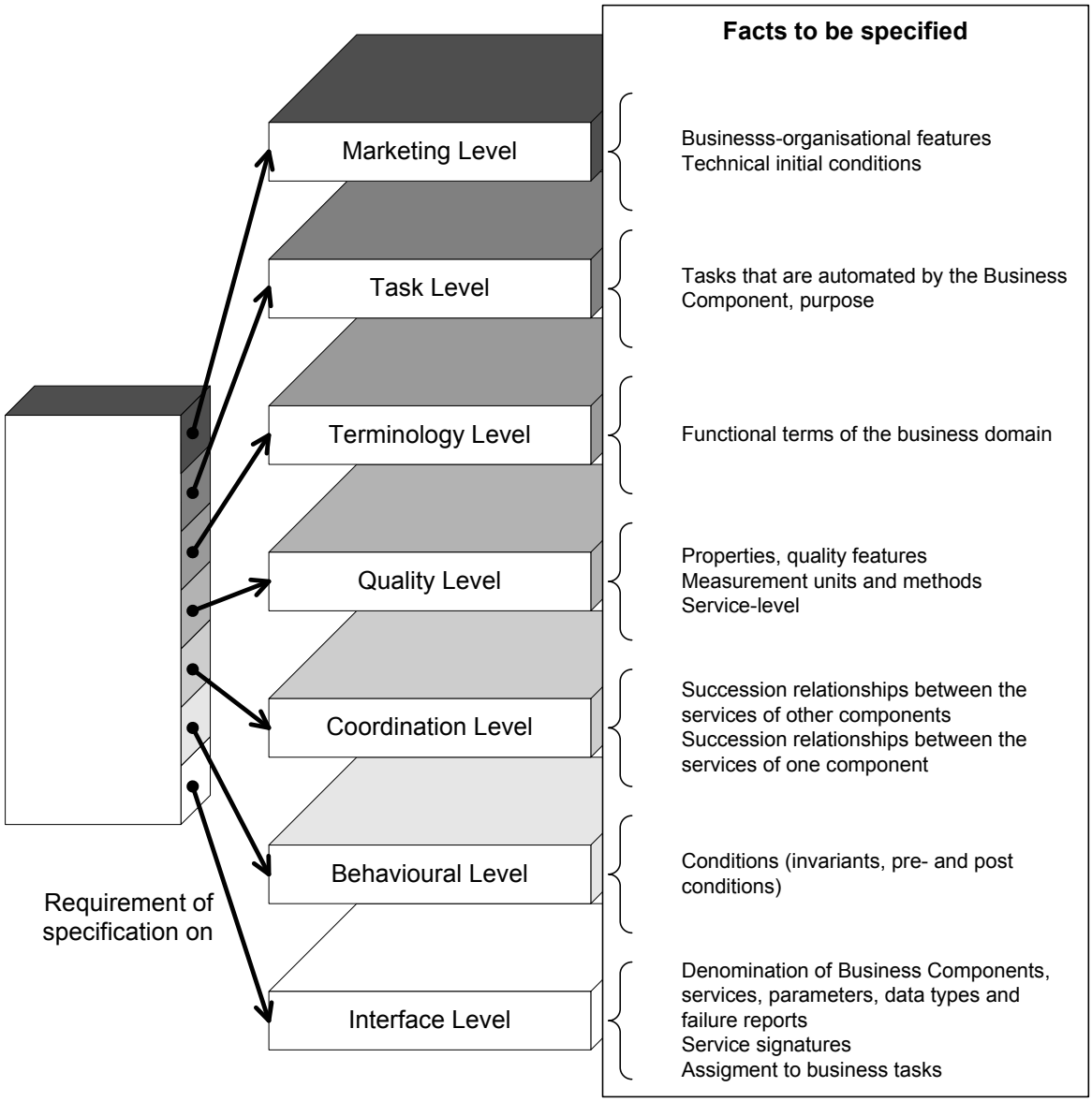


**Figure 1:** Software contract levels and facts to be specified

For a wide-spread acceptance in practice the memorandum recommends well-known and well-accepted formal notations as OMG IDL or UML. (Note that UML 2.0 details were not available at the time of discussion in 2001 and will be considered in the next version of the

memorandum.) The goal of the memorandum is not to invent new notations but to combine existing ones in order to enable a complete and standardized specification. Suitability in practice received special attention.

Below, the different contract levels are going to be characterized and supplemented with the proposed specification method. Note that we start with the more technical levels, that is in reversed order of fig. 1.

The *interface* (or syntactic) *level* contains basic agreements as names of services and data types, signatures of services, as well as the declaration of error messages. The OMG IDL [OMG2001a] has gained a broad acceptance as a standardized notation for the interface level and is therefore our notation of choice. The resulting contract guarantees that service client and service donator can technically communicate with each other. Semantic aspects remain unconsidered.

Agreements at *behavioral level* serve as a closer description of the behavior and describe how a given component acts in general or in borderline cases. As an example, we could define an invariant condition for a business component *stock keeping*, which says that the reordering quantity for each (stock) account has to be higher than the minimum inventory level. The Object Constraint Language (OCL), which complements the UML, is an example for a widespread notation to specify behavioral facts ([OMG2001b]). The UML (together with the OCL) is especially recommended to specify components, cf. [AlFr1998], [DSWi1999] and [ChDa2001]. Therefore we use the OCL as notation on the behavioral level.

Agreements at *coordination* (or synchronization) *level* regulate the sequence in which services of a business component may be invoked, and synchronization demand between its services. Here, e.g., we may lay down that a minimum inventory level has to be set before it is allowed to book on a (stock) account for the first time, or that it is not allowed to carry through more than one bookkeeping entry at the same time for the same account. The coordination level is also used to regulate the sequence in which services of *different* business components may be invoked. To restrict the number of used notations the OCL shall be used again. The OCL, however, is only partially suited to specify coordination issues. Therefore the OCL was enhanced by temporal operators, cf. [CoTu2001]. By using temporal operators one can specify requirements about the order in which different services can be performed.

As an extension to functional characteristics, we have to describe non-functional characteristics of business components. These are specified at the *quality level*. Examples for such characteristics are the distribution of the response time of a service or its availability.In all levels mentioned so far, the specification uses technical terms, which have a domain specific functional meaning (semantic), e.g. stock, inventory level, or account. Often, these terms do not have an unequivocal meaning or definition and, hence, have to be specified to guarantee their unequivocal use. The *terminology level* serves as central registry and keeps all used terms and their definitions in a dictionary. To achieve a high quality of specification, we use norm language reconstruction (cf. [Ortn1997]) to specify the respective issues. Norm language reconstruction is characterized by a dictionary of unequivocally defined functional terms and by using a rational grammar (reconstructed grammar of natural, colloquial language) to form the statements. A rational grammar consists of patterns and stencils to compose sentences.

We use the same technique to specify issues at the *task level*. There, we explain which business tasks are supported or automatically done through services offered by a business component.

At the *marketing level* we finally specify features of the business component that are important from a business-organizational point of view, e.g., (legal) contract terms, version, coarse business domain, or vendor contact persons. As notation predefined tables are used.

To summarize, the main contributions of the memorandum are given by the facts that it

- considers all specification relevant objects in one unified proposal,
- supports the connection to business domain models (business tasks, business terms),
- identifies suitable contract levels,
- proposes appropriate notation languages for each contract level and provides guidelines how to use them
- and extends the OCL by temporal operators to satisfy specification needs on the coordination level.

## 3 Customizable Business Components

The construction of business applications from components promises solutions that are flexible and well tailored to user requirements. Nevertheless there will be a need for adaptation in component based systems, cf. e.g. [Turo2001a] or [Reus2001].

The topics reuse, adaptation and variability for software in general are broadly covered in [JaGJ1997]. The authors define so called variation points, where the adaptation takes place, and identify the following adaptation mechanisms: inheritance, extensions, uses (use case reuse), configuration, parameters, template instantiation and generation. All of these mechanisms work (in accordance with the author's aim) at the implementation level.

Adaptation in component based systems is discussed by many authors, cf. [BRS+2000] or [Reus2001]. One can identify two general adaptation strategies: adaptation of component composition and architecture, and adaptation of single components. [BRS+2000] identifies the following adaptation mechanisms: wrapper, composition with adaptor, adaptation interface, inheritance and reimplementation. [Reus2001] additionally identifies superimposition and parameterized contracts.

For the adaptation of single components we distinguish between planned and unplanned adaptation. *Planned adaptation* means that adaptation opportunities are predefined by the component developer. This requires that the techniques to perform the adaptation are provided by the component developer as well. Known techniques for planned adaptation include maintenance of initializing files, maintenance of parameters in data base tables and programming of user exits. *Unplanned adaptation* means that adaptation opportunities are not predefined by the developer. Possible techniques are e.g. inheritance and reimplementation. Unplanned adaptation does not correspond to easy, plug-and-play like reuse of black box components and hence will not be considered further.

One technique for planned adaptation is customizing, that means setting of predefined parameters. Adaptation by customizing does not require knowledge about implementation details and fits well to our model of easy black box reuse. Moreover, customizing is a widely used technique for adaptation of standardized off-the-shelf business software like SAP R/3. From this we conclude that customizing is well suited for adaptation of business components and will play an important role.

Typical customizing settings are e.g. definition of organizational units (like plants), definition of control parameters (like working hours of a plant for production planning) and choices between different variants of business processes.

In the rest of the paper we focus on customizing and will now define some terms more precisely.

Under *Customizing* we understand the planned adaptation of a business component, which is based on

- assigning (at configuration time) values to predefined data fields
- that influence structure and behavior of the business component and
- have an impact on many business transactions.

Data fields provided for customizing are called *customizing fields*. We use the term *customizing data* when we want to address the sum of all customizing fields as a whole. A business component is called *customizable* if it allows adaptation by customizing.

It is sometimes not evident how to distinguish between customizing data and master data of an application, cf. e.g. [ApRi2000]. To help with the distinction we propose the collection of characteristics in Table 1. Note that the given characteristics in both columns shall not be considered as black-and-white, but rather as the opposite ends of a continuous spectrum. Table 1 is supposed to help in a classification, but in praxis not each of the characteristics needs to apply completely.

| | | **Customizing data** | **Master data** |
|---|---|---|---|
| **Time of definition** | | Configuration time (updates at run time possible) | Run time |
| **Impact on business transactions** | | Impact many business transactions (of the same type, possibly of different types) | Impact only business transactions involving the specific master data instance |
| **Frequency of entry updates** | **Insert (New)** | Seldom | Regularly |
| | **Change** | Seldom | Occasionally |
| | **Delete** | Very Seldom | Seldom |
| **Authorized for changes** | | Power user | Normal user |
| **Scope of data** | | More general | More specific |

**Table 1:** Characteristics of customizing data and master data

Customizing settings influence behavior and structure of a business component - that is they change its external view. Therefore customizing options and its consequences must be apparent for a component consumer and hence need to be included in the components specification. Moreover, customizing options often impact the functioning of a business component *substantially* and have to be set at configuration time. Thus it is also necessary, that the customizing options of a business component can be clearly recognized as such.

## 4 Extending the Memorandum to Specify Customizable Business Components

Before we can propose *how* to specify customizing options we first need to establish *what* shall be specified. There are only limited experiences for the customizing of business components [Acke2002]. There are, however, manifold experiences for the customizing of standardized off-the-shelf business software like SAP R/3. We assume that the basic customizing principles of such standardized business applications are a good starting point to analyze the customizing of business components.

Customizing standardized business software is a complex task and is discussed by many authors (for SAP R/3 cf. e.g. [KeTe1998] and [ApRi2000]). Main topics being discussed are general approach, configuration of business processes and project management. For the area of production planning and control (PPC) several investigations about management of pa-

rameters are available, e.g. [DiMH1999]. The PPC parameters of standardized business applications were determined and classified. Based on these findings tools for configuration support were developed.

The coupling between business process models and business application systems were extensively investigated by the WEGA project, cf. [FSH+1998]. Based on the results today's approach of customizing of SAP R/3 was developed, cf. [SAP1997]. Main focus of the approach was to reduce the complexity in the large. Not in the focus was, however, the question how to describe (specify) single customizing options. The so called customizing activities of SAP R/3 are documented only at a glance, but are not formally specified. Constraints and dependencies can often only be found by trial-and-error in the system itself.

In [Acke2002] we analyzed a subset of the customizing of SAP R/3. We determined what aspects of customizing need to be considered in a specification and we discussed if these findings will also be valid for business components. The results of this investigation serve as starting point for our specification proposals.

Before developing specification proposals we formulate a general goal: Specification of customizing options shall be included in the frame provided by the memorandum. The memorandums basic structure and specification techniques shall be preserved and extensions should be limited.

The specification of customizing options can be subdivided into two parts:

- specification of customizing data; that is specification of a) the customizing fields provided by a component including possible values, b) the means to manipulate the customizing fields and c) given restrictions,
- specification of the impact of customizing data; that is specification of consequences that customizing settings have on the structure and behavior of the component.

These two parts will be discussed in sections 4.1 and 4.2, respectively. In section 4.3 we apply our specification proposals to a simple example.

### *4.1 Specification of customizing data*

To describe the behavior of business services (that is operations of a business component) it is often necessary to refer to the state of the component. A mechanism to do so is given by Interface Information Models (IIM), cf. [ChDa2001] or [Andr2003]. Specification of behavior according to our memorandum also uses this technique.

We want to specify customizing aspects without changing the basic structure of the memorandum. To do so we use the idea of IIMs also for customizing. This means we must define a transformation from the customizing facts to be specified (cf. [Acke2002]) to an appropriate interface information model.

1. Adaptation by customizing works by setting values for predefined parameters (customizing data). The structure of the customizing data and its static dependencies must be described in a specification. To do so we make the following specification proposals:

- As a general rule, customizing data is represented by types in a UML class diagram that augments the behavioral level of the specification. This diagram acts as Interface Information Model.
- Customizing fields can be combined to logical units that will typically be maintained together. These units we call *customizing groups*. Each customizing group is represented by one type in the UML class diagram. The name of the type is given by the name of the corresponding customizing group.

- At one time customizing groups can have several instances. By structure they obey to the customizing group but can differ in their values for the customizing fields. These instances are naturally represented by the instances of the corresponding type.
- There are two possibilities for the value range of customizing fields:
  - The value range is fixed and not dependent from other customizing settings. Such a field is represented by an attribute of the respective type. The attribute will be furnished with the tagged value {C} and so labeled as relevant for customizing. The value range of the customizing field is expressed by the data type of the attribute. Additional restrictions to the value range can be expressed by an OCL condition.
  - The value range is customizing dependent, that is the value range of a field consists of all instances defined for some other customizing group. Such a field is represented by a binary association. Involved in the association are the type to which the customizing field logically belongs (association begin) and the type which defines the value range of the field (association end). The association is furnished with the tagged value {C} at the association end. A rolename can be used at the association end to represent the name of the customizing field. Otherwise the type name at the association end equals the name of the customizing field.
- There might be dependencies between customizing fields, that is, possible values for a field depend on the values of other customizing fields. Such dependencies are described by invariants in OCL. They will be part of the behavioral level and logically augment the class diagram.

2. There must exist means to manipulate the customizing data. We model them conceptually as services and call them *customizing services*. The properties of such customizing services must be specified. To do so we make the following specification proposals:

- Customizing services are represented as operations. All the operations manipulating one customizing group should be collected in one interface. Typically such an interface will include operations to create, change and delete instances of the customizing group.
- Customizing groups and their services will be listed at the task level. Their purpose and their effect on the business tasks are explained.
- The interfaces are shown in an UML diagram representing the component and its interfaces. The customizing services are shown as operations. These operations will be furnished with the tagged value {C} and so labeled as relevant for customizing.
- In accordance with our memorandum, the signature of customizing services are described on the interface level using the OMG IDL.
- When manipulating a customizing field there might be conditions, e.g. restriction of its allowed values or mandatory maintenance of this field. Such conditions are expressed as OCL pre and post conditions on the behavioral level.
- There might also be restrictions regarding the order in which customizing services can be performed. Such restrictions are specified at the coordination level. Again we use here the OCL supplemented with temporal operators.

To summarize, we identified specification relevant aspects of customizing data and proposed a way to specify them. To do so we did not need to define new notations but used the ones

already suggested in the memorandum. The basic structure and the notation mix of the memorandum were so preserved and only some enhancements were necessary: mandatory use of a UML diagram to represent the customizing data, use of the tagged value {C} to identify customizing relevant properties and concrete rules how to use UML (to represent customizing services and different types of customizing fields) in a standardized way.

### 4.2 Specification of customizing impact

The purpose of customizing is to influence structure and behavior of a business component by assigning values to predefined data fields. Correspondingly the setting of values to customizing data can impact the following aspects of the business component:

- the structure of its business entities and the relation between different entity types,
- the behavior of the business services, the required input parameters and the result of the service,
- the order in which different business services can be performed.

This means the pre- and postconditions of the business services might change depending on the customizing data. The impact of customizing settings is therefore concentrated on the behavioral and the coordination level of our memorandum. Customizing dependent changes on other levels (especially interface and quality level) seem possible, but will not be considered here.

In order to specify these issues, we need to consider the business data and interfaces together with the customizing data and interfaces. We use one integrated conceptual scheme to represent business and customizing data of the component (see fig. 2). This scheme will serve as interface information model for all the services belonging to the business component. By doing so, the above identified consequences of customizing settings can naturally be expressed in the pre- and postconditions of the business services. The specification is analogously to other constraints regarding these services.

Note that this approach seems only practical if the complexity of the customizing options is not exceedingly high. Although the approach is technically always possible, the constraints might become too complex to be understandable. How to enhance the approach for complex cases is still an open issue and direction of further research.

### 4.3 Specification example

As our specification example we discuss a business component *OrderProcessing* that supports business tasks from the area of production planning and control (PPC). For the sake of simplicity we only consider a very restricted set of functionality.

The component *OrderProcessing* has an interface *IOrder* that provides business related services for orders (see fig. 4). Additionally there is an interface information model (see fig. 2) where the orders managed by the components are represented by the type *Order*.

Suppose that the component *OrderProcessing* offers three customizing groups: *Plant*, *CustomerType* and *CustomerTypeProcessControl*. With the customizing group *Plant* one can define different plants (as organizational units). For the sake of simplicity we only consider two customizing fields belonging to plants: an *ID* and a *Name*. The customizing group can have an arbitrary number of instances, e.g. the plant with ID “0001” and the name “Plant Amsterdam”. The customizing group *CustomerType* serves to define different customer types whose orders shall be handled uniformly by the component *OrderProcessing*. Again we only consider two customizing fields: an *ID* and a *Name*. There can be an arbitrary number of *CustomerType* instances. Possible examples are “Business Customer” (ID = “BUS”), “Private Customer” (ID = “PRIV”) and “Anonymous Private Customer (Internet)” (ID = “ANON”).

The customizing group *CustomerTypeProcessControl* allows to define how certain processes shall be handled for customers of different customer type. To keep our example simple we consider only one control parameter: The customizing field *PaymentFirst* is of type Boolean and indicates if an order must be paid before it will be delivered. As this choice might be different for various plants, the decision can be made for each combination of plant and customer type. The customizing group *CustomerTypeProcessControl* therefore offers three customizing fields: *Plant*, *CustomerType* and *PaymentFirst*. A possible instance with the values *Plant* = "0001", *CustomerType* = "ANON" and *PaymentFirst* = true indicates, that anonymous customers buying from plant "Amsterdam" must pay their orders before they will be delivered to them. (Note that in praxis there will be more control parameters making the customizing group more useful than in our shortened example.)

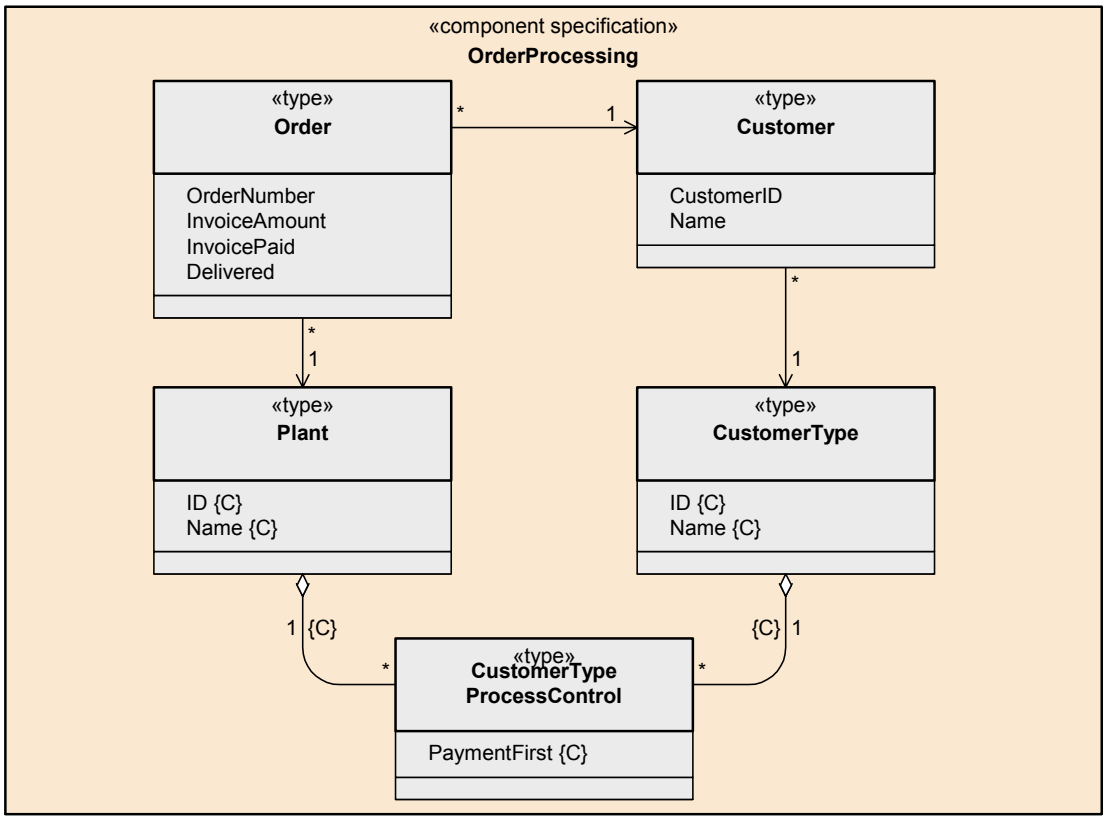


**Figure 2:** UML diagram acting as interface information model

The interface information model in fig. 2 shows the structure of the customizing data. The customizing groups *Plant*, *CustomerType* and *CustomerTypeProcessControl* are represented by types (having the same denominator). The customizing fields *ID* and *Name* of *Plant* are of type `string<4>` and `string<30>`, respectively. That is, they have a fixed value range. Therefore they are represented as attributes and they are tagged by {C} to indicate their relevance for customizing. The customizing fields *ID* and *Name* of *CustomerType* are represented in the same way.

The customizing group *CustomerTypeProcessControl* has one field with fixed value range: *PaymentFirst* is of type Boolean and also represented as attribute. The customizing field *CustomerType* of *CustomerTypeProcessControl*, however, does not have a fixed value range. Its value range is given by the customer types defined in the customizing group *CustomerType*. Thus this customizing field is represented by an association between the types *CustomerTypeProcessControl* and *CustomerType*. The association is tagged by {C} at the association end at *CustomerType*. The customizing field *Plant* is represented in the same way.

Conditions not shown in the diagram can be expressed by OCL on the behavioral level. The condition in figure 3, for example, declares that all plants defined in *Plant* differ in their ID.

```
OrderProcessing
  Plant->forAll(p1, p2: Plant | p1 <> p2 implies p1.ID <> p2.ID)
```

**Figure 3:** Example for the specification of customizing restrictions at behavioral level

Available customizing services are grouped to interfaces (according their customizing group) as shown in figure 4. There is an interface for each of the three customizing groups having the operations Create, Change and Delete. The operations are tagged by {C} to express their customizing relevance. The detailed interface specification is done on the interface level (using the OMG IDL) and will be omitted here. Restrictions regarding these services can be declared by OCL expressions.

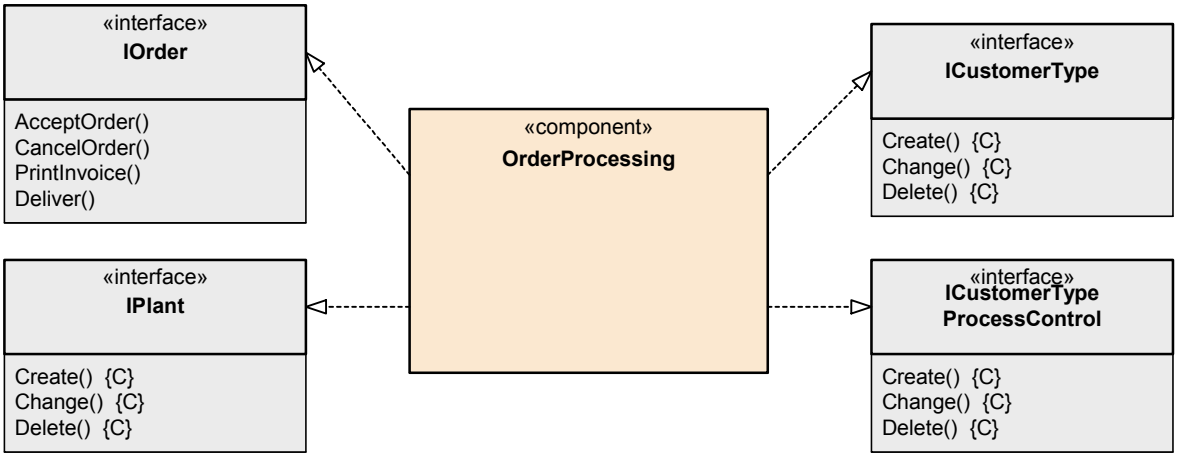


**Figure 4:** UML diagram for the business component *OrderProcessing* and its interfaces

By using one integrated interface information model for all interfaces we can readily specify the impact customizing settings have. In our example the business service *Deliver* can only be performed, if either the invoice is paid or the customer does not need to pay the invoice first. Figure 5 shows, how this is expressed on the behavioral level as a pre condition for the service *Deliver*.

```
OrderProcessing::Deliver(at:Order)
 pre: at.InvoicePaid = true or
   CustomerTypeProcessControl->any(pc | pc.Plant = at.Plant and
                   pc.CustomerType = at.Customer.CustomerType).PaymentFirst = false
```

**Figure 5:** Example for the specification of customizing impact on services

Practical experiences with the specification proposals were gained by a case study that demonstrated the feasibility of the approach, cf. [Acke2003].

## 5 Conclusions and Outlook

The use of business components can enable the building of business applications that combine the advantages of custom-made and standard software. To comply to user requirements, business components often need to be adapted. One prominent technique for planned adaptation is the setting of predefined parameters (called *customizing*). The availability and consequences of such parameters must be specified. As this aspect was so far not considered in the memorandum „Standardized Specification of Business Components" ([Turo2002]), we discussed how the memorandum can be enhanced accordingly.

We identified which aspects of customizing need to be specified and then showed how they can be specified. We transformed customizing data to an interface information model and enabled so to specify customizing options within the frame of the memorandum. By this approach the basic structure and the notation mix of the memorandum was preserved and only some enhancements were necessary: mandatory use of an integrated conceptual model (as UML class diagram) to represent business and customizing data, use of the tagged value {C} to identify customizing relevant properties and concrete rules how to use UML to represent customizing services and different types of customizing fields. Additionally, we discussed how the impact of customizing settings can be specified. How to do this appropriately in complex cases is still an open issue and direction of further research.

## References


[Acke2001] *Ackermann, J.*: Fallstudie zur Spezifikation von Fachkomponenten. In: *K. Turowski (ed.):* 2. Workshop Modellierung und Spezifikation von Fachkomponenten. Bamberg 2001, pp. 1-66.

[Acke2002] *Ackermann, J.*: Spezifikation des Parametrisierungsspielraums von Fachkomponenten – Erste Überlegungen. In: *K. Turowski (ed.)*: 3. Workshop Modellierung und Spezifikation von Fachkomponenten. Nürnberg 2002, pp. 17 – 68.

[Acke2003] *Ackermann, J.*: Zur Spezifikation der Parameter von Fachkomponenten. In: *K. Turowski (ed.)*: 5. Workshop Komponentenorientierte betriebliche Anwendungssysteme (WKBA 5). Augsburg 2003, pp. 47 – 154.

[AlFr1998] *Allen, P.; Frost, S.*: Component-Based Development for Enterprise Systems: Applying The Select Perspective. Cambridge University Press, Cambridge 1998.

[Andr2003] *Andresen, A.:* Komponentenbasierte Softwareentwicklung mit MDA, UML und XML. Hanser Verlag, München 2003.

[ApRi2000] *Appelrath, H.-J.; Ritter, J.*: SAP R/3 Implementation: Methods and Tools. Springer, Berlin, Heidelberg 2000.

[BJP+1999] *Beugnard, A.; Jézéquel, J.-M.; Plouzeau, N.; Watkins, D.*: Making Components Contract Aware. In: IEEE Computer 32 (1999) 7, pp. 38-44.

[BRS+2000] *Bergner, K.; Rausch, A.; Sihling, M.; Vilbig, A.:* Adaptation Strategies in Componentware. In: Proceedings 2000 Australian Software Engineering Conference. IEEE Computer Society 2000, pp. 87 – 95.

[ChDa2001] *Cheesman, J.; Daniels, J.:* UML Components. Addison-Wesley, Boston 2001.

[CoTu2001] *Conrad, S.; Turowski, K.*: Temporal OCL: Meeting Specification Demands for Business Components. In: *K. Siau; T. Halpin (eds)*: Unified Modeling Language: Systems Analysis, Design and Development Issues. Idea Group, Hershey 2001, pp. 151-165.

[DiMH1999] *Dittrich, J.; Mertens, P.; Hau, M.:* Dispositionsparameter von SAP R/3-PP : Einstellungshinweise, Wirkungen, Nebenwirkungen. Vieweg, Wiesbaden 1999.

[DSWi1999] *D'Souza, D. F.; Wills, A. C.*: Objects, Components, and Frameworks with UML: The Catalysis Approach. Addison-Wesley, Reading 1999.

[FeLT2001] *Fettke, P.; Loos, P.; Tann, M. v. d.*: Eine Fallstudie zur Spezifikation von Fachkomponenten eines Informationssystems für Virtuelle Finanzdienstleister – Beschreibung und Schlussfolgerungen. In: *K. Turowski (ed.):* 2. Workshop Modellierung und Spezifikation von Fachkomponenten. Bamberg 2001, pp. 75-94.

[FSH+1998] *Ferstl, O.K.; Sinz, E.J.; Hammel, C.; Schlitt, M.; Wolf, S.; Popp, K.; Kehlenbeck, R.; Pfister, A.; Kniep, H.; Nielsen, N.; Seitz, A.:* WEGA – Wiederverwendbare und erweiterbare Geschäftsprozeß- und Anwendungssystemarchitekturen. Abschlussbericht des Verbundprojektes. Walldorf 1998.

[JaGJ1997] *Jacobson, I.; Griss, M.; Jonsson, P.:* Software Reuse. ACM Press/Addison Wesley Longman, New York 1997.

[KeTe1998] *Keller, G.; Teufel, T.:* SAP R/3 Process-oriented Implementation : Iterative Process Prototyping. Addison Wesley Longman, Harlow 1998.

[OMG2001a] *OMG (ed.):* The Common Object Request Broker: Architecture and Specification, Version 2.5, September 2001. Framingham 2001.

[OMG2001b] *OMG (ed.):* OMG Unified Modeling Language Specification, Version 1.4, September 2001. Needham 2001.

[Ortn1997] *Ortner, E.*: Methodenneutraler Fachentwurf: Zu den Grundlagen einer anwendungsorientierten Informatik. Teubner, Stuttgart 1997.

[Reus2001] *Reussner, R.*: Parametrisierte Verträge zur Protokolladaption bei Software-Komponenten. Logos Verlag, Berlin 2001.

[SAP1997] *SAP (Hrsg.)*: R/3-Referenz(prozeß)modell 4.0 im R/3 Business Engineer – Zielsetzung, Inhalte, Vorgehensweise. Walldorf 1997.

[Turo1999] *Turowski, K.*: Standardisierung von Fachkomponenten: Spezifikation und Objekte der Standardisierung. In: *A. Heinzl (ed.):* 3. Meistersingertreffen. Schloss Thurnau 1999.

[Turo2001a] *Turowski, K.*: Fachkomponenten: Komponentenbasierte betriebliche Anwendungssysteme. Habilitationsschrift, Otto-von-Guericke-Universität Magdeburg, Magdeburg 2001.

[Turo2001b] *Turowski, K.*: Spezifikation und Standardisierung von Fachkomponenten. In: Wirtschaftsinformatik 43 (2001b) 3, pp. 269-281.

[Turo2002] *Turowski, K. (ed.):* Standardized Specification of Business Components: Memorandum of the working group 5.10.3 Component Oriented Business Application System. Augsburg, February 2002. URL: http://www.fachkomponenten.de. Date of Call: 2003-03-07.

Editors:

*Sven Overhage, Klaus Turowski*

Dept. of Application Engineering and Business Information Systems
Augsburg University
Universitätsstraße 16, D-86135 Augsburg
Phone: +49(821)598-4431; Fax: -4432
E-Mail: {sven.overhage, klaus.turowski }@wiwi.uni-augsburg.de
URL: http://wi2.wiwi.uni-augsburg.de