High Confidence Subsystem Modelling for Reuse

Birgit Penzenstadler, Dagmar Koss
penzenst@in.tum.de, koss@in.tum.de
Technische Universität München, Software & Systems Engineering

March 3, 2008

Abstract

Reuse of high confidence subsystems depends on their appropriate modelling and documentation. This paper discusses the different aspects that have to be considered when modelling a system and its subsystems. We propose a concrete artefact model for integrated reuse from requirements to technical architecture, which satisfies documentation demands with respect to functionality and the context assumed by the subsystem. Based on the artefact model, we describe the steps for conformity and compatibility checking at the development stage of subsystem integration and/or reuse.[1]

Keywords: subsystem, artefact, reuse, conformity, compatibility

1 Motivation

"How do we achieve reuse of high confidence software in large systems?" An answer to this question first requires an answer to the question "How do we build high confidence software?" To classify a software as deserving the predicate "high confidence", it has to be well-understood, predictable, and reliable: *Well-understood* requires a well documented requirements engineering and system's design, supported by a continuous modelling of the system. *Predictability* necessitates proper modelling with different, but consistent, interrelated views on the system to prevent modelling errors or architectural mismatches proactively. *Reliability* needs validation and verification of the modelled system.

Additionally, returning to the first question, we want the software to be reusable within large systems. Reuse is, inter alia, performed to reduce development costs. However, inappropriate or incomplete documentation sabotages this goal by increasing the effort for the (re)integration of an existing subsystem. Therefore it has to encompass clearly delimited and well documented subsystem borders.

We define a **subsystem border** as the *interface* of a subsystem *plus* the relevant surrounding *context* from the operational environment, the business domain, and organizational issues. To be able to model and document the subsystems in such a way, we need an adequate system requirements artefact model. A

[1]This work was partially funded by the German Federal Ministry of Education and Research (BMBF) in the framework of the REMsES project. The responsibility for this article lies with the authors.

corresponding subsystem model with its artefacts and explicitly modelled subsystem border information allows the developer to extract such a subsystem, including its documenting artefacts, and reuse it independent from the former environment within a new surrounding system.

Contribution: We present an approach for the explicit modelling of subsystem borders (interface plus context), thereby facilitating communication with subcontractors and reuse. We start with requirements engineering, go on to system's design and end with system (re-)integration. We present an artefact model, then discuss aspects of modelling subsystems for reuse, and propose concrete subsystem artefacts and explicit modelling of information about the subsystem borders. Finally we validate conformity and verify compatibility when reusing a subsystem. Our approach is useful for the development of high confidence COTS as well as all other types of high confidence software as soon as the system is complex enough to require decomposition into subsystems.

Related Work: There are some approaches that co-develop requirements and architecture, e.g. [23], and two that derive architecture from requirements, one for multi-agent systems [2] and an aspect-oriented method [21], but none that documents refinements of contextual issues within the subsystem.

Other work focuses on architecture design [26] and on the modelling of software libraries [12]. Close to the idea of software libraries is also software cartography [17], but without explicit consideration of compatibility and reuse. The composability and compatibility of services on the basis of components is discussed by [7] and [1].

Although the last ICSR [20] was concerned with COTS (components off-the-shelf), there was no work presented on either the representation of COTS borders with regard to ease reuse or concerning validation of conformity and verification of compatibility. The FLP component model [19] enhances systematic reuse by considering non-technical issues. However, there is no approach yet that explicitly models the borders of a subsystem.

Related work in terms of being able to make use of our ideas are the approaches to selecting adequate components for reuse, for example [18], who propose a systematic process for decision support in evaluating and ranking components, or [4], who focus on piecewise evaluation during component selection.

Outline: After introducing our background in Sec. 2, we start with our system requirements artefact model in Sec. 3 and explain subsystem border information documentation in Sec. 3.2. Then we give an example in Sec. 3.3 and present our steps for reuse in Sec. 4, before concluding with future work in Sec. 5.

2 Foundations

We follow an integrated approach of requirements engineering (RE) and system's design that develops a first sketch of the design during RE. This is based on an artefact-oriented requirements engineering reference model and a system architecture model with three abstraction layers.

The requirements engineering reference model (REM [11]) classifies the artefacts of the individual requirements engineering process into three content categories and assigns documents for them building an integrated view by the use of quality gates (not further discussed here). The general *business needs* (later on referred to as *context*) incorporate general business objectives, return-on-investment analysis, high level system vision given by the stakeholder, and so on. The *requirements* specification includes the domain analysis, the functional analysis, and quality requirements and their dependencies. The *system concept* (later on referred to as *design*) comprises a detailed functional system concept and the system test criteria. This point of view is gained from industrial best practices while its content also includes aspects known from process driven frameworks and templates (e.g. using the Volere requirements specification templates [25] and the IEEE 830-1998 standard [13]).

Our system architecture model is based on the following three abstraction layers: the *usage layer* gives a specification of the system behaviour as it is perceived at the system border by the user (black box). It is represented as a hierarchy of services, which give a formal specification of parts of the system's behaviour, and lateral relationships between them. Next, the *logical architecture* is a realization of the services from the usage model defined in the layer above. It is modelled as a net of communicating (logical) components and can simulate the system's behaviour. Usually, there is an $n : m$-relationship between the services of the usage layer and the components of the logical architecture. The third layer, the *technical architecture*, comprises a software and a hardware view, linked via the deployment description. The software is modelled in tasks that are structured in clusters and those are mapped to the hardware units. The system architecture model is explained in detail in [6].

3 Artefact Model

According to [24] the quality criteria for requirements are, inter alia, to be complete, consistent, unambiguous, and traceable. These criteria are decideable and can be evaluated. Further discussion of that aspect is out of scope for this paper, instead we assume the requirements to have sufficient quality so we can concentrate on their appropriate documentation within the system's artefact model.

3.1 General System Artefact Model

We use the system artefact model depicted in Fig. 1. It features the three content categories context, requirements, and design, and orthogonally the three abstraction layers usage, logical architecture, and technical architecture. The mapping of artefacts to content category and abstraction layer is not always unique, but we have placed them according to their main focus. Some of the artefacts may appear refined on lower abstraction layers with an increased degree of detail.

The context artefacts are - ordered according to a decreasing level of abstraction - a domain model, the business context, the stakeholder context with

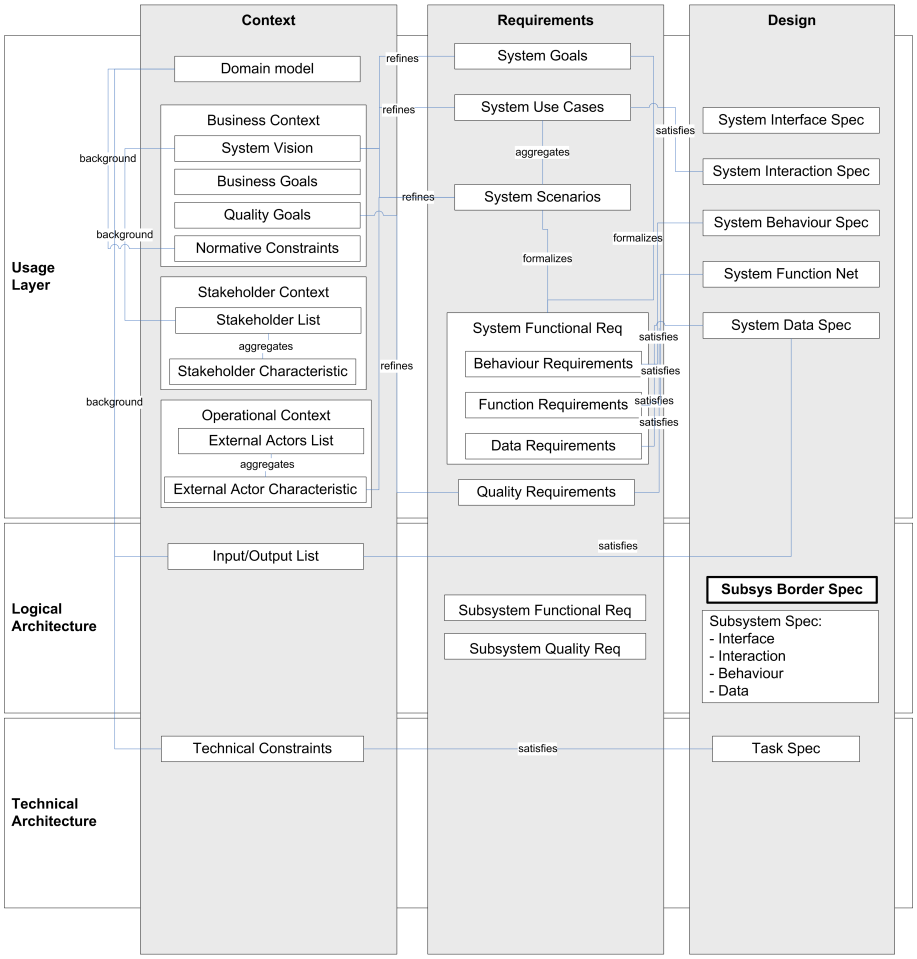

Figure 1: Artefact model

a listing and the characteristics for each of them, an operational context including a listing and characteristic of each external actor, an input/output list, and technical constraints. The business context can be further detailed into system vision, business goals, quality goals, and normative constraints.

The requirements artefacts are system goals, use cases and scenarios, functional requirements, and quality requirements. The functional requirements can be modelled as behavioural, functional, or data requirements according to the appropriateness for the system's design.

The design artefacts are interface, interaction, behaviour, and data specifications and a system's function net. The latter provides an intuitive functional overview of the system by depicting the interplay between the functions in form of a graph, but these details that are not relevant for this paper. The design artefacts will be refined on the lower abstraction layers.

The subsystems we are focusing on are modelled on the logical architecture layer. To enable a separate treatment of subsystems, there is an artefact called *subsystem border specification*. The explicit modelling and documentation of this artefact is crucial to enable integration or extraction and reuse of a subsystem. For that purpose we extract the subsystem model and complete it with the corresponding information required to document its borders as we detail in Sec. 3.3.

Due to limitations of space, we assume for the following that the system decomposition has been decided. The influencing criteria for this decision are discussed in [22].

3.2 Subsystem Borders

Garlan et al. have discussed, that architectural mismatch stems from mismatched assumptions a reusable part makes about the system structure it is to be part of. They blame this problem on conflicts of these assumptions with the assumptions of other parts, which are almost always implicit, thus they are extremely difficult to analyze before building the system. [10]

Therefore the appropriate modelling of the subsystem borders is crucial to avoid mismatches when integrating the subsystem into a (new) surrounding system. The guiding question regarding the artefact model is:
"What information can we use and what do we have to add?"
For the information that is already present, we have to decide whether the given form is already appropriate, or if we have to adapt a different form to avoid dragging along too much information.

Before reasoning on the representation of the (sub)system borders, we have to be aware of the information that is necessary to document them for appropriate retrieval when searching for solutions by reuse during development: the **interface** and any corresponding **constraints**. As we are aware of the challenges which the idea of software libraries bring with them, we do not attempt to solve all their problems, but instead focus on the adequate documentation of subsystems. The latter includes a clear description of the functionality of-

fered by the system and the functionality it requires from other system parts to perform at its optimum.

The constraints can roughly be divided into hardware and software constraints and then categorized as static or dynamic [15]. There is a great diversity of electrical and mechanical hardware constraints, but in this paper we consider only the ones that are related to software requirements.

The interface specification can be divided into a static (syntactic) and a dynamic (behavioural) part. The differentiation between static and dynamic specifications is cost-efficient, as verification of static specification usually requires less effort. So if the static specifications of the corresponding interfaces are compatible, the dynamic specifications are evaluated. A static interface specification includes:

Functions	name, header
Parameters	names and corresponding values (variable or object)
Data types	e.g. integers, strings, objects, ...
Type representation	e.g. volt, an email address,...
Value ranges	valid range for parameter
Stepping	maximum degree of increase or decrease
Pre-/postcond., invariants	e.g. voltage greater than zero
Communication	protocol and order of sent & received information

In the static specification there are some issues that, at a certain stage of development, could not be solved yet. For this reason, conditions and invariants have to appear in both listings, as some of them can be verified statically, but others only during runtime. The dynamic specification has to hold all information that can only be tested at runtime or (maybe) in a model simulation. Dynamic aspects, where compatibility has to be determined, are:

Message order	correct causal order as expected by receiver, including protocol communication messages
Message timing	e.g. message arrival within a certain time slot
Pre-/postcond., invariants	e.g. temporal logic formulae like $always(b < c)$
Ranges	conformance to limits
Stepping	conformance to min./max. in-/decrease

The constraints usually affect a greater part of the system or even the whole system, while the interface specifications described above normally apply only to the interfaces of components inside a system. Nevertheless, these constraints have to be refined for the subsystem and documented accordingly to have the complete relevant information available within the subsystem documentation. The constraints can also be categorized into static and dynamic parts and are imposed by the surrounding environment. The static constraints are for example:

Rules of conformity	e.g. standards, laws or business rules
Quality requirements	e.g. response time of database less than 20 seconds
Variability	optional parts within the subsystem

Note that laws or standards are to be considered as static in the sense of checking the conformity of the system to constraints they imply, because the internal system state at runtime does not take influence on whether the system is conform to a certain law or standard. However, on the other hand normative constraints do sometimes change during system development. This has to be

taken care of by requirements evolution and change management, but the issue is independent from the here presented concerns about conformity. The dynamic constraints affect the whole system and can only be evaluated at runtime:

Ressources	the required processing speed, required memory space
Realtime conditions	overall response time of the system, system speed
Reliability	for example Mean Time To Failure

All of the listed information should ideally already be present in the artefact model defined above. This would reduce the task to extracting the information using the appropriate filter and feeding it into a template that features fields for the interface and the constraints listed above. We are aware that this is the theory while, in practise, it will be necessary to actively accumulate the required knowledge from different sources of information.

3.3 Subsystem Border Artefacts

The first idea of how the border modelling for subsytems should be documented is depicted in Fig. 2.

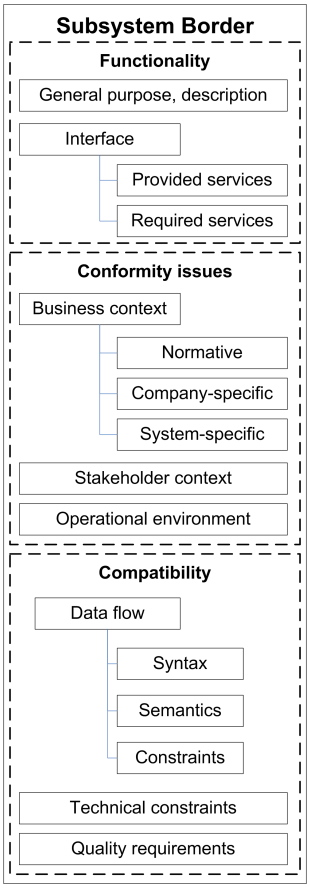

Figure 2: Subsystem Border Modelling

Functionality: For the functionality of the subsystem we extract part of the system model with explicit interface documentation. To describe the general purpose of the subsystem, we use a textual form with a short version of a system vision. This can be seen as kind of "abstract" of the subsystem that shall give a concrete idea of what the system is built for and what functionality it offers. The explicit interface documentation already reflects the realization, as it lists the services provided by the subsystem (also known as *export interface* [5]) and the services required from other subsystems by the subsystem (also *import interface*).

Conformity issues: These encapsulate contextual issues like laws, business rules, stakeholders, and operational environment. With regard to the three abstraction layers (Fig. 1) we can picture most of the content as imposed from "above", to say from the business needs and the context of the usage layer.

In detail, the business context contains normative, company-specific, and system-specific constraints. Normative constraints are implications from laws that have to be obeyed, e.g. data protection act, standards that the company wants to conform to, e.g. from ISO, and patents and licenses that are used.

Company-specific constraints are influences from business rules or information politics, e.g. servers may only be set up in countries with a special commercial agreement, and system-specific constraints derive from business goals, the system vision, and system goals, e.g. the system that shall achieve a 10% market share. Furthermore, we list implications from the stakeholder context, e.g. concerning reporting, and the operational environment, e.g. backup routines.

Compatibility: Compatibility issues contain descriptions of data flow, communication and technical constraints, and quality requirements. Most of the content in this section are implications from below with regard to the abstraction layers, to say from the software and hardware layer and the technical realization of the system. The data flow is characterized through its input and output, described through syntax, semantics, and further constraints imposed e.g. by value ranges or stepping. Communication and technical constraints are for example the message format that is used on the communication bus of the system. Quality requirements are usually the same as for the whole system. In some cases it may be possible to break them down, e.g. for response time.

Illustrating example "Travel booking system": A simple realization for the subsystem border documentation is depicted in Fig. 3 in form of a template. We use a travel booking system for illustration. The travel booking system includes all features related to travel booking, for example flight reservations and bookings, hotel bookings, check-in, reporting, accounting, and scheduling. It is designed as web service. The chosen subsystem is *flight reservation and booking*.

Subsystem Border Documentation	
Name of Subsystem	Flight booking
Purpose	The flight booking subsystem allows searching for flights and making reservations or bookings for listed results.
Interface	
Provided services	flight search, flight reservation, flight booking, cancelation
Required services	database query, user authentification, GUI
Conformity	
Business Context	Normative: data protection act Company-specific: only flights from XY are offered System-specific: store statistics on cancelations
Stakeholder Context	report cancelations to XY within 15 minutes
Operational Context	two redundant backup storage servers are running 24/7
Compatibility	
Data flow	Syntax: http request with search parameters w, x, y, and z Semantics: w: departure, x: destination, y: date, z: time Constraints: $w \neq x$ and y in future
Technical Constraints	requires SQL3 or higher
Quality Constraints	response time of database < 20 seconds

Figure 3: Template for subsystem border documentation with example "Flight Booking"

4 Reuse of a Subsystem

For the reuse of a subsystem that has been documented in the way described above, we have to ensure matching functionality as well as both conformity and compatibility. We assume, we have listed all available subsystems in our software (reuse) library with the purpose description provided in the subsystem border documentation template, as described in the box entitled *Functionality* in Fig. 2 or in the example in Fig. 3. By performing a search on the software library, we have found a subsystem with the functionality that matches our demands. For the integration into a new surrounding system, we have to check for conformity and compatibility with the new environment.

4.1 Validation of Conformity

In order to validate the conformity of the proposed subsystem with regard to the new surrounding context, we consider a guided review as the most appropriate technique. When developing a system, we move downwards through the abstraction layers, enrichening each layer with more technical detail. After implementation, the integration process follows the reverse path through the abstraction layers. Thereby we have to validate on each level, that the system under construction is conform to the specification developed earlier. This is the only way to ensure that we developed "the right system". The review has to be performed manually by checking the different areas of context and comparing them to the context of the new surrounding system. The structure of the subsystem border documentation (Fig. 2) may serve as guidance for the review and the corresponding part of the example in Fig. 3 is the box titled *Conformity*.

4.2 Verification of Compatibility with (U)CML

Verification ensures that we developed "the system right", meaning that we verify that our implemented technical solution is compatible to the surrounding environment. Therefore we have to check the static (syntactic) as well as the dynamic (behavioural) fitting of a component with its environment. (U)CML - (Unified) Compatibility Modelling Language has been developed to solve static compatibility issues. (U)CML enables to model systems from scratch, specify the interfaces of the components, compare corresponding interfaces, and check their compatibility. It is also possible to verify the exchangeability of components. Referring to our example in Fig. 3, we can verify the box headed *Compatibility*. Similiar to interface automata [8], (U)CML takes an optimistic view on compatibility, that means, interfaces do not have to be a perfect match to be compatible, but in contrast to interface automata this is not achieved by finding an environment which is compatible (via the game theory). Instead, it is defined by applying compatibility rules to the in- and output to expand the compatibility matching range. Furthermore, interface automata are restricted to software, whereas (U)CML can also model hardware and electrical aspects of a system.

(U)CML Models consist of one system package on the highest level, which defines and capsules the system. It consists of packages (containers) and com-

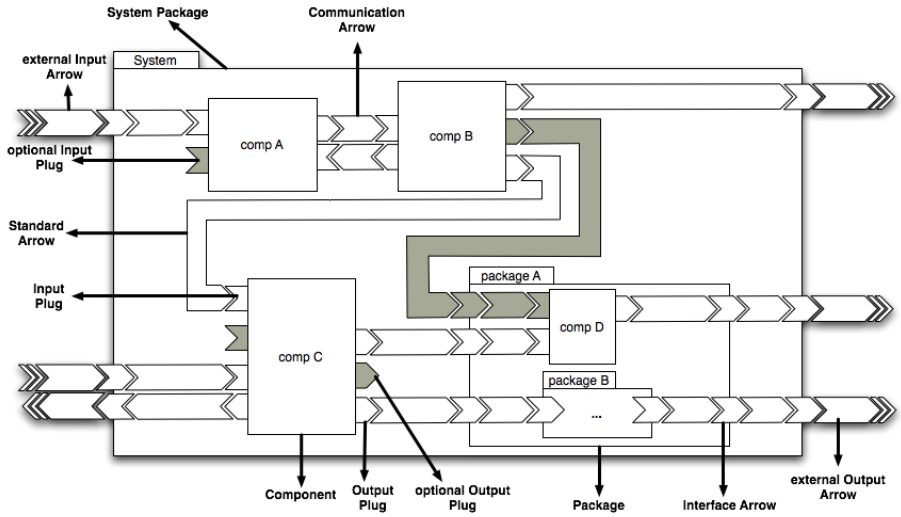

Figure 4: A UCML model

ponents. Packages can contain other packages and components, components cannot be decomposed any further.
Packages and components are connected via arrows and the endpoints on components are called plugs (in- and outputs). Arrows can only have one source and one destination point and cannot be recursive. For bi-directional communication there is a special communication-arrow. Information about the modelling objects is stored inside description fields. Description fields store structural information. Additionally, the component description field holds invariants and the plug description field holds part of the interface specification for each in- and output of the component. This can be for example a type, a variable name, or a function. For a detailed introduction, see [16].
The two outstanding features are compatibility rules and optional in- and outputs. The latter is especially useful when using the same components in different systems. In- or outputs do not have to be connected necessarily, as the system still works without errors, and therefore the component is (re)usable for more than one system.
The other feature is the assignment of compatibility rules to the whole system or to a specific in- or output. Rules are, e.g., that an integer output can be matched to a long input or that a certain output value range corresponds to a certain input value range. With these rules it is possible to expand the compatibility check for corresponding out- an inputs. Rules can also be added as pre- or postconditions and invariants. They can be written in CCL, the Compatibility Constraint Language, which is similar to OCL [14].

Static Compatibility Tests: (U)CML allows two different kinds of compatibility tests. One is the structural correctness of the (U)CML model, for example, if every mandatory output plug is connected to a mandatory input plug, or if every component has at least one input and one output plug. The other one are compatibility tests, which are mainly performed by evaluating the corresponding description fields of the output- and input-plugs. If they match,

the plugs are syntactically compatible, if they do not match, the compatibility rules have to be evaluated to check, whether they are compatible after applying the rules.
After the static checks, the dynamic checks have to be performed. This is currently being implemented in (U)CML by adding message sequence charts that model the behavior of the components [9].

5 Conclusions and Future Work

We have presented an approach to the explicit modelling of subsystem borders that facilitates communication with subcontractors and reuse. We have introduced a concrete artefact model for the integrated reuse from the requirements to the technical architecture. It provides for documentation demands with respect to functionality and the context assumed by the subsystem.

Based on that artefact model, we have described the steps for conformity and compatibility checking at the development stage of subsystem integration and/or reuse. The approach presented in this paper will be validated and applied more extensively to a case study within the REMsES research project [3].

Future work is to further detail the border documentation, as the proposed template is still scarce and needs to be extended with appropriate documentation techniques and tracing methods.

In parallel, we are currently working on an analysis of the criteria for the decomposition of systems, the influences and trade-offs between them, and the documentation of the rationale leading to such design decisions to further improve the trust for reuse of high confidence systems modeling.

Acknowledgements: We would like to thank Daniel Mendez-Fernandez, Felix von Ranke and the anonymous reviewers for their helpful comments and feedback on earlier versions of this paper.

References

[1] C. Attiogbé, P. André, and G. Ardourel. Checking component composability. In W. Lwe and M. Sdholt, editors, *Software Composition*, volume 4089 of *Lecture Notes in Computer Science*, pages 18–33. Springer, 2006.

[2] L. Bastos, J. Castro, and J. Mylopoulos. Deriving architectures from requirements. In *Requirements Engineering Conference. RE 2006. 14th IEEE International*, pages 332–333, 2006.

[3] Berghof Automationstechnik GmbH, DaimlerChrysler AG, SSE Universität Duisburg-Essen, and Software and Systems Engineering Technische Universität München. Project REMsES. http://www.remses.org, 2007.

[4] J. Bhuta and B. Boehm. A method for compatible cots component selection. In *COTS-Based Software Systems*, 2005.

[5] M. Broy. A core theory of interfaces and architecture and its impact on object orientation. In R. H. Reussner, J. A. Stafford, and C. A. Szyperski, editors, *Architecting Systems with Trustworthy Components*, volume 3938 of *Lecture Notes in Computer Science*, pages 26–47. Springer, 2004.

[6] M. Broy, M. Feilkas, D. Wild, J. Hartmann, J. Grünbauer, A. Gruler, and A. Harhurin. Umfassendes Architekturmodell für das Engineering eingebetteter software-intensiver Systeme. Technical report, Technische Universitä München, to be published.

[7] M. Broy, I. Krüger, and M. Meisinger. A formal model of services. *ACM Transactions on Software Engineering Methodology (TOSEM)*, 16(1), 2007. available at http://doi.acm.org/10.1145/1189748.1189753.

[8] L. de Alfaro and T. A. Henzinger. Interface automata. In *FSE '01: Proceeding of the 9h Annual Symposium on Foundations of Software Engineering*, pages 109–120, Vienna, Austria, 2001. ACM Press.

[9] C. Eckl. Analysis and adaptation of MSCs for the examination of behavioral compatibility. Master's thesis, Technische Universitä München, 2007.

[10] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17–26, 1995.

[11] E. Geisberger, M. Broy, B. Berenbach, J. Kazmeier, D. Paulish, and A. Rudorfer. Requirements Engineering Reference Model (REM). Technical report, Technische Universität München, 2006.

[12] J. Hunt and J. McGregor. A model for software libraries. In R. P. Institute, editor, *Library-Centric Software Design*, 2005.

[13] The Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY 10017-2394, USA. *IEEE Recommended Practice for Software Requirements Specifications (IEEE Std 830-1998)*, 10 1998.

[14] D. Koss. CCL reference. http://www4.in.tum.de/ koss/da/compMSCAdapt.pdf, 2007.

[15] D. Koss. *Kompatibilität und Kompatibilitätsmanagement.* PhD thesis, Technische Universität München, to be published 2008.

[16] D. Koss and M. Brandstätter. (U)CML - a modeling language for modeling and testing compatibility. In *Proceedings: Software Engineering and Applications*, 2007.

[17] J. Lankes, F. Matthes, and A. Wittenburg. Architekturbeschreibung von anwendungslandschaften: Softwarekartographie und ieee std 1471-2000. In *Software-Engineering, Essen 2005*, pages pp. 43–54, 2005.

[18] H. Lin, A. Lai, R. Ullrich, M. Kuca, K. McClelland, J. Shaffer-Gant, S. Pacheco, K. Dalton, and W. Watkins. Cots software selection process. In *Commercial-off-the-Shelf (COTS)-Based Software Systems, 2007. ICCBSS '07. Sixth International IEEE Conference on*, pages 114–122, Feb. 26 2007-March 2 2007.

[19] H. Mei. A component model for perspective management of enterprise software reuse. *Ann. Software Eng.*, 11(1):219–236, 2001.

[20] M. Morisio, editor. *Reuse of Off-the-Shelf Components*, volume 4039 of *LNCS*. Springer, July 9th International Conference on Software Reuse, ICSR 2006 Turin, Italy, June 12-15, 2006 Proceedings.

[21] E. Navarro. *ATRIUM Architecture Traced from Requirements by Applying a Unified Methodology.* PhD thesis, University of Castilla-La Mancha, 2007.

[22] B. Penzenstadler and D. Mendez-Fernandez. System decomposition for distributed development. submitted for ICSP'08.

[23] K. Pohl and E. Sikora. COSMOD-RE: Supporting the co-design of requirements and architectural artifacts. *RE*, 0:258–261, 2007.

[24] M. Recknagel and C. Rupp. Meßbare Qualität in Anforderungsdokumenten. *Automotive Vertikal*, 2:12–17, 2006.

[25] J. Robertson and S. Robertson. Volere: Requirements specification template, 2006. `http://www.volere.co.uk/`.

[26] R. Wojcik, F. Bachmann, L. Bass, P. Clements, P. Merson, R. Nord, and B. Wood. Attribute-driven design (ADD). Technical Report CMU/SEI-2006-TR-023, CMU SEI Pittsburgh, 2006.