University of West Bohemia
Faculty of Applied Sciences

Properties and Verification of Component-Based Systems

Habilitation Thesis
(the core text)

Přemysl Brada

Plzeň, October 2011

Contact:
brada@kiv.zcu.cz
http://www.kiv.zcu.cz/~brada/research/

Abstract

Component-based software engineering is concerned with the composition of software systems from well-defined building blocks, software components. They embody the principles of modularity and information hiding in perhaps the clearest form so far and can be in theory treated as pure black boxes, accessed only according to the specification of their interface. Third-party composition and substitution is one of the key consequences (and also benefits) of this nature of components.

For these benefits to materialize, the concrete component models must however adhere to the principles and provide support for safe component substitution. Several studies have shown that this is not always the case, leading to problems in component application consistency caused by hard-to-analyse hidden dependencies and weak verification mechanisms. Safe component substitution is particularly challenging in the case of component models enabling dynamic evolution of application architecture.

This thesis summarizes author's contributions to this field of research in two complementary areas. First, it presents the work on conceptually clean component models with complete type system representation of component interface amenable to automated processing. Secondly, it shows that efficient consistency verification mechanisms can be built on this basis, utilizing the unique characteristics of components and helping to reduce application failures caused by incorrect component composition or architectural reconfigurations, in particular substitution.

The thesis has the form of a collection of nine articles for which the textual part introduces key concepts and provides a contextual commentary. Four papers were written solely by the author, the remaining six were co-authored with at least 1/3 contribution by the author. The presented methods were validated by applications on industrial component frameworks and most of the works contributed to the results of several national research grants.

Contents

1 Introduction

Component-based software engineering is an area of research and practice concerned with the composition of software systems from well-defined building blocks, software components. This term was coined already in 1968 by McIlroy, who noted that "software production in the large would be enormously helped by the availability of spectra of [components]" [60] and called for techniques to produce families of standardized, reusable, proven software parts with specified properties. His strikingly clear vision has been slowly becoming a reality during the last 15 years.

The evolution of component-related research began within the investigations of software architectures [73, 81], with deeper roots in the principles of encapsulation and modularity [72]. Software architecture research started from the need to formally describe high-level structure of software systems and lead to defining the notions of architecture description lanugages (ADLs) [61] and architectural styles [94]. They define the constituent elements of the architecture, in particular components and connectors, and the rules of their correct composition.

In this context, components are understood mainly as elements of logical architecture. Soon a recognizable stream of research established itself with the primary focus on components per se [90, 25]. It looks into their fundamental properties as well as other aspects like deployment on target systems, lifecycle, or distribution of components as artifacts. Thus the notion of component and its impact on system building has gradually reached better understanding in the research community [91, 34] and software components began (largely independently of research though) being used for building industrial software systems [63, 88].

1.1 Motivation and Focus of this Thesis

Software components embody the principles of modularity and information hiding [72] in perhaps the clearest form so far. They can be in theory treated as pure black boxes which can be accessed only according to the specification of the component interface. This approach leads to good compositionality [2] of component-based applications, including the benefits of easy replacement of application parts.

Any move away from the guiding principles of information hiding then incurs the risk of diminishing the benefits just mentioned [7]. It is therefore important to study the properties embodied in component models and work towards cleaner ones to achieve greater flexibility and stronger compositional properties of component-based applications.

Third-party composition and substitution are one of the key consequences (and also benefits) of the black box nature of components [91]. To enable both human-oriented and programmatic analyses and manipulations of component-based architectures, including composition and substitution, a well-defined formal model of component interface is needed. At present however, individual component models mostly use each its own representation of the component interface – very often an ADL – depending on the features it can contain [35]. A sufficiently general yet descriptive meta-model and representation of the interface would improve architectural analyses and model interoperability.

Among the most important low-level requirements of component-based applications is that they preserve their architectural consistency [94] during component deployment and at run time. This implies that the infrastructure should check the mutual conformance of component interfaces on the client-supplier bindings as well as of any prospective replacement components. This is a challenging task [92] due to the rich variety of component interface specification formalisms, ranging from semi-formal UML models to rigorous ones based e.g. on temporal logic. It is true that the more precise the specification is the better guarantees of consistency we get; however, especially the run time consistency checks need to consider computational complexity and resource usage besides the formal strength of the methods used.

The methods, models and formalisms we present here concern various aspects of these issues, with the common theme to provide better component specifications and their handling for improved consistency of component-based architectures in realistic circumstances. The focus is mainly on the foundational properties of components, their type-based representations, and safe component substitution.

1.2 Structure of the Document

This thesis summarizes – in the form of a commented collection of published articles – author's contributions to the research of component-based software systems in the above areas. The two following chapters present in detail the concepts, issues and state of the art concerning the understanding of components as black box entities plus their interface representation (Chapter 2), and methods for verifying consistency of component-based architectures (Chapter 3), thus setting author's work in the context. They additionaly provide a commentary to the author's contributions which are listed in Chapter 4 and included thereafter.

We would finally like to add a few notes on the overall patterns that can be discerned in this thesis. The contributions are separated into two themes, which is reflected in the set of collected papers being split into two subsets and denoted by A1-A4 and B1-B5 identifiers. However, the topics are mutually related and often overlap which is why there are (very few) forward references within the text. Because the author believes that research should – besides enhancing the fundamental knowledge of the world – make impact to the current state of practice, cf. [39], the work on the properties and methods presented here also lead to several results which stem from their application to industrial component implementations.

2 Software Components: Models and Properties

From the onset of software architecture research, the concept of component has been central to the structural view of architectural models. In this chapter we review the origins, current the state of the art and author's contributions concerning (1) the basic properties of software components and how they are reflected in component model design, and (2) component interface specifications and their representations amenable to automated processing.

2.1 General Concepts

The debate about what *component* is and isn't from the late 1990s [90, 27] resulted in the terminology becoming largely settled, as witnessed by the classic textbooks by Szyperski [91] and Taylor et al [94]. Definitions of the term mostly emphasize that component should be an independently deployable and composable black-box software entity which explicitly declares both the features provided to its clients and required from its deployment environment.

The types of components permissible in an architecture, the particular features of components and the means of their specification, and the ways in which they can be interconnected and interact, are prescribed by a *component model* and enacted by a component *framework* [7, 35]. Dozens of component models have been created by both the research community (e.g. Darwin [56], SOFA [28], ProCom [78] or Fractal [15]) and industry (e.g. JavaBeans, Koala [97], CORBA Components [68] or OSGi [69, 71]).

Component models and consequently components can be further distinguished according to several characteristics. Most importantly, *hierarchical models* like Darwin or Fractal allow a component to be composed of sub-components whereas flat models like OSGi use only a single level[1]. Concerning component "weight" the approaches range from components comprised of a single function (ProCon) through single-interface components (EJB) to the prevailing case of components with many features on the interface.

Further, some models use only design-time components which get composed into a monolithic application (Koala) while most models use components as entities with separate identity during the whole *lifecycle* (design, deployment and run-time) [35]. Similarly we can distinguish *static architectures* (e.g. CORBA Components or original SOFA) which fix the set of components and their bindings at design or deployment time, and *dynamic architectures* like OSGi which allow the architecture to evolve at run time. This class also relates to service oriented architectures [47] whose dominant aspect is dynamic lookup, binding and orchestration of services.

Lastly, the specification of component interface or *surface*[2] can have many forms

[1] It is interesting to note that most industrial component models are flat, with the notable exception of Koala, whereas research models are almost invariably hierarchical. This may be related to the need for a manageable framework complexity in the industrial models.

[2] This term denotes the complete structure of a component visible and accessible from outside; we prefer it to the more customary term "interface" so as to avoid confusion with the language construct used in many current programming languages.

[35] but in principle presents the software component as a set of (usually named) features: $C = \{f_i\}, i \in \mathbb{N}$. An important discriminator is the *role* of these features, that is whether f_i is *provided* to component's clients or *required* to be bound for C to function properly. Using the concept of *contract levels* [62, 12, 94] we can orthogonally decompose the specification into four parts:

- *syntactic* — covers the signature and interface definition language constructs, declaring the existence of separate *features* and *properties* on the component interface; a foundational (*sine qua non*) layer;
- *semantic* — concerns the meaning of features often specficied by the expectations and effects of individual features, e.g. via pre- and post-conditions;
- *interaction* — defines the allowed sequences of interactions with the component, e.g. in the form of state models or event traces;
- *extra-functional* — quantifies or enumerates various qualities of the component and/or its individual features.

It is necessary to note that the terminology in this area is not unified: the first level is alternately called 'signature level' [95], semantic level is sometimes called 'behavioural' [94], other terms for the interaction level are 'protocol' [96] or 'synchronization', and extra-functional level is often called 'quality of service' (QoS) or 'non-functional properties' level. Furthermore, the term "behaviour" is at times used to denote the semantic level [55] while other times it denotes the interaction level [76].

Below we present the state of the art and discuss in detail the aspects of component models where the author of this thesis has contributed to the current state of research – fundamental issues related to component's foundational properties, their type representation, and means of enriching component specifications with extra-functional properties.

2.2 Black-box Nature of Components

The fact that a key property of components is their black-box nature, essentially manifestation of the well-known information hiding principle [72], is not agreed upon uniformly. For example, Crnkovic et al in their study [35] seem to prefer a more general definition in which the formulation of core component properties is delegated to the component model.

Since composition is the *raison d'être* of components, properties and mechanisms that enable composition are crucial in component model design. Being a black box is one of them, and its practical manifestation – context dependencies explicitly specified as required features – is a key supportive abstraction in this respect [91]. To reason about component properties and to compose components correctly, we should therefore rely solely on the specification of their surface (explicitly excluding the internals of a particular implementation from such reasoning).

It has also been shown that the lack of explicit specification of dependencies leads to property leaks or *hidden dependencies* [7, 98]. In such case there exists a binding of components C_x and C_y which is not established as a relation between a required feature $f_r \in C_x$ and a provided feature $f_p \in C_y$. Component models which allow or even necessitate hidden dependencies – due to weak specification means – pose problems for studying and modifying application architecture [38, 4].

Several studies analyzed component models for criteria including the black box nature [79, 106, 35]. Although it is hard to generalize, the results tend to agree that most research component models satisfy the black-box property quite well while industrial components tend to be rather weak in this respect (especially those enabling dynamic architecture evolution).

2.3 Representing Component Interfaces

Once the component can be treated as a black box, it is conceptually easy to obtain a representation of its complete surface. Such representations are however not usually formalized beyond the standard notion of a set of features or ports [35] or a meta-model specific for the given component model [28].

This lack of formal representations has several undesirable consequences. It hinders the evaluation of component properties such as those discussed in Chapter 3 of this thesis. Finding a suitable component based on actual or expected usage is difficult without rich models amenable to automated processing [79, 82] which diminishes their reusability potential. Lastly, dynamic architectures and adaptive systems need precise and rich representations in order to efficiently create and modify application architectures [66, 99].

2.4 Enriching Component Specifications with Extra-Functional Properties

The concept of extra-functional properties (EFPs) like performance or security is certainly not a new one, as are not the efforts in creating their adequate specifications. The need for such properties has been long recognized in the software engineering community, as exemplified e.g. by the FURPS scheme first used by Hewlett-Packard [44] and incorporated into the IBM Rational Unified Process framework [53] or by the UML[3] QoS and Fault Tolerance profile [1]. The research community has also been working on EFP formalisms for a considerable time [41, 77, 65, 31, 43] which lead to a proliferation of models and specifications.

Most recent and current research component models define elements at the syntactic and (less often) semantic or protocol contract levels – interfaces, ports, events, bahviour protocols, etc. Only very recently researchers have started investigating the use of extra-functional properties in the context of component models [67, 10, 107, 89], especially in the real-time and embedded systems domains due to the need to evaluate timing and performance properties [52]. The need for (and lack of) these properties has however been recognized from the early years of architectural research [61].

The problem brought by the diversity of proposed EFP formalisms is the lack of a universally accepted meta-model and resulting interoperability issues. Even within the individual approaches, authors usually assume internal consistency of the set of properties attached to a set of software modules. However, this becomes an issue in

[3]Unified Modeling Language

the area of component-based systems which enable composition of components from independent providers. Despite some standardization-related proposals [6], there have been few research efforts addressing this need for the consistency of EFP definitions and interpretations across multiple vendors and domains.

2.5 Contribution

Black-box nature Based on our experiences with research component models and building applications with several industrial ones, we defined a set of criteria to evaluate how well the black-box property is supported in the subject area of software components. They are described in paper A2 [20] which also gives the argument for the importance of the black-box property.

These criteria were used to analyse a set of component models and mutually influenced the design and implementation of our experimental component model called CoSi, described in paper A1 [17]. It preserves the benefits of dynamic, service-oriented architectures while adhering to the fundamental principles of component-based programming.

Component type representation Inspired by the implicit use by advanced component models (e.g. SOFA2, CORBA components), we defined a formal model of the component surface in the form of *component type* which aggregates the surface elements (representation of the features and properties at all contract levels) while preserving the important distinction between provided and required elements [21].

On this base we then defined the novel notion of component's *contextual complement* which represents the deployment context of a component in architecture. Our ENT meta-model [108] uniquely enhances this component type representation with semantic (ontology based) information and a model of inter-component bindings. Both of these formal models are accompanied by representations that enable automated processing of their data.

In paper A4 [9] we discuss the issues that can be encountered when the type representation needs to be obtained from the distribution packages of real-world components. Our contribution in this area is the method for reconstructing the type representation of required elements, since their data types are inherently not defined within the component's package. The problems related to obtaining parts of type information due to model inadequacies are discussed also in paper A2 [20].

Extra-functional property specifications For this level of component interface contract we have proposed a rich model of extra-functional properties which can be attached to components and their surface features. The model, formally described in paper A3 [48], is augmented by a system of federated registries which store EFP definitions and value assignments. This approach allows to define properties and their symbolic values in one place and use them in different domains and target components, while preserving consistent property semantics.

We have subsequently linked the model to component surface specifications [50] and extended it with algorithms for checking EFP compatibility (discussed in Section 3.2 further below).

3 Consistency of Dynamically Evolving Component Architectures

Consistency is a key property of any software system, meaning that its constituent elements do not contradict each other [94] so that they can cooperate correctly and the whole system provide the intended functionality. Since the ability to be composed is central to components [16], it is an important question how to ensure that a resulting composed architecture is a consistent one.

Dynamically evolving systems are those which allow "modifying the architecture and enacting those modifications in the system while the system is executing" [61]. They were brought into light by the research into architectures and components e.g. by the works of Magee and Kramer [57], Plasil et al [75] or the ArchJava team [5]. Their ability to flexibly change application structure – that is, evolve their architecture at run time – makes them well suited for many scenarios where larger software systems are deployed, including the case of service-oriented and adaptive systems.

The downside of these positives is that maintaining consistency and correctness of dynamically composed architectures is extremely challenging [92, 66]. Unexpected system failures due to consistency violations are at the same time one of the worst to prevent, detect and repair; especially in the world of component-based systems which allow the composition of black-box components independently created by various providers. Measures that help to prevent consistency violations to arise are therefore vital for correct functioning and wider success of dynamic component-based systems.

In the following two sections, we first clarify the core concepts used in this area and then present our contributions set within the wider context of the field.

3.1 Ensuring Architectural Consistency: Basic Concepts

The concept of *architectural consistency* (also "integrity") is used at the meta-level of whole application architecures. Here, architectural styles [3, 94] and component models define the rules for correct composition of components i.e. the allowed types of components in a concrete architecture, proper bindings of their elements and the required global static and dynamic properties [86] of the whole system (e.g. completeness or liveness). A consistent system is consequently one which is composed according to such rules.

At the level of bindings between the components (via surface features) within an architecture, we require *contract consistency*. This is achieved when the parties of a contract adhere to all its aspects – are of the same or matching type, their behaviour is equivalent or in subsumption relation and their properties are in an inclusion relation. We especially need to verify that the "requires" part of a component's contract is satisfied.

Due to the completeness of component surface specification, we can use the *assume-guarantee principle* [64, 83] to check each component's correct functionality within an architecture – if the component's "requires" contract part (the assumptions) is satisfied then it will guarantee the "provides" part according to its specification. Component-based architectures thus enable *compositional reasoning* [36, 2] – if we establish that each component's contract is satisfied individually then the whole composition is correctly assembled. Thus architectural consistency can be induced from that of individual components and their bindings.

From a broader perspective, contract consistency ensures the *interoperability* of components which can be defined as "the ability of two or more entities to communicate and cooperate despite differences in the implementation language, the execution environment, or the model abstraction" [96, 82].

A related term, *compatibility* is sometimes used to denote the same concept; synonyms "conformance" [94] or "horizontal compatibility" [11] are also sometimes used. However, the term "compatibility" alternately expresses the ability of one revision of an entity (component) to safely replace a preceding revision, i.e. the notion of "backward compatibility" which in turn is a special case of "vertical compatibility" [*ibid.*]. Due to this ambiguity, we avoid using the general term unless its meaning is clear from the context.

Substitutability is finally the most general concept (synonymical with vertical compatibility) expressing the ability of a replacement entity C_R to fully substitute a current one C_C (denoted $C_R \preceq C_C$); the replacement entity can be unrelated to the current one in terms of its origin and revision history. As noted above, a special case of substitutability is *backward compatibility*. These notions follow the well-founded general principle of substitutability introduced by Liskov and others [54, 101].

In the area of component-based systems, its applicability is enhanced by the completeness of (semi)formal component contract specification as discussed in Chapter 2. This property facilitates the use of *matching* methods [102] as a basis of the checks.

To ensure architectural consistency during initial composition of a (static) architecture, the adherence to architectural style respectively component model rules must be checked, followed by horizontal contract consistency checks. The measures applicable during run time architectural reconfiguration present a wider range of options, mostly involving vertical substitutability verification:

1. Do not allow run time architecture modifications (e.g. UniCon [80]).
2. Constrain changes to a well-defined set governed by architectural rules and patterns (e.g. Darwin [56], SOFA2 [46]).
3. Check that the (unconstrained) changes performed do not violate architectural rules (e.g. C2 [93] or service-oriented architectures [42]).
4. Let the reconfiguration happen and hope for good luck (e.g. plain OSGi [71]).

Our work focuses on the level of individual components and their contracts which are part of the second and third options as discussed in detail below.

3.2 Horizontal Compatibility

Methods for verifying horizontal compatibility can be primarily classified by the level of interface contract on which they operate; and at a second tier, by the means used to determine it — static or dynamic. As Stuckenholz observed [85], the higher up the contract level, the fewer methods are available as researched results.

Syntactic compatibility works with the signature part of component interface specification, in effect with its type representation (see Section 2.3 above). The early works on software architectures like Shaw et al [80] already mention the need for type checking at the granularity of architectural descriptions.

Compatibility is determined by establishing subtype relation on bound interface elements, possibly with relaxations as introduced by Zaremski and Wing [103] and used by e.g. Flores [40]. Some approaches use a constrained component model or language subset to facilitate full formal proof of subtyping [104, 14].

Semantic and interaction compatibility has been researched continuously using many formal models of behaviour (cf. surveys by Bradbury and Zhang et al [24, 105]). The component behaviour model can be specified either a-priori, or derived by reflection or run-time observation from component's implementation [59, 30]. Models on both sides of inter-component bindings are in most methods compared by static model-checking [32] to establish whether they are in *refinement* relation and thus compatible [76, 13]. Alternately, they can serve as a basis for generating a test suite which is executed to verify compatibility dynamically [58, 40].

There are two well known issues with methods of behavioural compatibility verification. First, the state space of the models tends to grow exponentially with model size [33] which makes the methods difficult to use with large systems, often found in real-world applications. Second, semantic and behavioural specifications are rarely available for current and near-future component systems [35] apart from specialized domains (hard real-time, high availability systems).

Extra-functional properties compatibility has only been gaining attention recently, since even the basic prerequisite of EFP models and specifications is an area of active research. The ability to specify, compare and verify component and service extra-functional properties [51, 84, 107] is important from systems point of view, because EFPs in the areas of performance or security have significant impact on application architecture and implementation [89] as well as on the configuration of its deployment environment.

3.3 Component Substitutability

Verification of component substitutability uses methods linked to the contract levels discussed above. Furthermore, the methods can vary depending on the size of the current and replacement component sets: from basic 1:1 replacement (single component for single component) to full M:N substitution (larger component sets).

In the general case of an "open world scenario," there need not be any relation between the current and replacement components; this requires to check substitutability in-situ for the pair of components or component sets under substitution. Approaches for the basic substitutability checks correspond closely to the compatibility verification methods discussed above. The M:N substitution has been researched by e.g. Stuckenholz, Desnos and others [87, 37, 30].

All the methods discussed so far consider the current and replacement components as closed sets, verifying their substitutability "in isolation". This approach is limited by the strict covariance and contravariance it enforces on the provided and required sides of component interface, respectively. While alternative approaches have been proposed based on global (rather than local) integrity checks [86], we are not aware of advanced methods working at the level of available component interface specifications.

When the replacement component C_R is a downstream revision of a current one C_C, general substitutability is reduced to backward compatibility[4] which is verified on component updates – this is useful for the common situation when the component revision stream is controlled by a single vendor (a "closed world scenario"). Substitutability in this case can be verified a-priori, by establishing backward compatibility relation between subsequent component revisions, and benefit from correctly set meta-data indicating compatibility [23]. Among this meta-data, version identifier has the primary role to indicate both the position of the particular revision in the component revision sequence and its backward compatibility. This is achieved by employing suitable semantic versioning schemes [70]; the current state of practice is however to set the data manually.

3.4 Contribution

Our work on ensuring consistency for dynamically evolving component-based systems has its roots in three particular streams of computer science and software engineering research: (1) Type safety [29], signature matching [103] and behavioural subtyping [55] which concentrate on the formal models for representing and comparing software elements at the lower levels of abstraction and finer granularity. (2) The part of the software architecture research dealing with verification of architecture consistency, e.g. in the UniCon ADL [80], which emphasize the issue at the coarser-grained level of software system architectures. (3) Versioning approaches that strive to provide formal backing for meta-data based compatibility of software modules, like the scheme used by the Distributed Computing Environment (DCE [74]) or the Gandalf system [45].

Horizontal Compatibility Unlike fully formalized but restricted methods of compatibility checking, we have striven for methods that can be used with readily available specifications/implementations of real-world component models on a wide scale. At the syntactic level, paper B1 [22] defines the subtype relation on component types in terms of specification differences. Compatibility verification can use these subtype checks on the corresponding parts of the component type representation.

These *type-based consistency checks* make sure that syntactic level consistency is verified when components are being assembled into an architecture. Consistency at this level is obviously checked by the compiler for monolithic applications or by specialized composition tools for static architectures [80], however, it can be easily broken in systems which employ ad-hoc composition, as we show in paper B1 [22]. This work provides an example that ensuring even just syntactic compatibility is a challenge especially for industrial systems with dynamically evolving architectures.

[4]We should note that this is considered a really special case in the area of component systems [92] which are meant to allow full general substitutability, even though Wallnau argues [100] that full-featured substitutability might never happen for software components for reasons related to market forces.

Our method was implemented in a tool [18, 19] to determine compatibility on inter-component bindings in the OSGi framework.

Based on our platform-independent EFP specification scheme, presented in Section 2.5, we have proposed a method to ensure inter-component contractual *consistency at the EFP level*. Paper B5 [50] introduces the algorithms and methods for checking that components in an assembly are consistent in terms of extra-functional properties (see also [49]), as a next stage after their syntactic compatibility has been determined.

Component Substitutability We apply the type-based consistency checks described in paper B1 [22] to verify subtype relation on the current and replacement components at manageable algorithmic complexity. This ensures so called *strong substitutability* applicable for run time component replacement. The method has been validated on an industrial component framework as described in paper B2 [18]; the work also uncovered the difficulties in integrating consistency checks with the component update process. The method for verifying compatibility of extra-functional properties, discussed above and described in paper B5 [50], can augment the type-based checks within the same update process scheme.

We introduced the novel notion of *contextual substitutability* which uses the contextual complement of current component during substitutability evaluation. Formally defined in paper B4 [21], this method evaluates the actual usage of the current component's features as well as the capabilities of its enclosing architecture against the replacement component. The consequence is that the current and replacement components need not be in subtype relation yet be safely substitutable, enabling consistency preserving architectural evolution not restricted to strict substitutability on updates.

Lastly, based on the general type-based consistency checking approach we designed a method for automated creation of semantically correct component version identifiers, described in paper B3 [8]. Such identifiers are used to simplify the update process, as follows. Assume this meta-data is correct, i.e. given a sequence of component revisions $(C_1..C_n)$, it holds for their version identifiers $v(C_i)$ that $\forall C_i, C_j \in (C_1..C_n)$: $v(C_i) \leq v(C_j)$ iff $i \leq j \wedge C_j \preceq C_i$ under a suitable ordering relation on v. Then we can safely verify compatibility during the update process only by comparing the identifiers, checking $v(C_C) \leq v(C_R)$, and similarly for horizontal compatibility in case of versioned features. Rather than running the verification methods in full with $O(n)$ computational complexity for simple EFPs or $O(e^n)$ for checking behavioural refinement, we thus achieve constant time checks. In an earlier work [23] we also proposed a more complex meta-data model to further enhance the consistency checks on update.

3.5 A Note on the Results Applicability

Formal or at least well-founded methods of substitutability verification are rare in industrial systems. Hard real-time and high availability systems use model checking to some degree (reasonable from development time point of view), while mainstream platforms usually support only backward compatibility checks based on manually created version data (which we might consider rather primitive). The latter approach has been used in such high-profile systems like DCE [74], Solaris libraries [26], Microsoft's .NET framework, Linux package distributions and OSGi [71, 70].

Since we have shown the fragility of this approach [19, 9], there is clearly a room for such substitutability verification methods that are both formally well-founded and relevant for real world systems. We believe the methods contributed and presented in this thesis are fully applicable in the wider context of practical solutions.

4 Collection of Papers

Collection "A": Software Components: Models and Properties

A1 [17] – Brada, P. The CoSi component model: Reviving the black-box nature of components. In *Proceedings of the 11th International Symposium on Component Based Software Engineering (CBSE)*, volume 5282 of Lecture Notes in Computer Science, October 2008. Springer Verlag. [17]

A2 [20] – Brada, P. A look at current component models from the black-box perspective. In *Proceedings of 35th Euromicro Conference on Software Engineering and Advanced Applications*, Patras, Greece, 2009. IEEE Computer Society Press.

A3 [48] – Jezek, K., Brada, P. and Stepan, P. Towards context independent extra-functional properties descriptor for components. *Electronic Notes in Theoretical Computer Science*, 264(1):55 – 71, 2010.

A4 [9] – Bauml, J. and Brada, P. Reconstruction of type information from Java bytecode for component compatibility. *Electronic Notes in Theoretical Computer Science*, 264(4):3 – 18, 2011.

Collection "B": Consistency of Dynamically Evolving Component Architectures

B1 [22] – Brada, P. and Valenta, L. Practical verification of component substitutability using subtype relation. In *Proceedings of the 32nd Euromicro Conference on Software Engineering and Advanced Applications*, pages 38–45. IEEE Computer Society Press, 2006.

B2 [18] – Brada, P. Enhanced OSGi bundle updates to prevent runtime exceptions. In *Proceedings of the 34th Euromicro Conference on Software Engineering and Advanced Applications*, Parma, Italy, September 2008. IEEE Computer Society Press.

B3 [8] – Bauml, J. and Brada, P. Automated versioning in OSGi: A mechanism for component software consistency guarantee. In *Proceedings of the 34th Euromicro Conference on Software Engineering and Advanced Applications*, Patras, Greece, 2009. IEEE Computer Society Press.

B4 [21] – Brada, P. Enhanced type-based component compatibility using deployment context information. In *Proceedings of Formal Approaches to Software Component Applications (FESCA)*, satellite event of European Conference on Theory and Practice of Software (ETAPS), 2011. Accepted for publication in Electronic Notes on Theoretical Computer Science, Elsevier.

B5 [50] – Ježek, K. and Brada, P. *6th International Conference on Evaluation of Novel Approaches to Software Engineering – Revised Selected Papers*, chapter Formalisation of a Generic Extra-functional Properties Framework. Accepted for publication in Communications in Computer and Information Science (CCIS), ISSN: 1865-0929. Springer-Verlag.

5 Bibliography

[1] UML profile for modeling quality of service and fault tolerance characteristics and mechanisms specification. OMG Specification formal/2008-04-05, Object Management Group, April 2008.

[2] Martín Abadi and Leslie Lamport. Composing specifications. *ACM Trans. Program. Lang. Syst.*, 15:73–132, January 1993.

[3] Gregory D. Abowd, Robert Allen, and David Garlan. Formalizing style to understand descriptions of software architecture. *ACM Trans. Softw. Eng. Methodol.*, 4:319–364, October 1995.

[4] Novia Admodisastro and Gerald Kotonya. An architecture analysis approach for supporting black-box software development. In *Proceedings of the 2011 European Conference on Software Architecture (ECSA)*, volume 6903/2011 of *Lecture Notes in Computer Science*, pages 180–189. Springer Verlag, 2011.

[5] J. Aldrich, C. Chambers, and D. Notkin. Archjava: Connecting software architecture to implementation. In *Proceedings of the ICSE'02*. ACM Press, Orlando, Florida, May 2002.

[6] Alexandre Alvaro, Eduardo Santana de Almeida, and Silvio Romero de Lemos Meira. A software component quality framework. *SIGSOFT Software Engineering Notes*, 35:1–18, January 2010.

[7] Felix Bachmann et al. Volume II: Technical concepts of component-based software engineering. Technical Report CMU/SEI-2000-TR-008, Software Engineering Institute, Carnegie Mellon University, 2000.

[8] Jaroslav Bauml and Premek Brada. Automated versioning in OSGi: a mechanism for component software consistency guarantee. In *Proceedings of the 35th Euromicro Conference on Software Engineering and Advanced Applications*, pages 428–435, Patras, Greece, 2009. IEEE Computer Society Press.

[9] Jaroslav Bauml and Premek Brada. Reconstruction of type information from Java bytecode for component compatibility. *Electronic Notes in Theoretical Computer Science*, 264(4):3 – 18, 2011.

[10] Steffen Becker, Heiko Koziolek, and Ralf Reussner. The palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1):3 – 22, 2009. Special Issue: Software Performance - Modeling and Analysis.

[11] Meriem Belguidoum and Fabien Dagnat. Formalization of component substitutability. *Electronic Notes on Theoretical Computer Science*, 215:75–92, June 2008.

[12] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making components contract aware. *Computer*, 32(7):38–45, 1999.

[13] Michel Bidoit and Rolf Hennicker. An algebraic semantics for contract-based software components. In *Proceedings of the 12th International Conference on Algebraic Methodology and Software Technology (AMAST)*, volume 5140/2008 of *Lecture Notes in Computer Science*, pages 216–231. Springer Verlag, 2008.

[14] Gavin Bierman, Matthew Parkinson, and James Noble. UpgradeJ: Incremental typechecking for class upgrades. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 5142 of *Lecture Notes on Computer Science*, pages 235–259. Springer Verlag, 2008.

[15] Gordon Blair, Thierry Coupaye, and Jean-Bernard Stefani. Component-based architecture: the fractal initiative. *Annals of Telecommunications*, 64(1-2):1–4, February 2009.

[16] Jan Bosch. Architecture in the age of compositionality. In *Proceedings of the European Conference on Software Architecture*, volume 6285 of *Lecture Notes in Computer Science*, 2010.

[17] Premek Brada. The CoSi component model: Reviving the black-box nature of components. In *Proceedings of the 11th International Symposium on Component Based Software Engineering (CBSE)*, volume 5282 of *Lecture Notes in Computer Science*, Karlsruhe, Germany, October 2008. Springer Verlag.

[18] Premek Brada. Enhanced OSGi bundle updates to prevent runtime exceptions. In *Proceedings of the 34th Euromicro Conference on Software Engineering and Advanced Applications*, pages 92–99, Parma, Italy, September 2008. IEEE Computer Society Press.

[19] Premek Brada. Bundle updates with verified compatibility. Presentation on OSGi DevCon Europe, June 2009. Zurich, Switzerland.

[20] Premek Brada. A look at current component models from the black-box perspective. In *Proceedings of 35th Euromicro Conference on Software Engineering and Advanced Applications*, pages 388–395, Patras, Greece, 2009. IEEE Computer Society Press.

[21] Premek Brada. Enhanced type-based component compatibility using deployment context information. In *Proceedings of Formal Approaches to Software Component Applications (FESCA), satellite event of European Conference on Theory and Practice of Software (ETAPS)*, 2011. Accepted for publication in Electronic Notes on Theoretical Computer Science, Elsevier.

[22] Premysl Brada and Lukas Valenta. Practical verification of component substitutability using subtype relation. In *Proceedings of the 32nd Euromicro Conference on Software Engineering and Advanced Applications*, pages 38–45. IEEE Computer Society Press, 2006.

[23] Přemysl Brada. Metadata support for safe component upgrades. In *Proceedings of COMPSAC'02, the 26th Annual International Computer Software and Applications Conference*, Oxford, England, August 2002. IEEE Computer Society Press.

[24] Jeremy S. Bradbury, James R. Cordy, Juergen Dingel, and Michel Wermelinger. A survey of self-management in dynamic software architecture specifications. In *WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, pages 28–33, New York, NY, USA, 2004. ACM.

[25] Alan W. Brown and Curt Wallnau. The current state of CBSE. *IEEE Software*, 15(5):37–46, September/October 1998.

[26] David J. Brown and Karl Runge. Library interface versioning in Solaris and Linux. In *Proceedings of the 4th Annual Linux Showcase and Conference*, Atlanta, Georgia, 2000. USENIX.

[27] Manfred Broy, Anton Deimel, Juergen Henn, Kai Koskimies, Frantisek Plasil, Gustav Pomberger, Wolfgang Pree, Michael Stal, and Clemens A. Szyperski. What characterizes a (software) component? *Software - Concepts and Tools*, 19(1):49–56, 1998.

[28] Tomas Bures, Petr Hnetynka, and Frantisek Plasil. SOFA 2.0: Balancing advanced features in a hierarchical component model. In *Proceedings of SERA2006*, pages 40–48, Seattle, USA, Aug 2006. IEEE CS.

[29] Luca Cardelli. Type systems. In *Handbook of Computer Science and Engineering*, chapter 103. CRC Press, 1997.

[30] Sagar Chaki, Edmund Clarke, Natasha Sharygina, and Nishant Sinha. Verification of evolving software via component substitutability analysis. *Formal Methods in System Design*, 32(3), June 2008.

[31] Lawrence Chung and Julio do Prado Leite. On non-functional requirements in software engineering. In Alexander Borgida, Vinay Chaudhri, Paolo Giorgini, and Eric Yu, editors, *Conceptual Modeling: Foundations and Applications*, volume 5600 of *Lecture Notes in Computer Science*, pages 363–379. Springer Berlin / Heidelberg, 2009.

[32] Edmund Clarke. Model checking. In S. Ramesh and G Sivakumar, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 1346 of *Lecture Notes in Computer Science*, pages 54–56. Springer Berlin / Heidelberg, 1997. 10.1007/BFb0058022.

[33] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Progress on the state explosion problem in model checking. In Reinhard Wilhelm, editor, *Informatics*, volume 2000/2001 of *Lecture Notes in Computer Science*, pages 176–194. Springer Berlin / Heidelberg, 2001.

[34] Ivica Crnkovic and Magnus Larsson. *Building Reliable Component-Based Software Systems*. Artech House, 2002. ISBN 978-1580533270.

[35] Ivica Crnković, Séverine Sentilles, Aneta Vulgarakis, and Michel R.V. Chaudron. A classification framework for software component models. *IEEE Transactions on Software Engineering*, 37(5):593–615, September/October 2011.

[36] William P. de Roever. The quest for compositionality. In *Proceedings of IFIP Working Conference on the Role of Abstract Models in Computer Science*. Elsevier Science B.V., 1985.

[37] Nicolas Desnos, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, and Guy Tremblay. Automated and unanticipated flexible component substitution. In *Proceedings of 10th International Symposium on Component-Based Software Engineering*, volume 4608/2007 of *Lecture Notes in Computer Science*, Medford, MA, USA, July 2007. Springer Verlag.

[38] Tudor Dumitras and Priya Narasimhan. Why do upgrades fail and what can we do about it? In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*. Springer Verlag, 2009.

[39] Jacky Estublier (Editor) et al. Impact of the research community on the field of software configuration management: Summary of an impact project report. *Software Engineering Notes*, 27(5), September 2002.

[40] Andres Flores and Macario Polo. Testing-based process for evaluating component replaceability. In *Proceedings of the 3rd International Workshop on Views On Designing Complex Architectures (VODCA 2008)*, pages 101 – 115, 2009. Electronic Notes in Theoretical Computer Science, vol. 236.

[41] Xavier Franch. Systematic formulation of non-functional characteristics of software. In *Third International Conference on Requirements Engineering (ICRE'98)*, volume 00, page 0174, Los Alamitos, CA, USA, 1998. IEEE Computer Society.

[42] Kiev Gama and Didier Donsez. Towards dynamic component isolation in a service oriented platform. In *Proceedings of Component-Based Software Engineering*, volume 5582/2009 of *Lecture Notes in Computer Science*. Springer, 2009.

[43] Stéphanie Gatti, Emilie Balland, and Charles Consel. A step-wise approach for integrating qos throughout software development. In Dimitra Giannakopoulou and Fernando Orejas, editors, *Fundamental Approaches to Software Engineering*, volume 6603 of *Lecture Notes in Computer Science*, pages 217–231. Springer, 2011.

[44] Robert B. Grady. *Practical Software Metrics for Project Management and Process Improvement*. Prentice Hall, 1992.

[45] Nico Habermann and David Notkin. Gandalf: Software development environments. *IEEE Transactions on Software Engineering*, 12(12), December 1986.

[46] P. Hnětynka and F. Plášil. Dynamic reconfiguration and access to services in hierarchical component models. In *Proceedings of CBSE 2006*, volume Lecture Notes in Computer Science, pages 352–359, Vasteras, Sweden, 2006. Springer.

[47] Michael N. Huhns and Munindar P. Singh. Service-oriented computing: Key concepts and principles. *IEEE Internet Computing*, 9(1):75–81, 2005.

[48] Kamil Jezek, Premek Brada, and Petr Stepan. Towards context independent extra-functional properties descriptor for components. *Electronic Notes in Theoretical Computer Science*, 264(1):55 – 71, 2010.

[49] Kamil Ježek and Premek Brada. Compatibility verification of components in terms of functional and extra-functional properties - tool support. In *Proceedings of the 12th International Conference on Enterprise Information Systems - Information Systems Analysis and Specification*, pages 510–514. SciTePress, 2010.

[50] Kamil Ježek and Premek Brada. *6th International Conference on Evaluation of Novel Approaches to Software Engineering – Revised Selected Papers*, chapter Formalisation of a Generic Extra-functional Properties Framework. Communications in Computer and Information Science (CCIS), ISSN: 1865-0929. Springer-Verlag, 2012. Accepted for publication.

[51] Antonio Ruiz-Cortés Octavio Martín-Díaz José María García, David Ruiz and Manuel Resinas. An hybrid, QoS-aware discovery of semantic web services using constraint programming. In *Proceedings of the 2007 International Conference on Service-Oriented Computing (ICSOC)*, volume 4749/2007 of *Lecture Notes in Computer Science*, pages 69–80. Springer Verlag, 2007.

[52] Heiko Koziolek. Performance evaluation of component-based software systems: A survey. *Performance Evaluation*, 67(8):634–658, August 2010.

[53] Philippe Kruchten. *The Rational Unified Process: An Introduction.* Addison-Wesley Professional, 3rd edition edition, 2003.

[54] Barbara Liskov. Keynote address – data abstraction and hierarchy. *SIGPLAN Not.*, 23:17–34, January 1987.

[55] Barbara Liskov and Jeanette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[56] J. Magee et al. Specifying distributed software architectures. In *Proceedings of ESEC'95*, Barcelona, Spain, 1995.

[57] Jeff Magee and Jeff Kramer. Dynamic structure in software architectures. *SIGSOFT Softw. Eng. Notes*, 21(6):3–14, 1996.

[58] Leonardo Mariani, Sofia Papagiannakis, and Mauro Pezze. Compatibility and regression testing of cots-component-based software. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 85–95, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[59] S. McCamant and M. Ernst. Predicting problems caused by component upgrades. In *Proceedings of the 10th European Software Engineering Conference and the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 287–296, Helsinki, Finland, 2003.

[60] Doug McIlroy. Mass-produced software components. In P. Naur and B. Randell, editors, *Proceedings of NATO Software Engineering Conference*, pages 138–155, Garmisch, Germany, October 1968.

[61] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, January 2000.

[62] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall International, 2nd edition edition, 2000.

[63] Microsoft Corporation and Digital Equipment Corporation. *The Component Object Model Specification*, October 1995.

[64] J. Misra and K.M. Chandy. Proofs of networks of processes. *Software Engineering, IEEE Transactions on*, SE-7(4):417 – 426, july 1981.

[65] M. Mohammad and V. Alagar. Tadl - an architecture description language for trustworthy component-based systems. In *Proceedings of Software Architecture: Second European Conference (ECSA 2008)*, page 290, Paphos, Cyprus, September-October 2008. Springer-Verlag New York Inc.

[66] Brice Morin, Olivier Barais, Grégory Nain, and Jean-Marc Jézéquel. Taming dynamically adaptive systems using models and aspects. In *Proceedings of the 31st IEEE International Conference on Software Engineering.* IEEE Computer Society Press, May 2009.

[67] Johan Muskens, Michel R.V. Chaudron, and Johan J. Lukkien. *Component-Based Software Development for Embedded Systems*, chapter A Component Framework for Consumer Electronics Middleware, pages 164–184. Springer Verlag, 2005.

[68] Object Management Group. *CORBA Component Model Specification, Version 4.0*, April 2006. OMG Specification formal/06-04-01.

[69] The OSGi Alliance. *OSGi Service Platform, Release 3*, March 2003. Available at http://www.osgi.org/.

[70] The OSGi Alliance. *Semantic Versioning: Technical Whitepaper*, revision 1.0 edition, May 2010.

[71] The OSGi Alliance. *OSGi Service Platform Core Specification*, June 2011. Release 4, Version 4.3.

[72] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, December 1972.

[73] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[74] M. Peterson. *DCE: A Guide to Developing Portable Applications*, chapter 17: UUID and Version attributes. McGraw-Hill, 1995.

[75] František Plášil, Dušan Bálek, and Radovan Janeček. SOFA/DCUP: architecture for component trading and dynamic updating. In *Proceedings of ICCDS'98*, Annapolis, Maryland, USA, 1998. IEEE CS Press.

[76] František Plášil and Stano Višnovský. Behavior protocols for software components. *IEEE Transactions on Software Engineering*, 28(10), November 2002.

[77] S. Röttger and S. Zschaler. CQML+: Enhancements to CQML. In J.-M. Bruel, editor, *1st Intl. Workshop on Quality of Service in Component-Based Software Engineering*, pages 43–56. Toulouse, France, June 2003.

[78] Séverine Sentilles, Aneta Vulgarakis, Tomáš Bureš, Jan Carlson, and Ivica Crnković. A component model for control-intensive distributed embedded systems. In Michel R. V. Chaudron, Clemens Szyperski, and Ralf Reussner, editors, *Proceedings of 11th International Symposium on Component-Based Software Engineering*, number 5282 in Lecture Notes in Computer Science, Karlsruhe, Germany, 2008. Springer-Verlag.

[79] Arun Sharma, P.S.Grover, and Rajesh Kumar. Reusability assessment for software components. *ACM SIGSOFT Software Engineering Notes*, 34(2), March 2009.

[80] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21(4):314–335, April 1995.

[81] Mary Shaw and David Garlan. *Software architecture. Perspectives on an emerging discipline.* Prentice Hall Publishing, 1996.

[82] Navonil Mustafee-Sergio de Cesare-Mark Lycett Simon J.E. Taylor, David Bell and Paul A. Fishwick. Semantic web services for simulation component reuse and interoperability: An ontology approach. In Angappa Gunasekaran and Timothy Shea, editors, *Organizational Advancements through Enterprise Information Systems: Emerging Applications and Developments*, chapter 21. IGI Global, 2010.

[83] Eugene Stark. A proof technique for rely/guarantee properties. In S. Maheshwari, editor, *Foundations of Software Technology and Theoretical Computer Science*, volume 206 of *Lecture Notes in Computer Science*, pages 369–391. Springer Berlin / Heidelberg, 1985.

[84] Hong Qing Yu Stephan Reiff-Marganiec and Marcel Tilly. Service selection based on non-functional properties. In *Proceedings of the International Conference on Service-Oriented Computing (ICSOC) 2007 Workshops*, volume 4907/2009 of *Lecture Notes in Computer Science.* Springer Verlag, 2009.

[85] Alexander Stuckenholz. Component evolution and versioning state of the art. *SIGSOFT Softw. Eng. Notes*, 30(1):7, 2005.

[86] Alexander Stuckenholz. Component updates as a boolean optimization problem. *Electronic Notes on Theoretical Computer Science*, 182:187–200, 2007. Proceedings of the Third International Workshop on Formal Aspects of Component Software (FACS'06).

[87] Alexander Stuckenholz and Olaf Zwintzscher. Compatible component upgrades through smart component swapping. In *Architecting Systems with Trustworthy Components*, volume 3938 of *Lecture Notes in Computer Science*, pages 216–226. Springer-Verlag, January 2006.

[88] Sun Microsystems, Inc. *Enterprise JavaBeans(TM) Specification (Version 2.0)*, August 2001. Available at http://java.sun.com/products/ejb/docs.html.

[89] Daniel Sykes, William Heaven, Jeff Magee, and Jeff Kramer. Exploiting non-functional preferences in architectural adaptation for self-managed systems. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, 2010.

[90] Clemens Szyperski. Independently extensible systems – software engineering potential and challenges. In *Proceedings of the 19th Australian Computer Science Conference*, Melbourne, Australia, 1996.

[91] Clemens Szyperski. *Component Software, Second Edition.* ACM Press, Addison-Wesley, 2002.

[92] Clemens Szyperski. Component technology - what, where, and how? In *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003)*, Portland, Oregon, May 2003.

[93] Richard N. Taylor et al. A component- and message-based architectural style for GUI software. *Software Engineering*, 22(6):390–406, 1996.

[94] Richard N. Taylor, Nenad Medvidović, and Eric M. Dashofy. *Software Architecture – Foundations, Theory, and Practice.* Wiley, 2010.

[95] Antonio Vallecillo, Juan Hernández, and José M. Troya. Object interoperability. In *Object-Oriented Technology ECOOP'99 Workshop Reader*, Lecture Notes in Computer Science. Springer Verlag, 1999.

[96] Antonio Vallecillo, Juan Hernández, and José M. Troya. Component interoperability. Technical Report ITI-2000-37, Universidad de Málaga, Spain, July 2000.

[97] Rob van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The koala component model for consumer electronics software. *IEEE Computer*, 33(3):78–85, 2000.

[98] Radu Vanciu and Václav Rajlich. Hidden dependencies in software systems. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance (ICSM).* IEEE Computer Society Press, September 2010.

[99] Thomas Vogel and Holger Giese. Adaptation and abstract runtime models. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. ACM New York, USA, 2010.

[100] Kurt C. Wallnau. On software components and commercial ("COTS") software. In *Proceedings of the 2nd Workshop on Component-Based Software Engineering, in conjunction with ICSE'99*, May 1999.

[101] Peter Wegner and Stanley B. Zdonik. Inheritance as an incremental modification mechanism or what like is and isn't like. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 322, pages 55–77. Springer-Verlag, 1988.

[102] Amy Moormann Zaremski and Jeanette Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4), October 1997.

[103] Amy Moormann Zaremski and Jeannette M. Wing. Signature matching: a tool for using software libraries. *ACM Trans. Softw. Eng. Methodol.*, 4(2):146–170, 1995.

[104] Matthias Zenger. Type-safe prototype-based component evolution. In *Proceedings of the European Conference on Object-Oriented Programming*, Malaga, Spain, June 2002.

[105] Pengcheng Zhang, Henry Muccini, and Bixin Li. A classification and comparison of model checking software architecture techniques. *Journal of Systems and Software*, 83(5):723 – 744, 2010.

[106] Jingang Zhou, Dazhe Zhao, Yong Ji, and Jiren Liu. Examining OSGi from an ideal enterprise software component model. In *Proceedings of the 2010 IEEE International Conference on Software Engineering and Service Sciences (ICSESS)*, pages 119–123. IEEE Computer Society Press, 2010.

[107] Steffen Zschaler. Formal specification of non-functional properties of component-based software systems: A semantic framework and some applications thereof. *Software and Systems Modeling*, 9(2):161–201, 2010.

[108] Jaroslav Šnajberk and Premek Brada. ENT: A generic meta-model for the description of component-based applications. In *Proceedings of the 8th International Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA).*, 2011. Accepted for publication in Electronic Notes on Theoretical Computer Science, Elsevier.