FORSCHUNGSVEREINIGUNG AUTOMOBILTECHNIK E.V.

FAT-SCHRIFTENREIHE

FAT 247

Zuverlässige Wiederverwendung und abgesicherte Integration von Softwarekomponenten im Automobil

VDA Verband der Automobilindustrie

Dependable reuse & guarded integration of automotive software components

Approaches to conditional specification & assurance

Research center:

RWTH Embedded software laboratory

Author:

Marc Förster

Das Forschungsprojekt wurde mit Mitteln der Forschungsvereinigung Automobiltechnik e. V. (FAT) gefördert.

Abstract

The assembly of software components to a properly functioning system is a key problem in the development and maintenance of software for safety-critical, real-time systems. The integration of binary components may take place during integration testing or at run time (updates, patches, adaptation, ...). It would be even more advantageous to be able to integrate components “virtually” already during the design phase, to analyze their interaction early and possibly fix any specification errors before implementation.

The concept of virtual integration requires modular and semantic component descriptions that go beyond the usual specification of syntactic interfaces. The core idea is to split descriptions into requirements of the component on environment inputs on one side, and assertions about the properties of the component’s output on the other, provided the requirements are fulfilled.

There exist several approaches to the realisation of this concept, called assume-guarantee/rely-guarantee/assumption-commitment reasoning, Design by contract, Rich components etc (subsumed in the present report as “conditional specification” or “conditional assurance”, respectively). Mostly they are basic research and have not been consistently applied so far in the context of automotive software or embedded systems development.

The objectives of this project are

- a survey of previous work in the area of conditional assurance,
- the classification of identified approaches in a practical schema,
- the exemplary application of a selected approach to a concrete system model and scenario.

Keywords. Automotive software, case study, compositional reasoning, conditional specification, contract algebra, embedded systems, model-based development, program verification, software components, software reuse, state of the art, survey, virtual integration.

Table of contents

1 Introduction

The integration of software components towards an operational system that reliably complies with requirements is one of the most important problems in the development and maintenance of automotive embedded software. This multifaceted task arises in various situations of diverse complexity. Integration concerns, for example, components that

- have been developed by different teams from the same organisation,
- have been developed by other organisations (suppliers),
- have been developed by customers (OEM, for example), and that may only be available as object code,
- have already been employed in the field (reuse).

In a broader view, the integration challenge occurs not just during development but also during runtime: with updates and patches of integrated components, during the integration of new components (after-sale upgrade) or the activation/deactivation of components due to energy management or load balancing.

1.1 Promises & challenges of software reuse

Of course, opportunistic reuse [172] and code scavenging [128] has always been employed by programmers. The advent of OO with implementation inheritance during the 1970s put code reuse on a more systematic basis. The drawbacks of this sort of inheritance, however, were soon noted ("Inheritance breaks encapsulation." [77] discuss the problem in depth), which lead to OO variants that separate subtyping from inheritance [13] [167]. At the end of the 1960s, around the same time as the notion of "software components" [145], the idea of modularising program abstractions appeared: the first techniques were for sequential programs [73] [98], followed by monolithic methods for concurrent programs [16] [132] [160], and eventually by compositional approaches [118] [119] [151].

In the context of software, the term “contract” was first used in passing by Lamport [133], Helm & al promoted it to the title of their article “Contracts: specifying behavioral compositions in object-oriented systems” [94], and Meyer took it up with the OO language *Eiffel* [147] [148] [149] [150].

At the end of the 1990s Beugnard & al came up with a call for “contract awareness” of software components, as “a way of determining beforehand whether we can use a given component within a certain context,” [29] aiming at design-phase integration and improved reuse. Since then, many articles and approaches about conditional specification and assurance have been published.

There exists, however, a tradeoff between usability (fitness for a *specific* purpose), along with the corresponding requirements on performance, timeliness, safety etc, and reusability (fitness for serving a purpose *in different environments*). Genericity, for example, increases the chance for component reuse but may have a negative impact on performance, thus in turn reducing the number of scenarios in which the component can be successfully deployed. Indeed, it has become clear that reuse does not work well on the level of binaries or even code [30] [78] [79], instead it should probably be sought on a more abstract model level [40].

1.2 Objectives & approach

Conventionally, development considers closed systems, in that the composition of a system and its environment presupposes a fixed environment, which leads to limited reusability. Accordingly, there is a need for specification and analysis techniques for systems that are “open” (at design time, and perhaps also, but not necessarily, at runtime). The problem is that the environment provided for a reusable component is unknown or only partly known beforehand.

To support reuse, the FAA advisory circular “Reusable software components” [183] requires a number of process-based measures but no (formal methods for) product-oriented measures. More recently, the AUTOSAR initiative [101] tackles reuse, system composition and component exchange by way of SW components that may be deployed

regardless of the underlying hardware structure (ECUs, buses etc). AUTOSAR defines interfaces, middleware (RTE) and communication mechanisms that serve to decouple applications from the unterlying HW and basic SW (system services, network management, bus communication, memory management, device drivers etc). SW component descriptions are used by the “Virtual functional bus” to validate the interaction of all components and interfaces before software implementation. The AUTOSAR virtual integration check for SW integration, however, besides basic requirements of the software to HW resources (latency, scheduling, memory size/type etc), mainly considers syntactical properties such as completeness of SW component descriptions and the integrity, correctness and compatibility of interfaces. The AUTOSAR concept does not regard semantic aspects of SW interaction and integration.

In consequence, AUTOSAR is currently missing the capability to answer the question: „Will this component integrate with other components in such a way as to attain the required system behaviour?“ The ability to answer the question *before* actual system integration would be extremely valuable. For this purpose a component framework is needed which, besides purely syntactic interface constraints (parameter types, data formats etc), also captures semantic and extrafunctional aspects (for example, jitter, precision, availability). Moreover, in the spirit of conditional specification, interface descriptions should discriminate between requirements of the component on its inputs and the output qualities it will guarantee in return.

There exist a number of approaches aiming at the objective described above: assume/guarantee, rely/guarantee, assumption-commitment reasoning, Design by contract, Rich components, verifying compiler etc. At present virtually all of them are essentially research in progress, with the exception of Design by contract which has been realised in the programming language Eiffel [103]. Nevertheless, none of the approaches mentioned has as yet been consistently applied in practice in the area of automotive software or embedded systems.

1.3 Research schedule

This FAT project intends to pave the way for the practical introduction of conditional specification and assurance to automotive software development, improving the methodological support for the integration and reuse of software components. This aim shall be achieved by a comprehensive survey of available approaches, a statement of relevant integration scenarios and the prototypical application of a selected approach in a case study with a realistic system. The potential of the approach shall become apparent.

The project serves to take a first step towards a semantic component framework for the German automobile industry. Its results will enable the FAT members (1) to comprehend the basic idea and potential of the investigated methods, (2) to assess their suitability for the selected integration scenarios, and (3) to get a deeper insight into the approach that is applied in the case study.

The work plan comprises

- a survey of the state of the art regarding pertinent approaches,
- a compilation of relevant integration scenarios,
- the application of a suitable approach to a concrete component architecture/system, and its validation.

The present report constitutes the main result of the research schedule and work plan delineated above.

2 Embedded components & modular reasoning

More than forty years ago, in the wake of the "software crisis", McIlroy's original vision of "mass-produced" software components [145] was inspired by the way standardised hardware parts are acquired and assembled. A (re)use, on this scale, of prefabricated binaries to build new systems has not been realised so far. Compiled software, as well as source code, is often hampered by its high specialisation, architectural mismatch [78] [79] and the difficulty of actually finding adequate candidates for composition. For this reason, De Bruin & van Vliet argue for component generation instead of retrieval [40].

Generating software automatically appears to be very much in line with model-based development (MBD) as it is found in the automobile industry. However, for a (component) model to be usable during the entire, possibly distributed, development cycle it needs to facilitate

- independent implementation (top down), such that separately developed components are ensured to work together as specified, and
- incremental design (bottom up), ensuring that components, as well as their models, can be reused and integrated in any order, with their assembly still satisfying higher-level requirements.

Existing "component" models, for example, CORBA (OO) [107] or Ptolemy (data-flow based, actor oriented) [108] typically are "flat" models without the possibility of defining hierarchical abstraction or refinement and cannot handle aspects of integration, reuse or substitution in an optimal way, thus, their usability in the design phase is limited.

2.1 Components & their specification

What exactly is a component? From the many existing definitions, the following is probably the most popular and best known:

> "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties." [178]

This statement is more general than others, in that it does not explicitly require to include a readily executable implementation. The SEI, for example, regards software components as the union of implementation and architectural abstraction [17]. Nevertheless, the above citation (which in turn is just a citation by Szyperski himself) refers to a component concept relatively close to COTS.

The present report assumes a perspective that is a bit more abstract and follows Lamport in "[making] no distinctions between specifications and implementations. They are all descriptions of a system at various levels of detail." [135] In the context of this report, the term "component" then already denotes the architectural abstraction alone. The "third party" using (enhancing, refining, composing) a component may, of course, simply be the same organisation at another point in time. This notion of contractual components is closer to the original contract concept with abstraction and refinement [94] [100] than to Szyperski's [178] or SEI's [17] more implementation/code/binary-oriented understanding.

2.1.1 Component- & model-based development

Components are self-contained entities meant for composition and, therefore, reuse, conforming to a component model that defines what can be said about their interaction with the environment [17]. There exist two paradigms of practical importance that are component centric: component-based development (CBD) and model-based development (MBD). Historically, CBD came first, assuming a bottom-up perspective of an assembly of preexisting pieces of software to a whole, while MBD is more top down in nature, decomposing top-level requirements to a degree where they can be assigned to (architectural) components which are specified by models that serve as input for code generation.

> "While CBD tries to combine a bottom-up and a top-down approach where the 'runtime thinking is strong,' MBD constitutes a top-down

approach where knowledge about the systems under development is a critical aspect." [181]

In the end, CBD and MBD are complementary in part, but also have important commonalities: the result of the latter are component models yielding software entities with unambiguously (and, wherever possible, completely) specified requirements and obligations, the former needs to rely on models to guide the integration of reusable software entities.

2.1.2 The component specification

Specifications (interfaces) should expose information to the clients that is necessary and sufficient for using it. This has two complementary aspects: (1) information about what the component offers, and (2) information about what the component requires to make good on the offer. Conventional interface concepts only observe the first part. Including the second directly leads to the notion of contracts.

"Necessary and sufficient" implies that interfaces should hide properties that are prone to change. Interestingly, Bachmann & al [17] remark that an interface "can only be silent about properties about which it can speak […]." If an interface says nothing about component latency, for example, clients might nevertheless (inadvertently) depend on the response time of a specific implementation. If the implementation changes, latency may change, too, breaking clients that depended on it. In consequence, the property should be stated as "No latency guaranteed". Specifying all properties that clients may possibly depend on, however, seems unrealistic. Going further, clients have to make sure that they do not depend on any property not explicitly specified for a component they use since it could change without notice. That, in turn, is also not entirely realistic as long as components are lacking even the most basic meaningful, "semantic", information about their conditions of use.

Insisting on component interfaces as exclusively "formal" specifications seems too narrow a perspective [17]. How do you formally define maintainability? Even a formal, and component-local, specification of such a crucial property as reliability is difficult in the general case, because reliability is not a component property. It is not even a system property

and is only meaningful in a global perspective comprising a component or system as well as the environment it operates in.

2.2 Reasoning about component systems

Standard requirements state properties that must be true unconditionally. Contracts (assume/guarantee statements) offer a more structured form with implication semantics: the assertion of a property (of a component) is subject to a condition (on its environment) [137]. While it is always possible to reason about system properties in a monolithic way, the consideration of component exchange, reuse and integration require a compositional/modular style of reasoning.

2.2.1 Assume/guarantee proof strategy

Expression 2.2.1.a shows the abstract rule followed in modular reasoning:

- if component M satisfies a property φ under assumption A about (all desirable variants of) the environment of M,
- and if the actual environment of M (denoted by M') satisfies A,
- then the (parallel) composition of M and M' satisfies φ.

(2.2.1.a)
$$\frac{\begin{array}{lll} \boldsymbol{M} \parallel \boldsymbol{A} & \models & \boldsymbol{\varphi} \\ \boldsymbol{M'} & \models & \boldsymbol{A} \end{array}}{\boldsymbol{M} \parallel \boldsymbol{M'} \quad \models \quad \boldsymbol{\varphi}}$$

M' can, in turn, be decomposed further, recursively applying the same rule. This method of reasoning is inherently circular because M as well as M' might depend on properties provided by the other part of the composition.

> "The assume-guarantee paradigm provides a systematic theory and methodology for ensuring the soundness of the circular style of postulating and discharging assumptions in component-based reasoning." [96]

Thus, circular reasoning is not unconditionally sound but requires some form of resolution of circular causality which is usually done by abstraction of some form. This can be, for example, fix-point iteration, finding an abstraction similar to a loop invariant, or implicit induction based on the

premise that if a property p holds up to step n, another property q will at least hold up to step $n + 1$ [2] [3] [146].

2.2.2 Properties, property theories & reasoning frameworks

This report essentially follows Beugnard & al [28] [29] in their classification of component (respectively contract) properties (Section 3.1). These properties, to become meaningful, require an associated property theory as well as a reasoning framework [186] to be able to derive assembly properties from those of its components. Simple properties, for example value constraints (level 2) such as $x = true$ or $0 \leq y \leq 100$, can be expressed in the property theories of Boolean or constraint logic and reasoned about/verified using a SAT solver; failures may require stochastics (level 4) to model them; execution orderings (level 3) call for automata, traces or temporal logic[1] as a theory, and for validation by model checking (alternatively, theorem proving to avoid problems with state space largeness); timings are reasoned about using scheduling analysis; and so forth.

The behaviour(s) of a system (respectively, its computations) can be seen as the sequences of variable assignments[2] ("traces", "words" etc) it is able to process and produce. The set of computations (the system's "language") can be represented by finite automata processing and producing the same traces, or a superset (abstraction) thereof. Model checking whether the system satisfies a property is then performed by checking if the language of the system automaton $\mathcal{L}(sys)$ is a subset of the language of an automaton $\mathcal{L}(prop)$, representing the property to be checked:

(2.2.2.a) $$\mathcal{L}(sys) \subseteq \mathcal{L}(prop) \leftrightarrow \mathcal{L}(sys) \cap \overline{\mathcal{L}(prop)} = \emptyset$$

From Expression 2.2.2.a it is clear that this style of model checking, like other methods requiring intersection and complementation, can only work

[1] Temporal logic was first proposed by Pnueli for the formal specification of concurrent programs [162].

[2] These may refer to sequences of states (variable values) as well as corresponding events/transitions/actions etc. In many respects the differences between event-based and state-based concepts are subtle or even nonexistent, as is the case with Kripke structures (state based) and Büchi automata (event based) which can be transformed into each other (similarly to Moore and Mealy automata). LTL, for example, is usually transformed to Büchi automata for model checking.

with classes of automata that are closed under these operations: this is exactly the class of deterministic automata [23] [25].

The types of component properties available for reasoning thus depend on the property theory or theories (that can be) used. CTL, for example, can express bisimilarity [37], considering internal actions of a process, while LTL cannot.[3] Other important properties to check are (formal) safety and liveness, and, in general, liveness cannot be verified in a compositional way [135] [186]. Nevertheless, McMillan proposes just such a method [146], based on a circular compositional rule which uses induction over time (see Section 2.2.1). For cases where timing constraints play a role, there are many variants of timed temporal logic that lend themselves to model checking (Bouyer surveys some of them in [35], for example, MTL, MTL+, MITL, TPTL, TCTL). Timed temporal logic makes it possible to express requirements such as $G(failure \rightarrow F_{\leq 5\ sec} reset)$: "Being in any failure state, a reset will occur within five seconds." As always, greater expressiveness increases analysis effort.

Bachmann & al [17] propose to augment Beugnard & al's property classes by considering global quality attributes of a component: portability, adaptability etc. One could even go further and include custom quality descriptors for services, such as "sensor based" vs "wheel based" for the quality of a gear rate signal in a vehicle [4]. In contrast to, for example, a property such as "maintainability", custom descriptors do lend themselves to formal reasoning. In [165], Raclet & al postulate the need to also cater for the locality of alphabets and for dealing with multiple viewpoints in interface theories, arguing that modal specifications [136] offer the necessary flexibility.

[3] Nain & Vardi argue that "branching-time-based notions of process equivalence are *not* reasonable notions of process equivalence, as they distinguish between processes that are not observationally distinguishable." [152] (Italics from original) They also note further limitations of CTL, in particular that it cannot be used for compositional reasoning.

3 Literature survey

Modular specification of and reasoning about software has been a very active field of research for many years, starting in the 1960s and continuing until today. The number of journal articles, conference papers and technical reports published over the past 15 years alone is in the hundreds. This section intends to give a short but representative overview of publications on relevant approaches directed towards practical application[4] and considers them in a thematic grouping.[5]

3.1 Classification criteria

At the end of the 1990s, Beugnard, Jézéquel, Plouzeau & Watkins' position paper "Making components contract aware" [29] triggered a new wave of interest in contract-based specification, this time in the broader context of component-/service-oriented architectures (as opposed to pure functional or OO contracts, such as in Meyer's *Design by contract* [147] [148]). Observing that component reuse is not without problems (referring to, for example, the *Ariane V* incident [116]), the authors call for specifications that "provide parameters against which the component can be validated", in other words, specifications that serve as a contract between (the provider of) a component and its environment (users).

[29] defines a contract taxonomy with classes of increasing negotiability:

1 – Syntactic level. On this level, the indispensable conditions for interoperability are specified: data types of variables and available operations and their signatures. All of these can be (and are) statically checked by compilers. They have a *yes/no* semantics.

[4] Two notable variants of CS/CA have been left out because they are still in their infancy: modular reasoning with separation logic [64] [72] [184], and proof-carrying code/verifying compiler [33] [99] [153] [163].
[5] This author's understanding of some of the presented approaches, the "classics" in particular, has greatly benefited from the surveys by de Roever [171] and Furia [75]. Furia also considers recent theoretical approaches that have been left out here.

2 – Behavioural level. Behavioural specifications are more expressive as they allow to define the actual content of variables, for example, value ranges or the relationship among or between outputs and inputs. Some of these, for example that a variable will never be *null*, can be statically checked.

3 – Synchronisation level. This level is about the temporal ordering of events and interactions and can be used to specify protocols and history-dependent relations as they are expressible, for example, with temporal logic and automata.

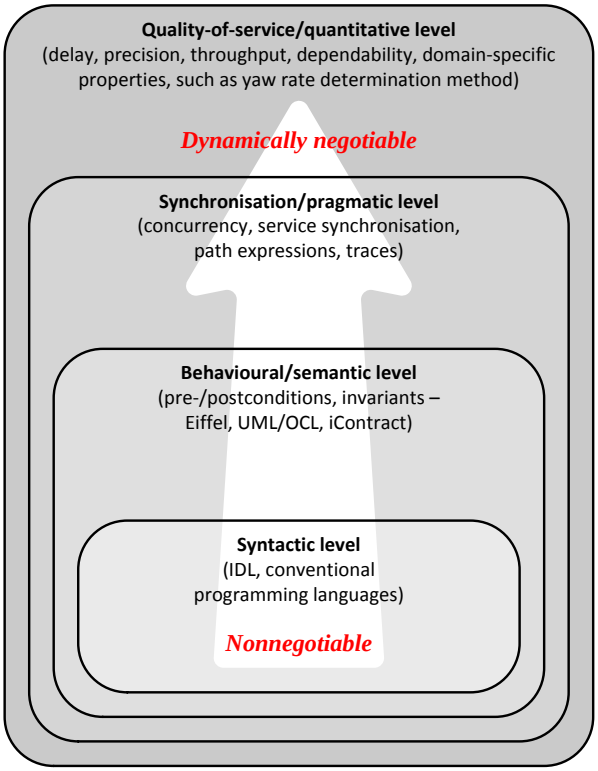

Figure 1. Contract levels (*nature* dimension) [29]

4 – Quality-of-service level. On this level, nonfunctional properties can be specified that not only may be discrete but also continuous: availability, precision, delay etc. Furthermore it is possible to define custom, domain-specific properties, for example, a yaw-rate signal may be associated with a property stating the method of yaw rate determination (sensor-based,

wheel-based, …). This property, in turn, can be used by other components to adapt their behaviour dynamically.[6]

In a recent follow-up, the authors have extended their classification scheme with two additional dimensions: the original negotiability levels are called the "contract nature" dimension, with the additional dimensions *contract location* and *process moment* of the contract [28].

For the purpose of contract classification this report adopts the dimensions *nature* and *process moment*. Within the sections, approaches are ordered by year of publication, from older to newer. With the exception of some early approaches, these are all assume-guarantee/compositional methods for parallel systems, even if not explicitly mentioned.

3.2 Early approaches to conditional assurance

The development from relatively weakly structured proving techniques for sequential programs to fully structured compositional methods for concurrent systems starts with Floyd's flowgraph-based inductive assertion method [73]. This method is not compositional; Figure 2 depicts an example.

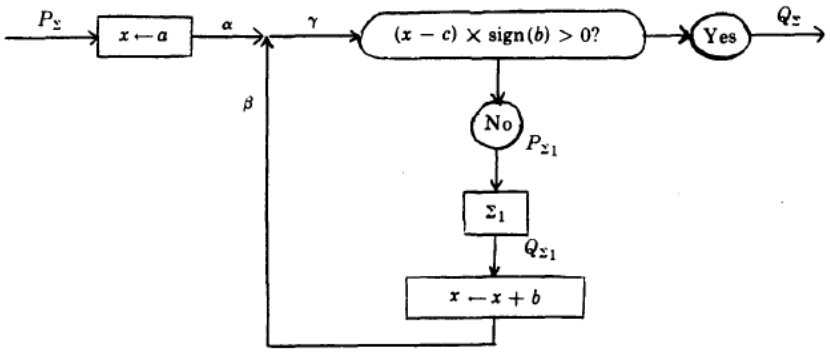

Figure 2. Flowgraph of a sequential program [73]

[6] The contract level hierarchy seems to suggest an increase in verification effort from level one to four. Interestingly, the verification of level-four contracts is not necessarily more difficult than that of level-three contracts. To the contrary, checking event ordering (level three) is probably the most difficult verification task, along with probabilistic aspects (which reside on level four).

Hoare augmented Floyd's approach by providing an algebraic structure that made it compositional. Formula 3.2.a shows the corresponding proof rule.

(3.2.a)
$$\frac{\begin{array}{ccc}\{\boldsymbol{P}\} & \boldsymbol{Q_1} & \{\boldsymbol{R}\} \\ \{\boldsymbol{R}\} & \boldsymbol{Q_2} & \{\boldsymbol{S}\}\end{array}}{\{\boldsymbol{P}\} \quad \boldsymbol{Q_1; Q_2} \quad \{\boldsymbol{S}\}}$$

A single statement ("$\{\boldsymbol{X}\}\ \boldsymbol{Y}\ \{\boldsymbol{Z}\}$") of this rule is today commonly called a "Hoare triple", and the rule asserts that

- if a property P holds before a computation Q_1, and if a property R then holds thereafter, and
- if R holds before a computation Q_2, and if a property S then holds thereafter,
- then the sequential composition of Q_1 and Q_2 will result in a computation before which P holds and after which S holds,
- provided that the computations do indeed terminate (meaning that they possess the *liveness* property).

During the mid-1970s, Ashcroft [16], Owicki & Gries [160] and Lamport [132] proposed, in close succession, monolithic/noncompositional approaches to the formal verification of *concurrent* programs. All of them apply to closed systems only, considering a fixed environment as a component that is already composed with the remaining components.

Jones, at the beginning of the 1980s, was the first to develop a (compositional) method for shared-variable concurrency that could handle open systems with no fixed environment [118] [119]. In naming it, Jones coined the term "rely-guarantee". At about the same time, Misra & Chandy published their compositional approach for concurrent programs whose components communicate via message passing [151]. Eventually, in 1990, Helm & al introduced the denomination of "contract" for the formal basis of this type of reasoning [94].

3.3 Contract-aware components

This section describes "stand-alone" approaches that focus on runtime support for contracts. They may be seen as directly, albeit with a few years delay, heeding to Beugnard & al's call for contract awareness (Section 3.1).

3.3.1 Extra-functional contract support

Jézéquel, Defour & Plouzeau [58] [117] acknowledge that "contractually specified [component] interfaces should go […] beyond mere syntactic aspects […]." They propose a dedicated contract[7] model (Quality-of-service constraint language, QoSCL), extending the quality-of-service modelling language (QML) by Frølund & Koistinen [74]. QML and similar languages aimed at the specification of nonfunctional properties lack the possibility to capture interrelationships and dependences between required and provided qualities. For example, one can specify that a component must return a result within a certain timespan, but the prerequisites for that (such as response times of other services the component depends on) are left out of the picture.

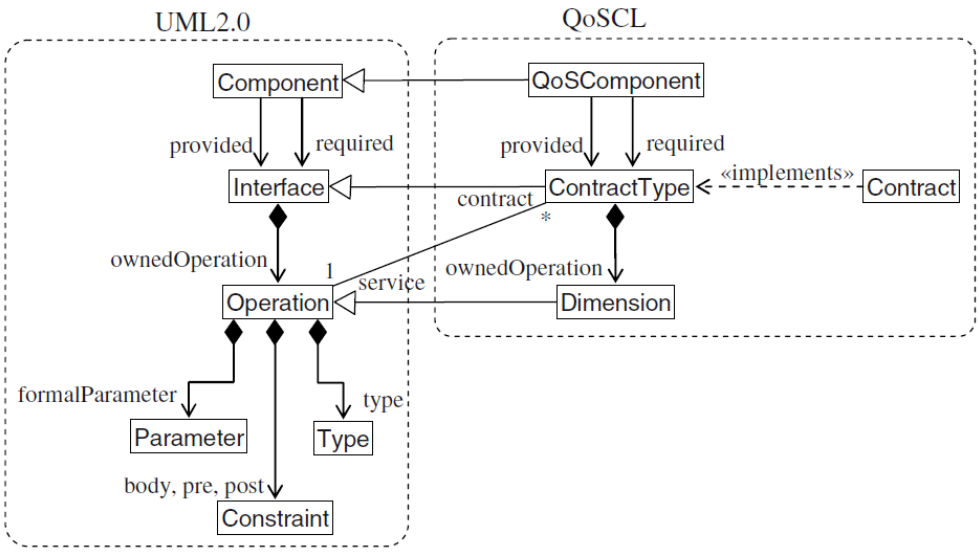

Figure 3. QoSCL metamodel [117]

[7] Note that the authors use "contract" in a way equivalent to the more common term *assertion*. The dependences they propose to include in QoSCL correspond to the usual contract concept.

QoSCL contracts are intended to be used in two ways: (1) as a basis for generating a runtime monitor, with aspect-oriented programming, for a single component; (2) at design time, to analyse a component assembly by converting contracts to statements in a language for constraint logic programming (CLP). For design-time validation, constraints (expressed in OCL) are transformed to a CLP language, yielding a system of linear and possibly nonlinear constraints that is checked for a solution with a dedicated solver.

The authors illustrate their approach with a simple model of a GPS software component and an associated timeout contract. Analysis of timing and precision requirements as well as response times of subcomponents with a CLP solver results in a global contract over input timings, output timings and location precision whose fulfilment is assured.

Application examples. GPS component.

Contract classification. Nature: level 1-2, level 4 (timing); process moment: design time, runtime (monitoring).

3.3.2 SoftContract

The SoftContract approach by Brunel & al [41] integrates the use of contracts into a design method for distributed development. The formulation of requirements as contracts, and their composition/decomposition in an assume/guarantee style, is envisioned by the authors to enable (1) faster design cycles by clear communication of requirements between involved parties, and (2) quality improvement by formal verification of assumptions and guarantees from the design phase up to and including runtime. The authors propose to use monitors that check compliance of system behaviour to the contracts, both in a bottom-up and in a top-down (distributed-development) design flow, and that may trigger recovery or logging actions in case of contract violations.

Assertions are formulated in a trace-based logic called “Logic of constraints” (LOC) [48] that enables reasoning about execution sequences. This formalism is essentially a decidable subset of real-time logic (RTL) [114] [115]. While RTL is strictly more expressive than LTL, LOC just

overlaps it: there are LOC properties that cannot be formulated in LTL, for example, the index-wise comparison of pieces of data from different indexed streams; on the other hand, LOC can only express safety properties, and thus contains no equivalent for LTL liveness properties such as $GF(\varphi)$, "Globally eventually φ", meaning that, starting in an arbitrary state, the system will definitely reach a state satisfying φ.

LOC assertions annotate signals/ports with event indexing and (discrete) timing or value constraints, for example,

```
10 <= Out[i].t – In[i].t <= 20
```

("Output event i is generated between ten and 20 time units after the corresponding input event.") or

```
|In[i-1].v – In[i].v| <= 0.5
```

("The absolute difference of the value of an input event and the value of the preceding input event does not exceed 0.5.").

These assertions can be converted to (1) offline query code that checks their satisfaction on stored simulation traces, (2) online monitor software that indicates assertion violations during simulations, (3) online code that is integrated with software tasks to supervise the runtime system, (4) off- and online hardware-assisted checkers. As an application example the authors report the offline checking of the simulation of a simple adaptive cruise control system regarding a safety property.

Application examples. Adaptive cruise control.

Contract classification. Nature: level 1-2, level 4 (timing); process moment: design time, runtime (monitoring).

3.3.3 The CoConES approach

CoConES by Berbers, Rigole & Vandewoude [27] is a method for the composition of applications from components, offering design-time and runtime support by a dedicated CASE tool (*CCOM*) and runtime environment (*Draco* middleware). CoConES uses contracts for timing and

bandwidth requirements that are specified and verified at design time and monitored during system execution. Contracts regarding memory use were planned at the time of writing. Besides monitoring, the middleware layer provides communication mechanisms for components as well as the creation and destruction of components at runtime.

CoConES builds systems out of *components* (as instantiations of reusable component blueprints), *ports* (as parts of the component interface) and *connectors*. Contracts with their nonfunctional constraints can be attributed to all three types, depending on their scope: contracts about memory use belong to components, those about timing constraints are attached to ports. In the latter case, so-called “hooks” are added to message sequence charts (MSC) specifying a port’s communication actions. Hooks start and end the time window during which a certain contract is active and monitored by the runtime system.

From the specification, the CCOM tool generates a skeleton code whose implementation is completed manually, using a custom language that is a superset of Java, with additional constructs for ports, messages etc. Contract violations are reported and analysed offline. An extension that triggers corrective measures at runtime is envisioned. A camera surveillance system is described as application example, reporting the successful replacement of some components by variants while reusing other components unchanged.

The authors specifically note that their approach is not aimed at hard real-time systems or applications that require a small memory footprint. The Draco middleware is Java based and intended to run on ES that provide significant processing power, for example, handheld computers or manufacturing robots.

Application examples. Camera surveillance system; code generator and dedicated runtime environment.

Contract classification. Nature: level 1-2, level 4 (timing, bandwidth); process moment: design time, runtime (monitoring, on-the-fly component exchange).

3.3.4 Assessment of approaches to contract awareness

CoConES: offers no possibility of abstraction/refinement; the approach resembles Simulink plus contracts. The contracts presented are unconditional assertions and thus more akin to conventional specifications.

SoftContract: expressiveness generally comes at the price of increased analysis effort, so LOC's limitations may be rather an asset, particularly since the focus of the SoftContract method lies on the generation of simulation monitors for on-the-fly checking, avoiding the use of more expensive verification techniques such as model checking or theorem proving.

3.4 Multiple-viewpoint specification & contract theory

In 2005, Damm & al postulated a "rich" component model that should be sufficiently expressive "to cover the complete development cycle from high-level specifications to design models" [55]. Since then, the Rich component model (RCM) has experienced many changes and additions. It is a concept still in flux, and this section tries to describe the common ground.

3.4.1 Rich components

To improve component reuse (in the context of automotive systems), Damm et al [55] propose a model of *Rich components* whose expressiveness would enable it to be used throughout the entire development cycle. Key features of the RCM are the integration of multiple viewpoints, especially those regarding "nonfunctional" properties such as timeliness, resource consumption, safety etc, and the use of assume-guarantee specifications for all of these viewpoints. State machines as well as variants of temporal logic are expected to serve as specification formalisms.

With a nod to the AUTOSAR [101] concept, the RCM assumes a layered design space comprising (1) a system layer, (2) a functional layer, (3) an ECU layer and (4) a hardware layer. This is the *vertical* dimension of the concept. The *horizontal* dimension facilitates the use of paraphernalia of

structured design: functional decomposition, abstraction, refinement etc. Figure 4 depicts the horizontal and vertical dimension, along with the corresponding assume-guarantee specification types. In the vertical dimension there are bottom-up and top-down specifications, explicating the requirements and interdependences of neighbouring layers (for example, performance requirements of a software component/function to an ECU). In the other dimension, horizontal assume-guarantee specifications describe, for example, compatibility requirements of neighbouring components on the same level of abstraction that need to work together, or relationships between different levels of abstraction (super- and subcomponents).

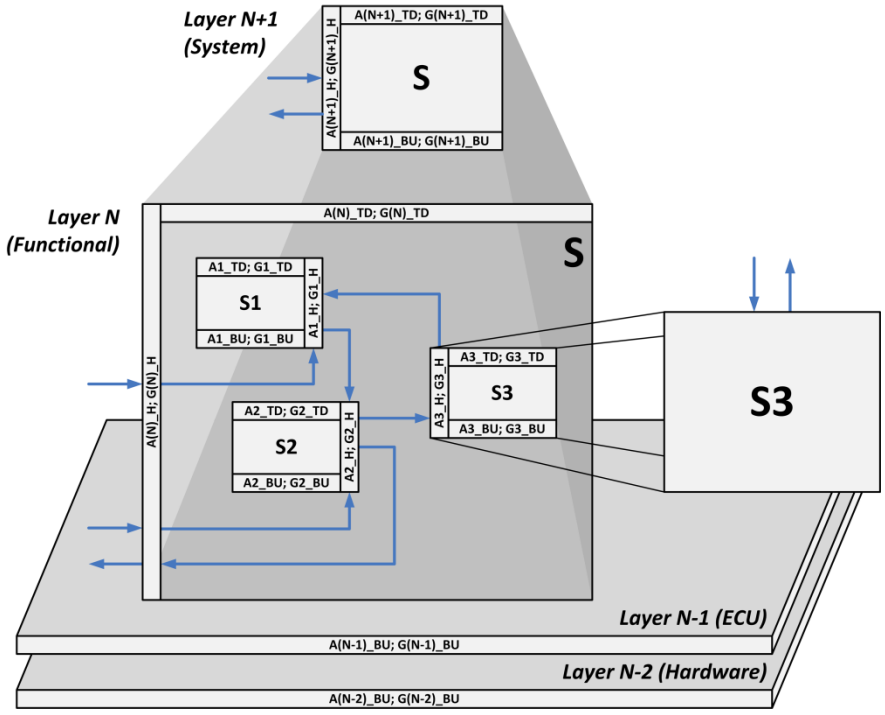

Figure 4. RCM design space layers

Specifications are expressed with separate automata for assumptions and promises, which are attached to viewpoints, components and ports. The complete component or system behaviour is derived by composition of all specification automata (or, equivalently, their traces). Composition takes place along the horizontal and vertical dimensions as well as between viewpoints that depend on each other.

Three types of analyses are considered: (1) a horizontal compatibility check determining whether a component's assumptions are satisfied by the guarantees of (interacting) neighbours or covered by assumptions of its supercomponent; (2) a substitutability test for components or their specifications, respectively; (3) a vertical analysis to ensure the realisability of higher-layer requirements with lower-layer resources. Together, these analyses should enable virtual integration during the design phase. [55] sketch the application of the RCM and its analysis methods to an embedded system for platoon driving.

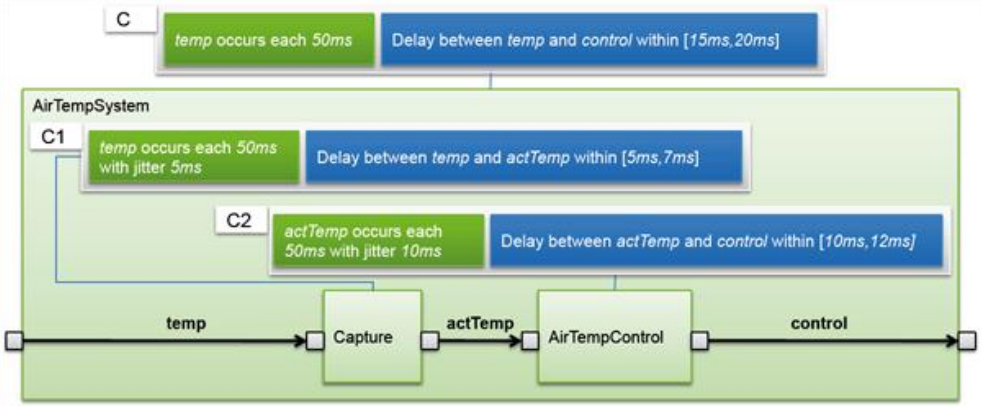

Figure 5. Real-time contracts in virtual integration testing [21]

In [23], Benveniste et al take up the idea of rich components (slightly rechristening them to *Heterogeneous* rich components, with a view to the integration of multiple viewpoints), explicitly call their assumption-commitment specifications "contracts", and augment the model with an *algebra of contracts* (see Sections 3.4.2 and **Fehler! Verweisquelle konnte icht gefunden werden.**) that formalises the construction of complex contracts from simple ones. A recent publication about the (H)RCM [21], however, eschews contract algebra in VIT examples and suggests the use of theorem proving to establish satisfaction, compatibility, refinement etc among contracts and contract compositions (Figure 5).

Application examples. Platoon driving [55]; production system controller [23]; various small examples from the automotive domain [21] [25] [26].

Contract classification. Nature: generic, with a focus on functional (level 1-2), real-time and safety (level 4). Process moment: design time, integration time, runtime (monitoring).

3.4.2 Contract theory

Benveniste & al [23] [24] [25] introduce contract algebra in the context of Heterogeneous rich components (HRC) to support "speculative design" and virtual engineering with contracts in a formal way. Based on execution traces, they define contract satisfaction and a refinement relation for contracts, as well as operators yielding composite contracts comprising (1) conjunction, which is for contracts referring to the same variables/components, (2) parallel composition, of contracts for different components/with different variables, and (3) fusion, as a combination of conjunction and composition, facilitating calculations with contracts from different viewpoints. Fusion computes the least specific contract satisfying a set of contracts, regardless of whether these belong to the same or different components.

HRC contracts differentiate between controlled and uncontrolled variables, which leads to a notion of compatibility where every variable/port can be controlled (included in a guarantee) by at most one contract. [25] adds to contracts the concept of strong and weak assumptions, where the violation of strong assumptions render a contract invalid (a further way of expressing compatibility), and weak assumptions are used to describe behavioural variants.

Glouche & al [81] [82] [83] propose a similar contract algebra, the major difference to HRC contracts being its formulation in terms of "process filters" and behavioural type inference. They illustrate their approach with the example of a simple four-stroke engine and the transformation of contracts to observers. Bauer & al [20] start from a more abstract "metatheory of contracts" and show how to derive from it Benveniste et al's contract algebra as well as operators for a contract theory based on modal specifications (see also 3.5.3).

Application examples. Various textbook examples [23] [24] [25]; four-stroke engine [81] [82] [83].

Contract classification. Nature: generic. Process moment: design time.

3.4.3 Assessment of contract algebra & rich components

With its inclusion in several European research projects (SPEEDS, COMBEST, SPES2020, CESAR, for example) the Rich component model belongs to an extensive body of industry-oriented research on embedded software component technology. From the changes the RCM has experienced over the years it appears to be aimed at becoming the swiss army knife of model-based development.

The RCM comprises, adapts and extends approaches originating from a multitude of sources, ranging from, for example, Hailpern & Ossher's view concept [89] via ideas of Goguen and Burstall [84] [85], Larsen's modal specifications [136], real-time interfaces, contract algebra, fault tree analysis, model checking, theorem proving, schedulability checking, simulation, testing etc, up to SEI's PECT framework [186]. Low-level specifications (Matlab models, Statecharts and C code) can also be integrated. Such eclecticism has advantages as well as drawbacks: on the one hand the RCM is becoming very flexible and expressive, on the other hand, interferences among different property theories and reasoning frameworks could arise [186].

3.5 Interface automata & modal contracts

The classes of interface automata (IA) and modal specifications can be seen as fairly orthogonal: IA are a game-based formalism with a focus on interface compatibility; modal specifications represent a fragment of the modal μ-calculus, offering a richer composition algebra (product, conjunction, residuum). Therefore it will be instructive to treat representatives of these classes side by side.[8]

3.5.1 Interface automata

De Alfaro & al [7] present a light-weight formalism for temporal aspects of software component behaviour, regarding the order (not the timing) of

[8] Because of their complementary strengths, Raclet & al [165], among others, propose a way to unify IA and modal specifications.

actions. Interface automata define input and output actions, and the composition of two IA consists in synchronising on complementary, shared in/output actions, and in interleaving other actions. Complementary actions become internal actions of the composition. Input actions denote incoming calls to methods offered by the component or the returning of outgoing calls, output actions represent outgoing calls or the returning of incoming calls. Constraints of a component (that is, assumptions) on the behaviour of its environment are implicitly present as all inputs that are not accepted. These are assumed to not be delivered by the environment (Figure 6, *fail* input is not accepted).

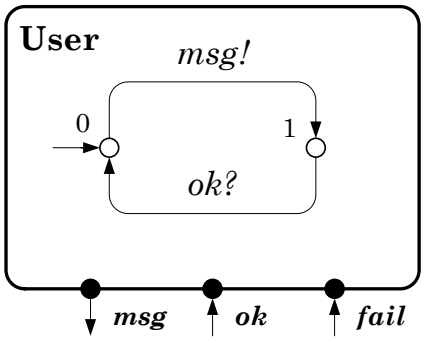

Figure 6. Interface automaton [7]

Component compatibility check thus represents an *optimistic* approach (as opposed to the more traditional pessimistic approach where components have to collaborate regardless of their environment): two components are compatible *if there exists* an environment that can make them work together. The composition of IA is their product automaton, with incompatible or unreachable states removed. Two components are compatible if and only if their composition is nonempty. Thus, automata composition can be seen as "a game between the product automaton (which attempts to get into an error state) and the environment (which attempts to prevent this)." [7] Thus, while composition requires solving a game, the compatibility check thereafter is trivial. The resulting automata are typically smaller than with a pessimistic approach which has to cope with all possible environments.

Since the optimistic view is environment constraining, assumptions can be used to "encode restrictions on the order of method calls, and on the types of return values and exceptions." [7] Accordingly, refinement is a choice among legal component behaviours without restricting the permissible behaviours of the environment. Refinement of IA is defined in the usual contravariant fashion of behavioural type systems,[9] checked by simulation (as opposed to language inclusion): an interface automaton P' refines an interface automaton P if P' simulates all input actions of P, and if P simulates all output actions of P'.

The authors, with collaborators, have later augmented IA with notions of timing and resource usage [9] [47] (see also Section 3.7.1).

Application examples. Simple message transmission service.

Contract classification. Nature: level 3 (synchronisation); process moment: design phase.

3.5.2 Interface I/O automata

Larsen & al [137] marry Lynch's I/O automata [141] to IA (Section 3.5.1) by splitting the interface of a component into one environment I/O automaton (representing the component's assumptions) and another one for specification I/O automaton (representing the component's guarantees). They define refinement between components (requiring identical signatures) in terms of set operations on input and output traces of their I/O automata. Component composition is performed as interface composition. The result is optimal in the sense that the environment automaton of the composition maximises the set of allowed input traces. Thereby, compatibility of the composite with an environment is maximised.

Larsen & al derive their composition method formally (as opposed to de Alfaro & al's interface automata, where the definition of composition is convincingly motivated but rather ad hoc) from the requirements of (1) independent implementability, (2) mutual deadlock freeness and (3)

[9] Liskov & Wing [139] introduce behavioural types; Lee & Xiong [138] build on IA in their treatment of behavioural typing; Maydl & Grunske [144] present a survey of behavioural typing for embedded software.

associativity of the composition operator. To find the solution of the composition they construct a modal transition system (MTS) (Section 3.5.3) from the interface I/O automata to be composed. The interface of the composite is then derived by fixpoint iteration over the MTS. The authors also define a refinement relation for interface I/O automata with identical signatures by contravariant trace inclusion and additional conformance tests.

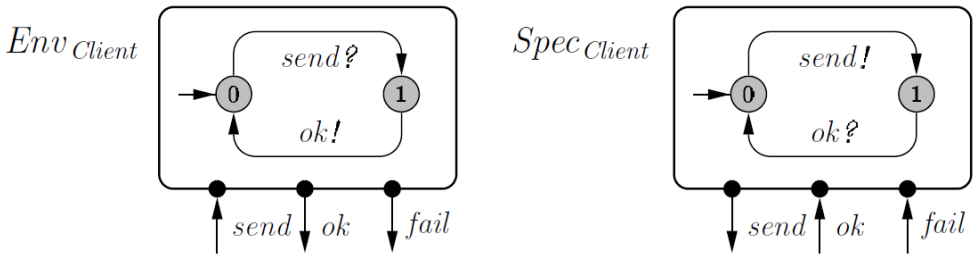

Figure 7. Exemplary interface of a client component as pair (Env_{Client}, $Spec_{Client}$) [137]

The authors note that a further advantage of splitting interfaces is reuse: guarantees (as well as assumptions) may be part of different interfaces. In the context of product lines, for example, "a family of component variants may be specified using a single specification (guarantee) and multiple environmental restrictions (assumptions)." [137]

Application examples. Textbook example of a communication protocol.

Contract classification. Nature: level 3 (synchronisation); process moment: design phase.

3.5.3 Modal contracts

Gössler & Raclet [87] use modal specifications,[10] introduced by Larsen [136], to define modal contracts. As opposed to interface I/O automata (Section 3.5.2), where the assumption of the composition of two contracts ensures that both of their guarantees are fulfilled (*eager composition*), the

[10] Modal specifications offer a special way of refinement through the concept of *may*- and *must*-transitions. May-transitions specify optional behaviour which later, in a refined model, can be changed to must-transitions representing mandatory behaviour. In diagrams, may-transitions are denoted by dashed lines while solid lines represent must-transitions.

composition of modal contracts must fulfil each guarantee if and only if the corresponding assumption is satisfied (*lazy composition*). Lazy composition thus enables a component to provide different guarantees in different environments.

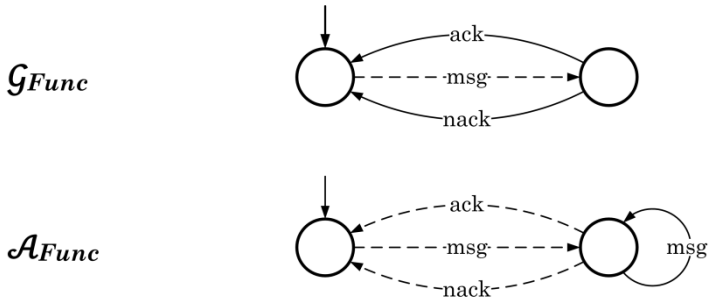

Figure 8. Communication channel – contract of functional aspect, $\mathcal{C}_{Func}$ [87]

Modal contracts differentiate between *component contracts* and *requirement contracts* called "aspects". Both have the same syntax but are composed in different ways. Their semantic difference is that component contracts describe the guarantees and assumptions "inside out" from a component-centric perspective, whereas aspects describe properties expected from a component under specific circumstances ("outside in") from a developer/environment-centric perspective.

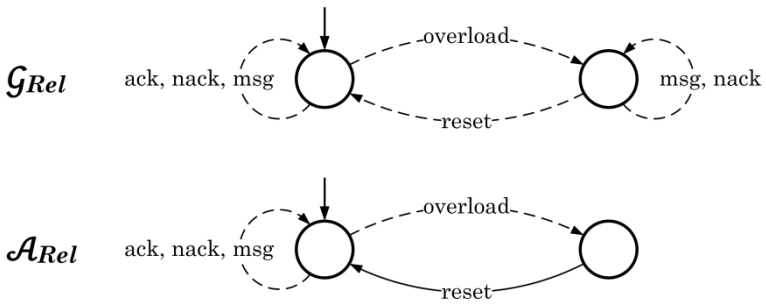

Figure 9. Communication channel – contract of reliability aspect, $\mathcal{C}_{Rel}$ [87]

As an example, Figure 8 and Figure 9 depict two aspects of a communication channel component: functionality and reliability. The functional contract guarantees that every message will be acknowledged with *ack* or *nack*, under the assumption that messages are retransmitted as long as they have not been acknowledged. The reliability contract

guarantees that, when overloaded, messages can only be acknowledged with *nack*, assuming that on overload there will certainly be a reset.

A special feature of modal contracts is the operation of priority composition of aspects, whose result obeys lower-priority aspects if no higher-priority aspect makes a choice about an action. Priority composition is associative, enabling incremental design (which is, however, only valid as long as no intermediate-priority aspect needs to be "slipped in" later, because priority composition is, of course, not commutative).

Application examples. Simple model of a communication channel with functional and reliability aspects.

Contract classification. Nature: level 3 (synchronisation);[11] process moment: design time.

3.5.4 Assessment of interface automata & modal contracts

Interface I/O automata: appear to be primarily directed towards modelling communication protocols; neither timing nor data values can be considered; event communication. A disadvantage of the approach could be that there is but a single contract per component, and no definition of the composition of several contracts for the same component exists. Compared to conventional IA, their main strength is the separation of required and provided interfaces. Furthermore, as with IA, the optimistic approach to composition promises smaller product automata (although the extent of savings remains inconclusive).

Modal contracts: have the possibility to model multiple aspects/viewpoints and ensure independent implementability, thus offering support for distributed development.[12] Priority composition is an interesting and unique concept, but it compromises incremental design because priority composition is not commutative. Modal contracts (called "modal interfaces" there) are also included in the Rich component model (Section

[11] As shown in Figure 9, the reliability aspect is realised by a function performing a reset in case of channel overload; therefore the nature dimension is not regarded as being level 4.
[12] See [165] and [166] for a comprehensive assessment of modal specifications.

3.4.1). As opposed to conventional modal specifications, modal contracts allow an explicit treatment of assumptions and guarantees [62].

3.6 Component substitutability & evolution

The monolithic verification of a safety-critical system has to be repeated as a whole whenever properties of one or several constituents change, as is the case, for example, with evolving designs, bug fixes, system upgrades or variants. Approaches to substitutability and evolution serve to minimise the reverification effort.

3.6.1 Safety interfaces

Elmqvist and Elmqvist & al [67] [68] propose safety interfaces to reason about the safety of a component system in a modular way, supporting the addition or modification of components. Their model is based on Reactive modules [12], a formal model for synchronous and asynchronous concurrent systems that enables incremental design (by a notion of refinement) as well as conditional assurance. Safety interfaces are an add-on specification of a component's behaviour in the presence of faults in its environment. They are intended to complement the functional specification by defining formal safety properties in the form of environment assumptions under which a module is immune against single and double "input faults". These assumptions are constraints on the environment, thus, the safety interface rules out behaviours of the environment that will trigger a failure of the module.

The method to define a safety interface starts out with searching for the least restrictive set of environment assumptions E such that the respective module M satisfies a given safety requirement φ. This is done using a fixpoint iteration: model-checking the module until all environment behaviours that lead to the violation of φ by M have been removed through additional assumptions/constraints on the environment. Thus, $M \parallel E \vDash \varphi$ (E now represents an environment in which the component fulfils safety requirements in the presence of single and double faults). This is similar to the CEGAR method.

Based on E representing an environment without faults, a safety interface is defined as a triple that contains E, X and Y, where X is a pair of (1) a fault resilience set and (2) an environment that captures those assumptions necessary to make the module resilient to all single faults from the resilience set. The third part of the safety interface is an analogous tuple of pairs of fault pairs and an environment that makes the module resilient to the respective combination of two different faults.

The safety interfaces of a component/module system enable the analysis of the safety of the overall system against single or double faults without having to compose the system. Modular analysis is based on the assumptions that (1) a module resilient against a particular single value fault will not propagate it, such that it constitutes a safe environment for the other modules if it experiences just this one and no further fault, and that (2) a module constitutes a fault-tolerant environment for the rest of the system as long as it does not experience any fault.

As an example (quoted by the authors), assume that the least constrained environment specifies that the safety property is fulfilled as long as the disjunction of Boolean variables a and b is true. The safety interface for the single value fault resilience of variable a would then constrain the environment to offering b as always true. Of course, an additional fault of the other variable would not be tolerable. Also, if a and b are assumed to be positive integers between 0 and 10 and their sum never more than 20, there would be no environment that could make the module resilient against a single value fault of either a or b.

The safety interface technique could be described as a “complemented FMEA” for single and double faults, where instead of analysing the incoming faults that lead to the failure of a module those incoming faults are specified that *do not* lead to a failure. All other cases are left unspecified, and thus the method yields an underapproximation of the actual reliability. The application to addition/modification of components is indirect.

Application examples. Hydraulic leakage detection system of an aircraft.

Contract classification. Nature: level 4 (fault tolerance, safety); process moment: design time.

3.6.2 Substitutability & verification with evolving software

Chaki & al and Sinha [44] [45] [176] tackle the substitutability problem in a more direct way. They propose a method to localise the reverification effort after modifications of one or several components to *only* those components. Since the environment can be modelled as a component, too, the approach can also be applied to environmental changes. Components are assumed to be realised by C programs that communicate via blocking message passing. Their method extracts automata models from component implementations. To check substitutability of new for old components, two distinct checks are applied: containment check and compatibility check.

Containment means that a modified component must offer at least the same behaviours as the original one. This is verified by a CEGAR-like iterative refinement of over- and underapproximations of the behaviours of the original and the modified component, and by checking if the overapproximation entails the underapproximation. Approximations are obtained using predicate abstraction (overapproximation) and modified predicate abstraction (underapproximation). In case behaviours are missing from the modified component, the algorithm provides feedback. If the lack is not accidental, for example, after a bugfix removing undesired behaviour, the modification is in order if it offers at least as many behaviours as the original one without the bug.

Compatibility ensures that the additional behaviours of the modified component do not violate global safety requirements. The compatibility check again proceeds in a CEGAR-like manner: in the end, there is either a valid counterexample for compatibility, or compatibility is asserted. To generate the necessary assumptions for the assume-guarantee reasoning, a learning algorithm is used. In the case of reverification, already generated assumptions are reused if possible instead of starting with an empty set of assumptions; this leads to effort reduction. The verification job is decomposed using a non-circular rule for CA; by recursive application the

modified system can be verified without actually composing it. A special aspect of this substitutability concept is that it does not require refinement but "only" compatibility.[13]

Hung & Katayama and Hung [112] [113] have a similar approach; it is, however, a simplified version of the approach of Chaki & al and Sinha: they do not consider the removal of behaviour (meaning no bugfixing, for example), therefore, the containment check can be skipped.

Application examples. Analysis of an interprocess communication (IPC) protocol with seven components (given as C source code) and a state space size of 10^6 (Chaki & al); benchmarks derived from a portion (handshake protocol) of the OpenSSL source code by aggressive abstraction from an infinite state space; several benchmarks derived from the NuSMV tool set, from an IPC library used by an industrial robot controller software, as well as benchmarks based on the classical dining philosophers problem (Sinha).

Contract classification. Nature: level 3 (synchronisation); process moment: design phase, verification phase (since models were derived from C implementations).

3.6.3 Service compatibility & substitutability

Ouederni, Salaün & Pimentel and Ouederni & Salaün [158] [159] treat the question of compatibility and substitutability in the context of (web service) protocols. They argue that the consideration of observable behaviour only (as it is done, for example, in trace inclusion) is insufficient to ensure that two services communicating via synchronous message passing can be composed or substitute each other without deadlock. The reason for this are internal, unobservable actions (denoted τ transitions) that may lead services into incompatible states, while in a more abstract model of the same services, without τ transitions, they seem to be compatible. τ transitions may, for example, describe timeouts.

As an example, consider Figure 10: a process $S2$ executes output action $b!$ and is compatible with process $S1$ (left side) which accepts input action $b?$.

[13] See also Ouederni & Salaün (Section 3.6.3): a component could be compatible with the same environment as another, but still not be a replacement because it might serve entirely different purposes.

If $S1$ is replaced with the more detailed specification $S1$' (right side) it becomes obvious that $S2$ and $S1$' are actually incompatible, because as soon as $S1$' executes the internal action tau, it can no longer accept $b?$ as input.

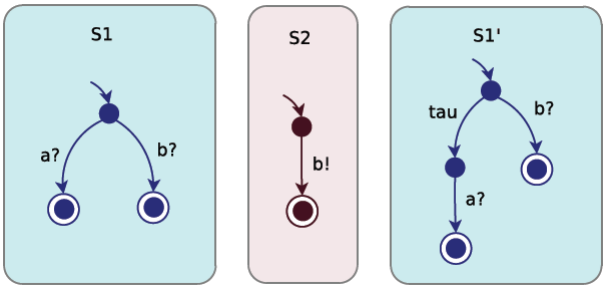

Figure 10. Trace equivalence and compatibility [159]

The authors propose different notions of compatibility with increasing strictness, where deadlock-freeness is the least strict. Furthermore, it is argued that checking compatibility to determine substitutability may be misleading since a compatible replacement might have entirely different behaviours and thus not be substitutable in a semantic way. (compare to Chaki & al, Section 3.6.2).

Therefore, it should be ensured that a replacement is related to the original component "by a certain *relation*" [159] (emphasis in the original). Such relations again come in various degrees of strictness: from trace equivalence via weak bisimulation, branching equivalence up to strong bisimulation. The authors propose to use branching equivalence since trace equivalence can be too loose to preserve compatibility and it can be checked more efficiently than weak bisimulation while at the same time being stricter. Strong bisimulation will be too restrictive because it requires matching all tau transitions, besides observable actions.

The authors conclude by formulating four challenges in compatibility and substitutability analysis: (1) generalising from the consideration of two interacting components to several components, (2) extending the model to asynchronous communication, (3) proving that branching equivalence is superior to weak bisimulation for substitutability checks, and (4) instead of

returning a yes/no answer, actually *measuring* the degree of compatibility and substitutability (first results have been reported in [158]).

Application examples. Web service protocol.

Contract classification. Nature: level 3 (synchronisation), level 4 (quantified degree of compatibility/substitutability).

3.6.4 Assessment of approaches to component substitutability & evolution

Safety interfaces: enable qualitative analysis of single and double value errors. The possibility to verify fault tolerance, besides (formal) safety properties, adds to the interest. As with most techniques, liveness cannot be treated. Common mode/cause failures cannot be analysed.

Substitutability & verification with evolving software: interestingly and contrary to other techniques, the approach does not require refinement between original and modified components. At the same time, the missing notion of refinement means that the method is primarily aimed at the testing phase; it may be less useful for the design phase. A crucial step for CA is the generation of useful assumptions to decompose reasoning: in the validation example provided, all previously generated assumptions could be reused for verification of the modified system, thus, the validity of the reported speedup in the general case is questionable. This approach is an extension of Cobleigh & al [51] (Section 3.7.3); in consequence, the proof rule used is not applicable to circular structures.

3.7 Further approaches

This section considers more specialised or cutting-edge approaches to conditional assurance.

3.7.1 Timing

Discrete real-time systems. Ostroff [156] proposes a framework for the compositional design and analysis of real-time systems. It is based on timed transition models (TTM) for system modelling, and on real-time temporal logic (RTTL) as a language for the specification of requirements on a TTM. RTTL is a timed extension of linear temporal logic (LTL), as a TL

plus tick transition (= discrete time). In the TTM/RTTL framework, a system consists of concurrent components. These components, called “modules”, can be composed using a composition rule that facilitates the verification of the composite’s correctness (3.7.1.a: the composed system satisfies a global requirement r if this requirement follows from the conjunction of module specifications). The composition rule supports bottom-up as well as top-down development.

(3.7.1.a)
$$\left.\begin{array}{c} m_1 \vDash s_1 \\ m_2 \vDash s_2 \\ (s_1 \wedge s_2) \rightarrow r \end{array}\right\} m_1 \parallel m_2 \vDash r$$

TTMs are model checked to show their fulfilment of local requirements. The satisfaction of the composed system of global requirements is done using theorem proving. The framework also offers a (restricted) notion of refinement, called “implementation”; a corresponding rule ensures that a refinement preserves the higher-level specification.

The framework allows RTTL requirement specifications to be conditional, corresponding to an assume/guarantee-like implication $\psi \rightarrow \varphi$, with ψ and φ being RTTL expressions.

Timed interface automata. In [9], de Alfaro, Henzinger & Stoelinga extend interface automata [7] (Section 3.5.1) by a notion of (real-valued) timing of inputs and outputs, expressing temporal constraints on component interactions. Timed interface automata (TIA) have a synchronous communication model.

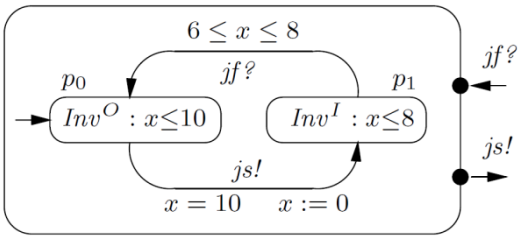

Figure 11. Timed interface automaton

Similar to the untimed approach, TIA specify a game with the environment. The rules of the game are described by their timed interfaces. Again, the

notion of compatibility is optimistic: interfaces are compatible if there exists an environment in which they can work together, such that no component violated its deadlines. Subtyping (refinement) is possible for TIA [144], but a corresponding explicit relation is not defined.

As an example for a TIA, Figure 11 shows the specification of an application that periodically triggers another process (not shown) every ten seconds by executing $js!$ (job start). Variable x is a clock measuring the time that has elapsed since the last triggering action. The output and input invariants Inv^O and Inv^I are the deadlines for the corresponding actions. In the start state p^0 the application can spend time (and possibly do other things) until the clock has accumulated ten time units. Upon $x = 10$, the output action $js!$ is executed, the clock is reset ($x := 0$) and the triggered process must return with input action $jf?$ (job finished) within six to eight time units. After that, the application can spend the rest of the time, until the clock again reaches ten time units, in p^0.

Real-time interface algebra. The real-time interfaces (RTI) of Henzinger & Matic [97] subsume a group of tasks in a bounded-delay resource model which guarantees the allocation to a task of a specified number of timeslots in a specified time interval. This imposes a fixed limit on the waiting time of a component for processing capacity. The functional component specification is a set of task sequences. Its resource consumption is a function of the workload and the maximum delay clients of the component can tolerate for execution. The component interface describes the required processing power by a capacity function. It asserts that if the environment provides resources not less than the capacity needed, and if incoming requests do not exceed a specified rate, then the component produces outputs with a delay satisfying the deadlines. Scheduling analysis can then be applied to determine schedulability of a component assembly.

The algebra of RTI provides formalisations of interface composition, the mapping of task sequences to an interface and a refinement relation. Associativity of interface composition facilitates incremental design because it implies that components can be integrated in any order. Interface compatibility is checkable before the system specification is

complete. Refinement ensures that specifications, once they have been shown to be compatible, can be independently developed because their compatibility will be retained.[14]

Application examples. Shutdown system of industrial nuclear reactor (Ostroff); textbook scheduling example (de Alfaro, Henzinger & Stoelinga); real-time robotic application (Henzinger & Matic).

Contract classification. Nature: level 4 (timing); process moment: design phase.

3.7.2 Stochastic modelling & analysis

Delahaye & Caillaud present a method for probabilistic reasoning on contracts [60] to analyse reliability aspects of a system in a modular way. For this purpose they extend the notions of contract satisfaction, composition and refinement (based on system runs/traces) to the stochastic domain. The source of randomness in this model is the environment which can randomly set the value of a subset of the system's uncontrolled variables (that is, those corresponding to inports that are not controlled by other contracts).

An implementation *M* probabilistically satisfies a contract $(A; G)$ with a *level* (probability) of $\beta \in [0, 1]$ if, whatever the probabilistic choices of the environment may be, the proportion of the number of those runs that fulfil the guarantee G against the number of all runs is at least β. Given that two implementations, M_1 and M_2, satisfy their contracts, C_1 and C_2, with levels of α and β, respectively, their composition, $M_1 \times M_2$, satisfies the parallel composition, $C_1 \parallel C_2$, of their contracts with a level of $\alpha \cdot \beta$. Probabilistic refinement between two contracts is defined in a similar way. It should be noted that the prerequisite for the compositionality of this method is the independence of the definition of satisfaction/refinement levels from the inputs from the environment.

[14] Easwaran & al [66] propose an interface theory and algebra with similar properties as Henzinger & Matic's (see above): demand interfaces. Composition, refinement, incremental design, compatibility and schedulability are formally defined/supported. The main difference to [97] is the ability to specify varying resource models and scheduling algorithms, while Henzinger & Matic rely on earliest-deadline-first scheduling.

In [59] [61] [62], Delahaye & al propose an approach where random variables can be output as well as input variables. Probabilistic choices are resolved by a set of schedulers associated with a system.[15] There are two types of probabilistic satisfaction: P-A and P-R satisfaction. P-A satisfaction corresponds to a notion of availability, while P-R satisfaction is akin to reliability, with runs R-satisfying a contract if and only if all runs that satisfy the assumption are included in the guarantee. A system has a level of P-R satisfaction of α when the fraction of its runs that either satisfy the guarantee or do not satisfy the assumption is at least α. This means that runs violating the assumption are considered as "good" and do not need to satisfy the guarantee, which is the reason why this method can be compositional. Probabilistic refinement (P-refinement) is defined as well.

The main result of the article is the derivation of a lower bound for probabilistic satisfaction and refinement: if two systems P-A-/P-R-satisfy their contracts with levels α and β, respectively, their composition will satisfy the parallel composition of the contracts with a level of at least $max(0, \alpha + \beta - 1)$. Furthermore, in analogy to [60], if a system S P-R-satisfies its contract C with level α and C P-refines contract C' with level β, then S P-R-satisfies C' with a level of at least $max(0, \alpha + \beta - 1)$.

Xu & al [188] describe a method similar to that of Delahaye & al: while the latter use Markov decision processes to handle stochasticity, the former focus on interactive Markov chains.

Kwiatkowska & al [131] represent contracts by *probabilistic assume-guarantee triples* of the form $\langle A \rangle_{\geq p_A} M \langle G \rangle_{\geq p_G}$, with the meaning: whenever the environment of component M (that is, the system M is a part of) satisfies assumption A with a probability of at least p_A, the system (or the component, respectively) will satisfy guarantee G with a minimum probability of p_G. A and G have to be formal safety properties.[16] These safety properties are modelled by probabilistic finite state automata, and their fulfilment verified by multi-objective model checking for probabilistic

[15] The use of schedulers in this context was previously proposed by de Alfaro & al [10].

[16] As opposed to the fulfilment of a liveness property, a trace prefix that does not satisfy a formal safety property cannot have an extension that satisfies it (for this reason, the violation of a safety property is generally easier to check than the violation of a liveness property).

automata. The authors present several compositional proof rules that can be used (*Asym*, *Asym-Mult*, *Asym-N*, *Circ*, *Async*). They validate their approach with its application to two case studies (a randomised consensus algorithm and a network configuration protocol), comparing the performance of the proposed technique with more conventional verification by the probabilistic model checker PRISM [111]. They observe that the compositional approach is faster in most cases and that "the numerical values produced […] are generally good."[17]

Application examples. Textbook examples (Delahaye & Caillaud); simple dependable computing system with time redundance (Xu & al); randomised consensus algorithm, network configuration protocol (Kwiatkowska & al).

Contract classification. Nature: level 4 (probability); process moment: design phase.

3.7.3 Learning

Finding appropriate assumptions to decompose a system for assume-guarantee verification is hard, and (similar to finding loop invariants, for example) there is no generally valid algorithm to follow. Nevertheless, learning approaches can be used to automatise the process.

Cobleigh & al [51] were the first to present a learning-based technique (akin to CEGAR) to obtain assumptions for the compositional verification of formal safety properties (Figure 12). Properties are represented by labelled transition systems (LTS). The learning process "converges to an assumption that is necessary and sufficient for the property to hold in the specific system." The approach is exemplified by analysis of a design-level model of a subsystem for the K9 Mars rover developed by NASA. While the state space could be reduced with compositional verification, the time complexity increased due to the necessary learning. The authors conclude that the observed improvement in terms of memory requirements outweigh the resulting time overhead.

[17] In fact, the experiments that use a combination of the proof rules Async and Asym yield exact results, while those based on the Circ rule tend to overestimate error probabilities by an order of magnitude or more.

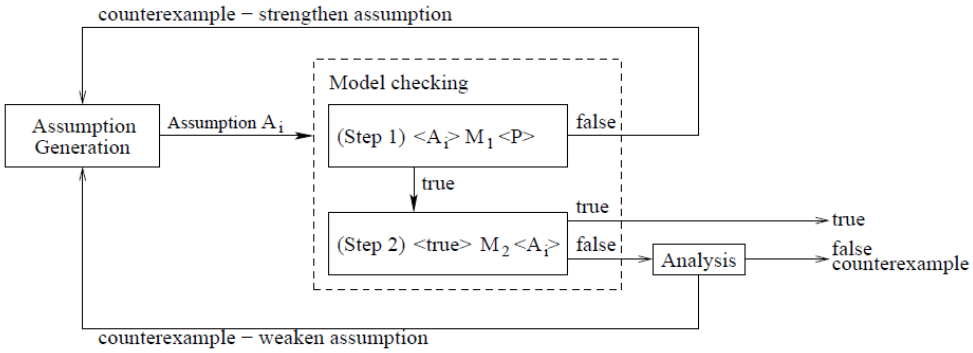

Figure 12. Counterexample-guided learning algorithm [51]

Emmi & al [70] apply the learning approach to the verification of assume-guarantee specifications modelled by interface automata, in basically the same fashion as Cobleigh & al (see above). Their case study is based on the high-level description of a protocol for *Autonomous rendezvous and docking* (ARD) of a spacecraft. Again, compositional analysis shows a reduction of the state space accompanied by an increase of analysis time.

Feng & al [71] combine the probabilistic assume-guarantee approach of [131] (Section 3.7.2) with a learning technique to generate *probabilistic* assumptions. The results of the case study, a flight software module of Mars exploration rover verified by use of proof rules Asym and Asym-N [131], are mixed: while conventional (noncompositional) analysis is much faster in general, there are some cases where compositional analysis wins out. Furthermore, in a number of instances conventional analysis cannot finish ("mem-out") while compositional analysis succeeds; in others, it is the other way around.[18]

Application examples. K9 Mars rover (Cobleigh & al); ARD protocol (Emmi & al); flight software module of JPL Mars rover (Feng & al).

Contract classification. Nature: level 3 (synchronisation) (Cobleigh & al, Emmi & al), level 4 (stochastics) (Feng & al); process moment: design phase.

[18] Differences may be related to the proof rule(s) used in the respective experiments; the authors just mention this aspect but do not investigate it further.

3.7.4 Assume-guarantee testing

Blundell & al [32] and Giannakopoulou & al [80] introduce assume-guarantee testing as a testing method based on the automatic derivation of local requirements from system-level requirements in the design phase. Local requirements are then used for unit testing, enabling the detection of system-level requirements violations prior to the actual integration tests. These unit tests can be performed as soon as a unit/component becomes code complete, and before the implementation of other components is finished.

Application examples. ARD protocol, K9 Mars rover (see 3.7.3).

Contract classification. Nature: level 3 (synchronisation); process moment: design phase, unit testing.

3.7.5 Locality

Lomuscio & al [140] propose a bounded approach to assume-guarantee reasoning that exploits dependences between components in an incremental fashion. The background of this method is the “assumption explosion” that affects compositional reasoning (Section 3.8). Taking advantage of locality can serve to reduce assumption size since each component usually interacts with only a limited number of other components in the system.

The bounded rule applied by the authors tolerates circularity and is shown to be sound and complete, also for liveness properties. Experiments with a simple example of a network topology (verifying the stability of a distributed congestion control system) demonstrate a significant reduction in the number of generated assumption without loss of expressiveness.

Application examples. Distributed congestion control system.

Contract classification. Nature: level 3 (synchronisation); process moment: design phase.

3.7.6 Parameterised contracts

Reussner and Becker & al [22] [168] [170] propose a generalisation of Design by contract: parametric contracts that describe how the behaviour and local failure flow of a component depends on the way provided and required interfaces rely on each other.

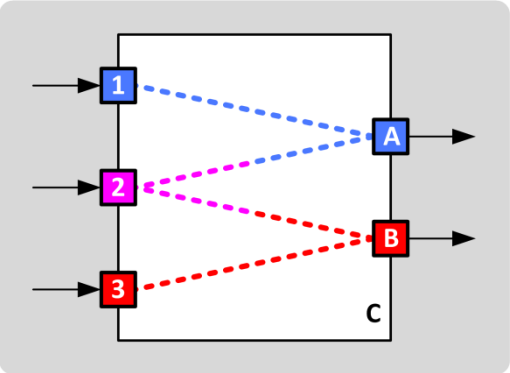

Figure 13. Conditioning component provisions on requisitions (and vice versa)

In the component model parametric contracts are used to describe which input interfaces have to be available for an output interface to be used. In this way it becomes possible to use only subsets of component interfaces, depending on those services provided and required by the component's current environmental context (Figure 13). There exists a probabilistic extension of the approach [169].

Application examples. Textbook examples.

Contract classification. Nature: level 3 (synchronisation); process moment: design phase.

3.7.7 Goal structuring notation contracts

Bate, Hawkins & McDermid [18] augment Kelly's goal structuring notation (GSN) [19] [125] with interfaces and contracts to improve the evolvability and reusability of modular safety cases for certification. In this way, changes can be made to component contracts without then having to repeat the complete system verification; only the changed modules must be recertified, if their contracts are still valid. Timing and safety contracts

are considered. Conmy, Nicholson & McDermid [54] use GSN contracts to define a modularised safety analysis process for integrated modular avionics (IMA). They apply it to the integration of separately developed hardware interface layer software into a navigational display system in the context of IMA, focussing on the properties of computer resource management software and operating systems and their modular safety cases.

Application examples. Navigational display system (IMA) (Conmy & al); aircraft stores management system (Bate & al).

Contract classification. Nature: level 4 (safety); process moment: design phase, certification.

3.7.8 Assessment of further approaches

Timing: Ostroff makes an interesting case for reverse engineering with the example of a reactor control. The refinement concept of his approach, however, allows just a single level of refinement (called “implementation”); there are no hierarchies of refinements that could be used to facilitate top-down development.

Stochastic modelling and analysis: a probabilistic method for compositional reasoning would be of great importance in dependability engineering (although the stochastic treatment of software behaviour is an open issue). The approach of Delahaye & Caillaud and Delahaye & al, however, depends on their decision to consider contracts as fulfilled when the assumption is not satisfied. In this way, stochastic dependences from common causes are disregarded, but is is exactly these dependences and causal chains that are indispensable for reliability assessment. Furthermore, it is not possible to describe reliable systems built from less reliable components since, based on the proposed method, one cannot argue that a system is more reliable than its single components. The lower bound the authors derive for reliability is equivalent to stating that the upper bound for the unreliability of a system is the sum of component unreliabilities: it is actually impossible for a composite system to do any worse. Thus, their result is a rather weak one.

Assume-guarantee testing: appropriate model-based coverage criteria remain to be developed.

Locality: this technique is especially interesting because it addresses the essential weakness of compositional reasoning, namely its efficiency problem.

3.8 Modular reasoning: is it efficient?

Kupferman & Vardi [129] prove that various combinations of linear and branching time TL, together with compositionality possess the same theoretical complexity as monolithic reasoning. They observe that modular reasoning trades state space explosion for assumption size explosion. Cobleigh & al [52] [53] report that even in practice experimental results regarding the effort of compositional reasoning with learning “are discouraging”. In most cases they observe a significant increase in effort compared to monolithic verification.

The complexity problem of compositional techniques might be less severe with real-world systems because of their tendency to have a “benign” structure, and because, as result of a top-down design process, requirements on higher abstraction levels will often be present.

4 Application scenarios for conditional assurance

In collaboration with the FAT working group, out of the surveyed approaches one promising CA/CS approach was selected, as well as four main scenarios for its application. This chapter reports these scenarios, suitable approaches, and the resulting decision on the approach selected.

4.1 Conditional specifications for automotive component systems

The ability to formulate conditional specifications (contracts) of ES properties constitutes the basic requirement for the method to select here to enable modular reasoning. Optionally, the handling of variants may be considered.

Suitable approaches. Depend on the properties to express (timing, probabilities etc) and the formalism an approach is based on. All methods discussed in Chapter 3 are modular/compositional; as a property metatheory, the most generic of them is contract algebra. Some, such as GSN contracts or Assume-guarantee testing for example, will be too specialised to consider them for a first, exemplary application.

4.2 Virtual integration & reuse

Virtual integration requires formal composition operators for conditional specifications.

Suitable approaches. Contract algebra (integration); substitutability approaches (reuse).

4.3 Composition of function & monitor

The composition of a function and its monitor is a special case of virtual integration.

Suitable approaches. All methods which lend themselves for contract formulation, and which offer a composition operation.

4.4 Handling of changes

(Optional) With two levels of refinement, what happens when there is a change on the lower level? How does this affect the higher level? When there is a change of the higher level, how does this affect the lower level?

Suitable approaches. Approaches to component substitutability and evolution.

4.5 The selected approach

Because of its genericity, elaboration and broad publication base, contract algebra was determined as the preferred method to apply in the case study.

5 Exemplary application of contract-based verification

To obtain a preliminary assessment of the general applicability of CA approaches in automotive software, one particular approach was chosen and applied to an industrial example as a case study. This approach was *contract algebra*, and the example was a software-realised function from the drivetrain domain with safety-critical aspects. The example was provided by one of the industrial members of the AK 31. It consisted of a simplified software model and formal subsystem requirements from the validation of the function.

For the evaluation, the contractual specifications of the model's components were derived, and a safety requirement was verified by use of contract algebra. Additionally, the composition of a speedometer function and a monitor was demonstrated for an academic example.

In this chapter, first, the selected formalism is introduced, and second, the main results of the case study are reported. Details of the example are omitted for confidentiality reasons. Finally, the composition of components in the additional academic example are presented and evaluated.

5.1 Contract algebra

Contract algebra [20] [23] [24] [25] [82] [83] is a metatheory of properties that can be applied with various specification theories, such as automata, traces, temporal logic, modal transition systems etc. The following exposition uses logical connectives.[19]

A contract $C = (A; G)$ is a pair of assertions, with an *assumption* A and a *guarantee* G, which have an implication-like relation: if the assumption is satisfied, the guarantee will be fulfilled, too. With assertions on variable/port values, an example of a contract for an adder component could be

[19] A set-theoretic representation would feature the corresponding set operators: $\subseteq$ for $\rightarrow$, $\cap$ for $\wedge$ etc.

(5.1.a) $$(x \in \mathbb{Z}, y \in \mathbb{Z}; z \in \mathbb{Z} \wedge z = x + y),$$

simply stating that if (the inputs) x and y are whole numbers, (the output) z is a whole number and also the sum of the inputs.

Refinement. Refinement (also called "dominance")[20] is a relation $\preccurlyeq$ between two contracts $C = (A; G)$ and $C' = (A'; G')$ such that C' refines C if and only if A implies A' and G' implies G:

(5.1.b) $$C' \preccurlyeq C \leftrightarrow A \rightarrow A' \wedge G' \rightarrow G.$$

Thus, in the context of component systems, C' (or a component satisfying C') can replace C (or a component satisfying C) because it assumes no more than C and guarantees at least as much (contravariance). With assertions about variable value ranges, an example for refinement could be

(5.1.c) $$(x \in [-8, 8];\ y \in [0, 0.5]) \preccurlyeq (x \in [-5, 5]; y \in [0, 1]),$$

meaning that the contract on the left assumes less about input variable x (allowing a wider range for it) and offers a stricter guarantee on output variable y (providing for a tighter range), and therefore it is a refinement of the contract on the right. The refinement relation induces a partial order on the set of contracts.

Conjunction. The greatest common lower bound of two contracts[21] $C_1 = (A_1; G_1)$ and $C_2 = (A_2; G_2)$ in the partial order is their conjunction

(5.1.d) $$C_1 \wedge C_2 = C_{1,2} = \left(A_{1,2}; G_{1,2}\right) = (A_1 \vee A_2; G_1 \wedge G_2).$$

The greatest lower bound conjoins contracts that refer to the same variables/ports, so they belong to the same component. The contract $C_{1,2}$

[20] Authors that differentiate between more abstract and more concrete specifications ("implementation", code etc) sometimes reserve the term "refinement" to describe the relation between two implementations, and use "satisfaction" for the relation between an implementation and a specification. In that convention, specifications (contracts) are related by *dominance*.

[21] In the following, contracts are assumed to be in their normal form, which simplifies the rules of the algebra. Normalisation of a contract implies its guarantee by the assumption, thus, for any contract $C = (A; G)$, the contract $C^{Norm} = (A; A \rightarrow G)$ or, equivalently, $(A; \neg A \vee G)$ is in normal form. Normalisation is idempotent.

refines both C_1 and C_2 because $A_1 \rightarrow A_{1,2}$, $A_2 \rightarrow A_{1,2}$, $G_{1,2} \rightarrow G_1$and $G_{1,2} \rightarrow G_2$.

Composition. Parallel composition, denoted ‖ or ⊗, combines two contracts belonging to different components (referring to different variables/ports). Essentially, it yields the conjunction of assumptions as well as of guarantees:

(5.1.e) $$C_1 \otimes C_2 = (G_1 \wedge G_2 \rightarrow A_1 \wedge A_2; G_1 \wedge G_2).$$

The assumption of the composition, however, is relaxed by implication with the guarantee of the composition because assertions in the assumption of one contract that are satisfied by assertions from the guarantee of the other contract do not anymore need to be satisfied by the environment of the composition [24].

Incremental design & independent implementability. More operations can be defined for contract algebra, but conjunction and composition already suffice to ensure the properties of incremental design and independent implementability:

- Contract conjunction as well as contract composition are commutative and associative, so both operations can be performed in any order. For associativity: $(C_1 \wedge C_2) \wedge C_3 = C_1 \wedge (C_2 \wedge C_3)$ and $(C_1 \otimes C_2) \otimes C_3 = C_1 \otimes (C_2 \otimes C_3)$ (incremental design).
- For any two contracts refining two other contracts, $C'_1 \preccurlyeq C_1$ and $C'_2 \preccurlyeq C_2$, their composition refines the composition of the other contracts: $C'_1 \otimes C'_2 \preccurlyeq C_1 \otimes C_2$ (independent implementability). Thus, two contracts can be refined/implemented by different teams or organisations, and these implementations will in the end work together as specified by the original contracts and their composition.

Receptiveness/directedness. Contracts must be receptive: guarantees have to be *uncontrolled receptive* (u-receptive), meaning that uncontrolled ports (inports) may not be referred to in guarantees, and assumptions have to be *controlled receptive* (c-receptive), meaning that controlled ports (outports) may not be referred to in assumptions [24]. This ensures that

contracts describe systems in a combinational/Boolean way. Damm & al [56] require the same property for contracts, calling them “directed” because such contracts cannot form loops.[22]

5.2 Results of the case study

The application of contract algebra to the verification of the example system yielded the following main results:

- it was possible to manually transform those components of the system model necessary for the verification of a safety requirement to contracts;
- the selected safety requirement was succesfully (and manually) verified using the approach;
- contract formulas may tend to get large in the general case (see also Section 5.3);
- the validation of a general applicability of the approach to complex automotive systems and complex verification conditions would require tool support.

5.3 Composing the contracts of a function & its monitor

To further evaluate the application of contract algebra in system specification and verification, this section describes the use case of a simple function, its associated monitoring function and their composition.

Consider a vehicle speedometer consisting of a velocity calculator and a monitor that identifies outliers, transient errors. The calculator receives wheel revolutions and converts them into a speed measurement; the monitor is a type of sanity checker signalling whether consecutive velocity values are too far apart to be credible (Figure 14).

[22] Obviously, contracts cannot guarantee properties of uncontrolled ports. The necessity of requiring c-receptiveness is less obvious and can possibly be circumvented by specifying the whole state trajectory from the initial state, referring to inports only.

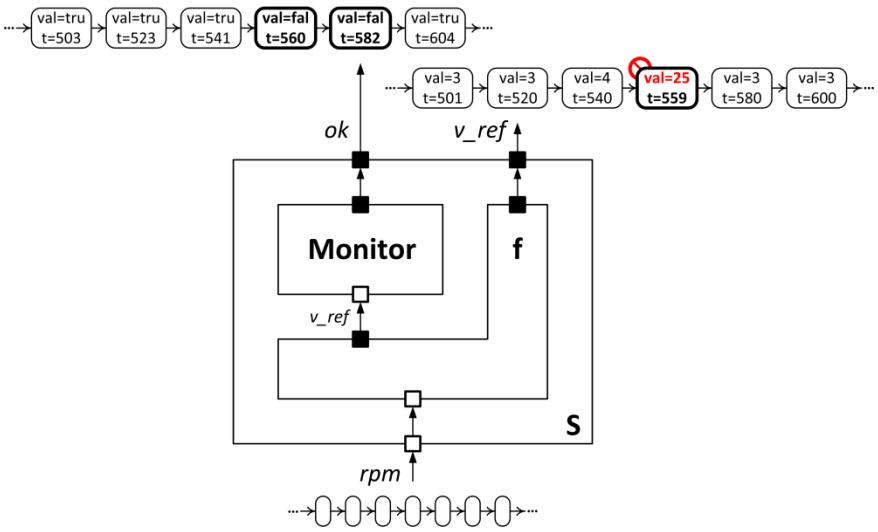

Figure 14. Speedometer consisting of a velocity calculator (*f*) and a sanity checker (*monitor*)

Table 1 shows the contracts of both function and monitor, with temporal logic expressions.

Component	***Assumption***	***Guarantee***
f	rpm occurs every 100µs with jitter ≤ 20µs	v_ref occurs every 20ms with jitter ≤ 2ms
Monitor	|v_ref.value – N[v_ref.value]| ≤ 10	N[ok == TRUE]
	|v_ref.value – N[v_ref.value]| > 10	N[ok == FALSE]
	v_ref occurs every 20ms with jitter ≤ 5ms	ok occurs every 20ms with jitter ≤ 5ms

Table 1. Contracts of function and monitor

The contract for the function itself states that, as long as it receives 10,000 values per second of the number of wheel revolutions per minute, with the specified jitter of 20µs max, it will produce 50 values per second of the vehicle's reference velocity, with a jitter not exceeding 2ms. The monitor in turn checks the absolute difference between consecutive velocity values and outputs *false* whenever the difference is higher than 10.

Comp	***Assumption***	***Guarantee (DNF)***
f ⊗ **Monitor**	rpm occurs every 100µs with jitter ≤ 20µs AND (v_ref occurs every 20ms with jitter ≤ 5ms OR | v_ref.*value* – N[v_ref.*value*]| > 10) OR !(*Guarantee* →)	!(rpm occurs every 100µs with jitter ≤ 20µs) AND !(v_ref occurs every 20ms with jitter ≤ 5ms) AND !(| v_ref.*value* – N[v_ref.*value*]| > 10) OR !(rpm occurs every 100µs with jitter ≤ 20µs) AND !(v_ref occurs every 20ms with jitter ≤ 5ms) AND N[ok == FALSE] OR

		!(rpm occurs every 100µs with jitter ≤ 20µs) AND ok occurs every 20ms with jitter ≤ 5ms AND !(| v_ref.*value* – N[v_ref.*value*]| > 10) OR !(rpm occurs every 100µs with jitter ≤ 20µs) AND ok occurs every 20ms with jitter ≤ 5ms AND N[ok == FALSE] OR v_ref occurs every 20ms with jitter ≤ 2ms AND !(v_ref occurs every 20ms with jitter ≤ 5ms) AND !(| v_ref.*value* – N[v_ref.*value*]| > 10) OR v_ref occurs every 20ms with jitter ≤ 2ms AND !(v_ref occurs every 20ms with jitter ≤ 5ms) AND N[ok == FALSE] OR v_ref occurs every 20ms with jitter ≤ 2ms AND ok occurs every 20ms with jitter ≤ 5ms AND !(| v_ref.*value* – N[v_ref.*value*]| > 10) OR v_ref occurs every 20ms with jitter ≤ 2ms AND ok occurs every 20ms with jitter ≤ 5ms AND N[ok == FALSE]

Table 2. Parallel composition of the contracts of function and monitor[23]

One example for the behaviour of the speedometer is depicted in Figure 14: both output sequences have a timestamp (*t*) and a data value (*val*), and the fourth data value of v_ref (val=25, t=559) of the regarded sequence triggers the monitor to indicate an outlier.[24] As in the case study (Section 5.2), component contracts are normalised, conjoined and composed to yield the overall contract (Table 2).

Component	***Assumption***	***Guarantee***
S	rpm occurs every 100µs with jitter ≤ 20µs	v_ref occurs every 20ms with jitter ≤ 2ms AND ok occurs every 20ms with jitter ≤ 5ms AND (|v_ref.value – N[v_ref.value]| ≤ 10 IMPLIES N[ok == TRUE]) AND (|v_ref.value – N[v_ref.value]| > 10 IMPLIES N[ok == FALSE])

Table 3. Higher-level contract for entire speedometer (not normalised)

[23] The assumption contains the negation of the guarantee, which is only indicated for conciseness.

[24] To keep the specification simple, the monitor indicates *false* in both directions: at the time the outlier occurs, and also when the value returns to its former level.

The resulting contract is relatively large and nonmonotonic, containing at the same time positive propositions and their negation. This shows the effects of assumption explosion reported by Kupferman & Vardi [129] (Section 3.8). It would indeed be more efficient to prove fulfilment of the higher-level contract (Table 3) by (an exceedingly simple case of) theorem proving. Theorem proving, however, has its own problems: it is unaffected by state space largeness but does not lend itself to the same degree of automatisation as model checking [90].

Mapping the stochastic approach of Delahaye & al (Section 3.7.2) to the speedometer example shows that the resulting lower bound for contract satisfaction probability is never higher with a monitor than for the function alone; this is undesirable and demonstrates the limitation of the stochastic approach.

6 Conclusion

It appears to be a popular myth (which this author believed, too, well into the research described by the present report) that compositional reasoning helps in coping with the state space largeness of complex systems. By itself, however, it does not. Modular reasoning with assume-guarantee contracts, in principle, is of the same complexity as monolithic techniques. As Lamport famously quipped, compositionality even “make[s] proofs harder” [135]. Nonetheless, for the assurance of component integration and reuse we depend on it.

As with monolithic reasoning, the only way to make state spaces tractable is abstraction. In single cases it might be possible, following a system’s structure, to find modular abstractions that reduce verification complexity. With learning algorithms such a reduction will be paid for by a comparable increase in verification time (Section 3.8).

The exemplary composition of contracts with a dedicated algebra has (tentatively) shown that this approach is, in principle, valid. Nevertheless, the composite contract tends to get incomprehensible, even with the most simple assumptions and just one hierarchy level. Resulting logical propositions are highly nonmonotonic, which impedes their efficient automatic processing and analysis. Still, these difficulties do not invalidate the basic necessity of conditional specifications for the virtual integration and reuse of software components. A dedicated tool support is desirable. Whether, and to which extent, the analysis of larger models and more complex properties or verification conditions will be feasible can, at present, not be concluded.

Further results of this project are:

- contract algebra turns out to be a less efficient approach to enabling virtual integration and component reuse than initially expected. It easily leads to “assumption explosion” [129], which puts into question its applicability to system models of realistic size;

- existing stochastic approaches based on contracts or contract algebra cannot be used to assess, for example, the reliability to expect from a component system in a meaningful way.

If they are not already available, component contracts have to be manually abstracted from the concrete system model (for the example system of the case study reported this was a Simulink/Stateflow model), or they have to be derived automatically from a superordinated system specification, by way of learning algorithms. In spite of numerous publications on contract algebra and its conceptual integration into the industry-oriented RCM approach, with the exception of anecdotal hints no reports about a successful practical application seem to exist.

The efficiency of modular reasoning could be improved based on the causal structure of a system model, by considering only those contracts, or partial contracts, that actually do interact with each other: Lomuscio & al [140] propose such a technique (Section 3.7.5). Model with contracts based on simple pre- and postconditions and invariants can be investigated using constraint logic programming (Section 3.3.1). In this context it would make sense to consider the development of a corresponding dedicated, extensible language for conditional, formal specifications of embedded component systems [190] [191].

Possible questions for future research:

- How far could the example system used in the case study be analysed with the aid of a tool?
- How can the mathematical foundations best be “hidden” behind a user interface for software engineers and developers?
- What should a semantic CS language look like?
- What is the optimal trade-off between specification expressiveness and analysis effort?

Abbreviations, acronyms & symbols

CA	Conditional assurance
CASE	Computer-aided software engineering
CBD	Component-based development
CEGAR	Counterexample-guided abstraction refinement
COTS	Commercial off-the shelf (software)
CS	Conditional specification
CTL	Computation tree logic
DNF	Disjunctive normal form
ES	Embedded system(s)
GPS	Global positioning system
HRCM	Heterogeneous rich component model
HW	Hardware
IA	Interface automaton/a
LOC	Logic of constraints
LTL	Linear temporal logic
MBD	Model-based development
MSC	Message sequence chart
OCL	Object constraint language
OO	Object-oriented, object orientation

PPP	Public-private partnership
RCM	Rich component model
RTE	Runtime environment
SW	Software
TIA	Timed interface automaton/a
TL	Temporal logic
UML	Unified modelling language
VIT	Virtual integration testing
$\rightarrow$	*Implies*
$\preccurlyeq$	*Refines/dominates*
$\vDash$	*Satisfies*
$\boldsymbol{A}$	*For all paths/futures …*
$\boldsymbol{E}$	*There exists a path/future where …*
$\boldsymbol{F\varphi}$	*In some future state, φ holds*
$\boldsymbol{G\varphi}$	*Globally, in the present state and all subsequent states, φ holds*
$\boldsymbol{N\varphi}$	*In the next state, φ holds*
$\boldsymbol{\psi R\varphi}$	*Release*
$\boldsymbol{\psi U\varphi}$	*Until*
$\boldsymbol{\psi W\varphi}$	*Weak until* (‘*unless’*)
$\boldsymbol{X\varphi}$	*In the next state, φ holds* (same as $N\varphi$)

Glossary

This glossary gives definitions and explanations of concepts from the field of formal verification.

Compositionality

See *modularity*.

Conditional assurance (conditional specification)

Coined by this report to subsume the multitude of terms used over the years by different people in various contexts to denote modular techniques for specification and reasoning: assume/guarantee, rely/guarantee, assumption/promise, assumption/commitment, assume/assert, contract-based design, Design by contract etc. Acronyms: CA and CS, respectively.

Counterexample-guided abstraction refinement

Program verification relies on abstraction techniques because an exact specification typically is too detailed for analysis because of the large state spaces involved.

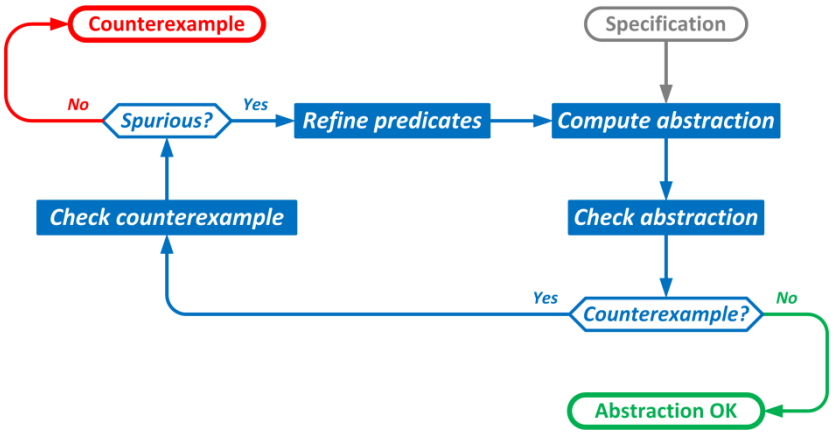

Figure 15. The CEGAR loop [127]

Finding an abstraction that is "just right" to prove or disprove a property is one of the main problems in program verification. Counterexample-guided

abstraction refinement (CEGAR) [49] is a method to automatically determine such an abstraction (Figure 15).

Formal safety vs operational safety

While operational safety is the property of the operation of a real-world, safety-critical system to remain below a specified limit risk, formal safety is a more abstract concept denoting a class of model properties applying, for example, to finite state automata: a typical formal safety property, expressed in LTL, is $G\neg\varphi$ (read "φ is false for all states of the model").

Independent implementability

The combination of (1) incremental design support, enabling the composition of component interfaces/specifications in any order, and (2) refinement as a sort of contravariant behavioural subtyping, meaning that a component specification A' is a refinement of specification A if and only if A' accepts at least the same set of environment behaviours as A and produces no other (but possibly less) behaviours than A. "Fulfilling these constraints ensures that components with compatible interfaces can be refined independently and still remain compatible […]." [8] (See also [180])

Model checking

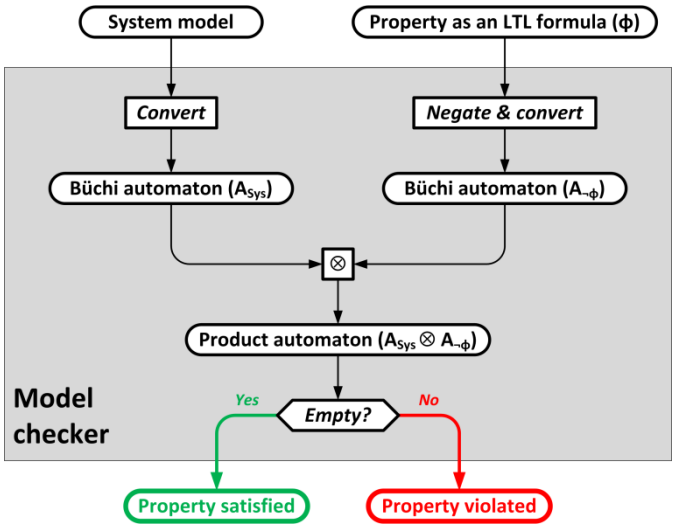

Figure 16. Model checking by emptiness of product automaton

"An important way to model check is to express desired properties […] using LTL operators and actually check if the model satisfies this property. One technique is to obtain a Büchi automaton that is "equivalent" to the model and one that is "equivalent" to the negation of the property. The intersection of the two non-deterministic Büchi automata is empty if the model satisfies the property." [104]

Modularity

"We speak of modularity if the interface abstraction of an architecture is the result of the composition of the interface abstractions of all its components." [38]

Predicate abstraction

Abstraction technique used in model checking to reduce the state space of a model, by partitioning the value range of variables into equivalence classes. Checking a property on the abstracted model is then expected to be easier than checking it on the original. Figure 17 shows an example with a system that has two variables, x and y:

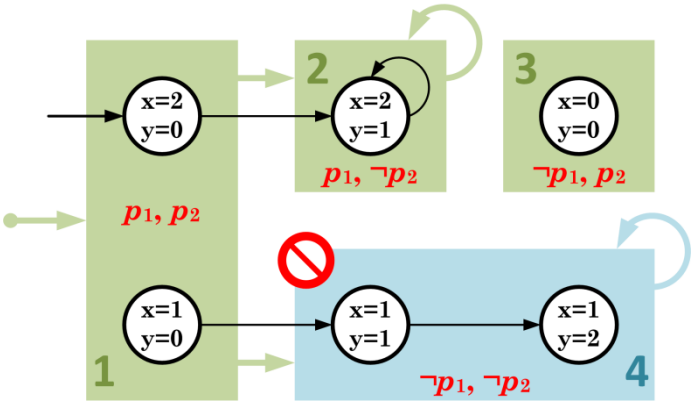

Figure 17. Predicate abstraction with spurious counterexample [127]

Let the predicates used for abstraction be p_1: $x > y$ and p_2: $y = 0$. Defining the abstract states (p_1, p_2), $(\neg p_1, p_2)$, $(p_1, \neg p_2)$ and $(\neg p_1, \neg p_2)$ reduces the original state space of six states to four abstract states with corresponding abstract transitions. Checking the property $p_1 \vee p_2$ on the abstraction gives the result that (abstract) states 1 and 2 satisfy the

property while state 4 does not (state 3 is not reachable even in the abstraction). The counterexample, however, is *spurious*: it is not valid for the original system because the two states that make up abstract state 4 are not reachable from the original start state. If all counterexamples should turn out to be spurious, which is checked on the original system, it can be concluded that the system satisfies the property that was checked on the abstraction.

Sequential vs parallel composition of processes/programs

The sequential or parallel composition of processes refers to the relative execution order of their operations. In the execution of a sequential composition of two processes *A* and *B*, all operations of *A* either precede or succeed all operations of *B*. "[Sequential] composition is therefore not well defined for […] programs that share control points." [185] In a parallel composition of *A* and *B*, their operations may be interleaved.

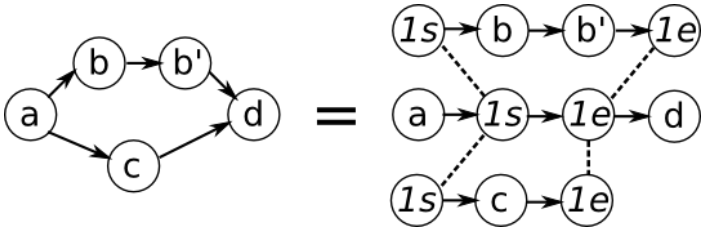

Figure 18. Transforming the sequential composition of parallel processes [36]

The sequential composition of parallel processes, however, can be transformed to an equivalent parallel composition of (sequential) processes with synchronisation points. Figure 18 shows such a transformation to the parallel composition of three processes with synchronisation, where *a*/*d* precede/succeed all of *b*, *b'* and *c*, and where the latter may be executed in any order, provided *b* comes before *b'*.

References

Abadi, M, L Lamport (1991). The existence of refinement mappings. *Theoretical computer science*, 82: 253-284.

[1] Abadi, M, L Lamport (1993). Composing specifications. *ACM Transactions on programming languages and systems*, 15(1): 73-132.

[2] Abadi, M, L Lamport (1995). Conjoining specifications. *ACM Transactions on programming languages and systems*, 17(3): 507-534.

[3] Adler, R, I Schäfer, T Schüle, E Vecchié (2007). From model-based design to formal verification of adaptive embedded systems. In: Butler, Hinchey, Larrondo-Petrie (eds), *ICFEM'07*. LNCS 4789: 76-95.

[4] Aichernig, B K, H Jifeng, Zh Liu, M Reed (2008). Integrating theories and techniques for program modelling, design and verification. In: Meyer, Woodcock (eds), *Verified software*. LNCS 4171: 291-300.

[5] Åkerholm, M (2008). Reusability of software components in the vehicular domain. Doctoral thesis, Mälardalens högskola.

[6] Alfaro, L de, Th A Henzinger (2001). Interface automata. ESEC/FSE'01.

[7] Alfaro, L de, Th A Henzinger (2005). Interface-based design. In: Broy, Grünbauer, Harel, Hoare (eds), *Engineering theories of software-intensive systems*. NATO science series: mathematics, physics and chemistry, 195: 83-104.

[8] Alfaro, L de, Th A Henzinger, M Stoelinga (2002). Timed interfaces. In: Sangiovanni-Vincentelli, Sifakis (eds), *EMSOFT 2002*. LNCS 2491: 108-122. DOI 10.1007/3-540-45828-X_9.

[9] Alfaro, L de, Th A Henzinger, R Jhala (2001). Compositional methods for probabilistic systems. In: Larsen, Nielsen (eds), *CONCUR 2001*. LNCS 2154: 351-365.

[10] Alur, R, Th A Henzinger (1992). Logics and models of real time: a survey. In: de Bakker, Huizing, de Roever, Rozenberg (eds), *Real-time: theory in practice* (REX workshop 1991 proceedings). LNCS 600: 74-106. DOI 10.1007/BFb0031988.

[11] Alur, R, Th A Henzinger (1999). Reactive modules. *Formal methods in system design*, 15(1): 7-48. DOI 10.1023/A:1008739929481.

[12]

America, P (1990). Designing an object-oriented programming language with behavioural subtyping. In: de Bakker, de Roever, Rozenberg (eds), *Foundations of object-oriented languages*. LNCS 489: 60-90. DOI 10.1007/BFb0019440.

[13] Andrade, L F, J L Fiadeiro (1999). Interconnecting objects via contracts. In: France, Rumpe (eds), *UML'99*. LNCS 1723: 566-583.

[14] Arbab, F (2005). Abstract behavior types: a foundation model for components and their composition. *Science of computer programming*, 55: 3-52.

[15] Ashcroft, E A (1975). Proving assertions about parallel programs. Journal of computer and system sciences, 10: 110-135.

[16] Bachmann, F, L Bass, Ch Buhman, S Comella-Dorda, Fr Long, J Robert, R Seacord, K Wallnau (2000). Volume II: Technical concepts of component-based software engineering, 2nd edition. Technical report CMU/SEI-2000-TR-008.

[17] Bate, I, R Hawkins, J McDermid (2003). A contract-based approach to designing safe systems. 8th Australian workshop on safety critical systems and software (SCS'03).

[18] Bate, I, T Kelly (2003). Architectural considerations in the certification of modular systems. *Reliability engineering & system safety*, 81(3): 303-324. DOI 10.1016/S0951-8320(03)00094-2.

[19] Bauer, S S, A David, R Hennicker, K G Larsen, A Legay, U Nyman, A Wąsowski (2012). Moving from specifications to contracts in component-based design. Technischer Bericht 1201, Ludwig-Maximilians-Universität München.

[20] Baumgart, A, E Böde, M Büker, W Damm, G Ehmen, T Gezgin, St Henkler, H Hungar, B Josko, M Oertel, Th Peikenkamp, Ph Reinkemeier, I Stierand, R Weber (2011). Architecture modeling. Technischer Bericht, OFFIS.

[21] Becker, St, R H Reussner, V Firus (2003). Specifying contractual use, protocols and quality attributes for software components. *1st International workshop on component engineering methodology*: 13-22.

[22] Benveniste, A, B Caillaud, A Ferrari, L Mangeruca, R Passerone, Chr Sofronis (2008). Multiple viewpoint contract-based specification and design. In: de Boer & al (eds), *FMCO 2007*. LNCS 5382: 200-225.

[23]

[24] Benveniste, A, B Caillaud, R Passerone (2007). A generic model of contracts for embedded systems. Rapport de recherche no 6214, Institut national de recherche en informatique et en automatique (INRIA). ISSN 0249-6399.

[25] Benveniste, A, J-B Raclet, B Caillaud, D Nickovic, R Passerone, A Sangiovanni-Vincentelli, T Henzinger, K Larsen (2011). Contracts for the design of embedded systems, part II: Theory. Submitted for publication.

[26] Benveniste, A, W Damm, A Sangiovanni-Vincentelli, D Nickovic, R Passerone, Ph Reinkemeier (2011). Contracts for the design of embedded systems, part I: Methodology and use cases. Submitted for publication.

[27] Berbers, Y, P Rigole, Y Vandewoude, St van Baelen (2005). CoConES: an approach for components and contracts in embedded systems. In: Atkinson & al (eds), *Component-based software development*. LNCS 3778: 209-231.

[28] Beugnard, A, J-M Jézéquel, N Plouzeau (2010). Contract aware components, 10 years after. In: Cámara, Canal, Salaün (eds), *Component and Service Interoperability* (International workshop on component and software interoperability, WCSI'10): 1-11. DOI 10.4204/EPTCS.37.1.

[29] Beugnard, A, J-M Jézéquel, N Plouzeau, D Watkins (1999). Making components contract aware. *IEEE Computer*, 32(11): 38-45.

[30] Biggerstaff, T (1994). The library scaling problem and the limits of concrete component reuse. *International conference on software reuse*: 102-110.

[31] Black, J A (2009). System safety as an emergent property in composite systems. Doctoral dissertation, Carnegie Mellon University.

[32] Blundell, C, D Giannakopoulou, C S Păsăreanu (2005). Assume-guarantee testing. *SAVCBS'05*: article no 1. DOI 10.1145/1123058.1123060.

[33] Böhme, S, M Moskal, W Schulte, B Wolff (2010). HOL-Boogie – an interactive prover-backend for the verifying C compiler. *Journal of automated reasoning*, 44(1-2): 111-144.

[34] Bouhadiba, T, Fl Maraninchi (2009). Contract-based coordination of hardware components for the development of embedded software. In: Field, Vasconcelos (eds), *COORDINATION'09*. LNCS 5521: 204-224.

[35] Bouyer, P (2009). Model-checking timed temporal logics. *Electronic notes in theoretical computer science* (ENTCS), 231: 323-341. DOI 10.1016/j.entcs.2009.02.044.

[36] Brown, N (2010). Transposing sequential and parallel composition. Blog posting, http://chplib.wordpress.com/2010/06/22/transposing-sequential-and-parallel-composition/ (as of 2012-08-07).

[37] Browne, M C, E M Clarke, O Grümberg (1988). Characterizing finite Kripke structures in propositional temporal logic. Theoretical computer science, 59(1-2): 115-131. DOI 10.1016/0304-3975(88)90098-9.

[38] Broy, M (2004). Time, abstraction, causality and modularity in interactive systems – extended abstract. *Electronic notes in theoretical computer science* (ENTCS), 108: 3-9. DOI 10.1016/j.entcs.2004.11.003.

[39] Bruin, H de (1999). A grey-box approach to component composition. *1st International symposium on generative and component-based software engineering* (GCSE'99).

[40] Bruin, H de, H van Vliet (2002). The future of component-based development is generation, not retrieval. *9th Annual IEEE International conference and workshops on engineering of computer-based systems*.

[41] Brunel, J-Y, M Di Natale, A Ferrari, P Giusto, L Lavagno (2004). SoftContract: an assertion-based software development process that enables design-by-contract. DATE'04. DOI 10.1109/DATE.2004.1268873.

[42] Černá, I, P Vařeková, B Zimmerová (2006). Component substitutability via equivalencies of component-interaction automata. *Electronic notes in theoretical computer science* (ENTCS), 182: 39-55.

[43] Chaki, S, E Clarke, D Giannakopoulou, C Păsăreanu (2004). Abstraction and assume-guarantee reasoning for automated software verification. RIACS technical report. http://ti.arc.nasa.gov/m/pub-archive/957h/0957%sou20%28Dimitra%29.pdf (as of 2012-10-28).

[44] Chaki, S, E Clarke, N Sharygina, N Sinha (2005). Dynamic component substitutability analysis. *Formal methods* (FM). LNCS 3582: 512-528.

[45] Chaki, S, E Clarke, N Sharygina, N Sinha (2008). Verification of evolving software via component substitutability analysis. *Formal methods in system design*, 32(1): 235-266. DOI 10.1007/s10703-008-0053-x.

[46] Chaki, S, E M Clarke, J Ouaknine, N Sharygina, N Sinha (2004). State/event-based software model checking. In: Boiten, Derrick, Smith (eds), *IFM 2004*. LNCS 2999: 128-147.

[47] Chakrabarti, A, L de Alfaro, Th A Henzinger, M Stoelinga (2003). Resource interfaces, *EMSOFT'03*. LNCS 2855: 117-133.

[48] Chen, X, H Hsieh, F Balarin, Y Watanabe (2004). Logic of constraints: a quantitative performance and functional constraint formalism. *IEEE Transactions on computer-aided design of integrated circuits and systems*, 23(8): 1243-1255. DOI 10.1109/TCAD.2004.831575.

[49] Clarke, E, O Grumberg, S Jha, Y Lu, H Veith (2000). Counterexample-guided abstraction refinement. In: Emerson, Sistla (eds), CAV'00. LNCS 1855: 154-169.

[50] Clarke, E M, D E Long, K L McMillan (1989). Compositional model checking. *Fourth annual symposium on logic in computer science* (LICS'89): 353-362.

[51] Cobleigh, J M, D Giannakopoulou, C S Păsăreanu (2003). Learning assumptions for compositional verification. *9th International conference on tools and algorithms for the construction and analysis of systems* (TACAS'03).

[52] Cobleigh, J M, G S Avrunin, L A Clarke (2006). Breaking up is hard to do: an investigation of decomposition for assume-guarantee reasoning. *International symposium on software testing and analysis* (ISSTA'06): 97-107.

[53] Cobleigh, J M, G S Avrunin, L A Clarke (2008). Breaking up is hard to do: an evaluation of automated assume-guarantee reasoning. ACM Transactions on software engineering and methodology, 17(2): article no 7.

[54] Conmy, Ph, M Nicholson, J McDermid (2003). Safety assurance contracts for integrated modular avionics. *8th Australian workshop on safety-critical systems and software*.

[55] Damm, W, A Votintseva, A Metzner, B Josko, Th Peikenkamp, E Böde (2005). Boosting reuse of embedded automotive applications through Rich Components. Proceedings of the *Foundation of interface technology workshop*.

[56] Damm, W, H Hungar, B Josko, Th Peikenkamp, I Stierand (2011). Using contract-based component specifications for virtual integration testing and architecture design. *DATE'11*: 1-6.

[57] Daskaya, I, M Huhn, St Milius (2011). Formal safety analysis in industrial practice. In: Salaün, Schätz (eds), *FMICS'11*. LNCS 6959: 68-84.

[58] Defour, O, J-M Jézequel, N Plouzeau (2004). Extra-functional contract support in components. In: Crnkovic & al (eds), *CBSE'04*. LNCS 3054: 217-232.

Delahaye, B (2010). Modular specification and compositional analysis of stochastic systems. Doctoral dissertation, Université de Rennes.

[59] Delahaye, B, B Caillaud (2008). A model for probabilistic reasoning on assume/guarantee contracts. Rapport de recherche no 6719, Institut

[60] national de recherche en informatique et en automatique (INRIA). ISSN 0249-6399.

Delahaye, B, B Caillaud, A Legay (2009). Compositional reasoning on

[61] (probabilistic) contracts. Rapport de recherche no 6970, Institut national de recherche en informatique et en automatique (INRIA). ISSN 0249-6399.

Delahaye, B, B Caillaud, A Legay (2011). Probabilistic contracts: a

[62] compositional reasoning methodology for the design of systems with stochastic and/or non-deterministic aspects. *Formal methods in system design*, 38(1): 1-32. DOI 10.1007/s10703-010-0107-8.

[63] DeRemer, F, H H Kron (1976). Programming-in-the-large versus programming-in-the-small. *IEEE Transactions on software engineering*, SE-2(2): 80-86.

[64] Dodds, M, X Feng, M Parkinson, V Vafeiadis (2009). Deny-guarantee reasoning. In: Castagna (ed), *ESOP 2009*. LNCS 5502: 363-377.

[65] Doyen, L, Th A Henzinger, B Jobstmann, T Petrov (2008). Interface theories

[66] with component reuse. *EMSOFT'08*: 79-88. DOI 10.1145/1450058.1450070.

Easwaran, A, I Lee, O Sokolsky (2008). Interface algebra for analysis of hierarchical real-time systems. *Foundations of interface technologies*

[67] *workshop* (FIT'08).

Elmqvist, J, S Nadjm-Tehrani, M Minea (2005). Safety interfaces for

[68] component-based systems. In: Winther, Gran, Dahll (eds), *SAFECOMP'05*. LNCS 3688: 246-260.

[69] Elmqvist, J (2007). Components, safety interfaces and compositional analysis. Doctoral dissertation no 1317, Linköpings universitet.

[70] Emerson, E A (1990). Temporal and modal logic. In: van Leeuwen (ed), *Handbook of theoretical computer science, volume B: formal models and*

[71] *semantics*: 995-1072.

Emmi, M, D Giannakopoulou, C S Păsăreanu (2008). Assume-guarantee verification for interface automata, *FM'08*. LNCS 5014: 116-131.

Feng, L, M Kwiatkowska, D Parker (2011). Automated learning of probabilistic Assumptions for compositional reasoning. In:

Giannakopoulou, Orejas (eds), *Fundamental approaches to software engineering* (FASE'11). LNCS 6603: 2-17.
Feng, X (2009). Local rely-guarantee reasoning. *POPL'09*: 315-327.
[72] Floyd, R W (1967). Assigning meaning to programs. *Mathematical aspects of computer science*, 19: 19-32.
[73] Frølund, S, J Koistinen (1998). Quality-of-service specification in distributed object systems. *Distributed systems engineering*, 5(4): 179-202.
[74] Furia, C A (2005). A Compositional world: a survey of recent works on
[75] compositionality in formal methods. Technical report 2005.22, Dipartimento di elettronica e informazione, Politecnico di Milano.
Gafni, V, A Benveniste, B Caillaud, S Graf, B Josko (2008). Contract
[76] specification language (CSL). Technical report D.2.5.4, v0.3, SPEEDS project (Speculative and exploratory design in systems engineering).
[77] Gamma, E, R Helm, R Johnson, J Vlissides (1994). Design patterns: elements of reusable object-oriented software. Boston: Addison-Wesley. ISBN 0-201-63361-2.
[78] Garlan, D, R Allen, J Ockerbloom (1995). Architectural mismatch: why reuse is so hard. *IEEE Software*, 12(6): 17-26.
[79] Garlan, D, R Allen, J Ockerbloom (2009). Architectural mismatch: why reuse
[80] is *still* so hard. *IEEE Software*, 26(4): 66-69.
Giannakopoulou, D, C S Păsăreanu, C Blundell (2008). Assume-guarantee testing for software components. *IET Software*, 2(6): 547-562. DOI
[81] 10.1049/iet-sen:20080012.
Glouche, Y, J-P Talpin, P Le Guernic, Th Gautier (2009). A module language
[82] for typing by contracts. In: Denney, Giannakopoulou, Păsăreanu (eds), *The first NASA formal methods symposium*: 81-90.
Glouche, Y, P Le Guernic, J-P Talpin, Th Gautier (2009). A Boolean algebra of contracts for assume-guarantee reasoning. In: Sun, Schätz (eds), *6th*
[83] *International workshop on formal aspects of component software* (FACS'09): 37-52.
Glouche, Y, P Le Guernic, J-P Talpin, Th Gautier (2010). A Boolean algebra of contracts for assume-guarantee reasoning. *Electronic notes in theoretical computer science* (ENTCS), 263: 111-127. DOI 10.1016/j.entcs.2010.05.007.

[84] Goguen, J A (1986). Reusing and interconnecting software components. *IEEE Computer*, 19(2): 16-28. ISSN 0018-9162. DOI 10.1109/MC.1986.1663146.

[85] Goguen, J A, R M Burstall (1980). CAT, a system for the structured elaboration of correct programs from structured specifications. Technical report CSL-118, SRI International.

[86] Gössler, Gr, D Le Metayer, J-B Raclet (2010). Causality analysis in contract violation. In: Roşu & al (eds), *RV'10*. LNCS 6418: 270-284.

[87] Gössler, Gr, J-B Raclet (2009). Modal contracts for component-based design. *7th IEEE International conference on software engineering and formal methods* (SEFM'09). DOI 10.1109/SEFM.2009.26.

[88] Gotzhein, R, M Kronenburg, Chr Peper (1996). Specifying and reasoning about generic real-time requirements – a case study. Report 15/96, Sonderforschungsbereich 501. Universität Kaiserslautern.

[89] Hailpern, B, H Ossher (1990). Extending Objects to support multiple interfaces and access control. *IEEE Transactions on software engineering* 16(11): 1247-1257. DOI 10.1109/32.60313.

[90] Halpern, J Y, M Y Vardi (1991). Model checking vs theorem proving: a manifesto. In: Lifschitz (ed), *Artificial intelligence and mathematical theory of computation (papers in honor of John McCarthy)*: 151-176.

[91] Harbird, L, A Galloway, R F Paige (2010). Towards a model-based refinement process for contractual state machines. *13th IEEE International symposium on object/component/service-oriented real-time distributed computing workshops*: 108-115. DOI 10.1109/ISORCW.2010.25.

[92] Hardung, B, Th Kölzow, A Krüger (2004). Reuse of software in distributed embedded automotive systems. *4th ACM International conference on embedded software*, EMSOFT'04: 203-210.

[93] Harel, D (1987). Statecharts: a visual formalism for complex systems. Science of computer programming, 8(3): 231-274. DOI 10.1016/0167-6423(87)90035-9.

[94] Helm, R, I M Holland, D Gangopadhyay (1990). Contracts: specifying behavioral compositions in object-oriented systems. ECOOP/OOPSLA'90: 169-180.

[95] Henzinger, Th A (2000). Masaccio: a formal model for embedded components. LNCS 1872: 549-563.

[96] Henzinger, Th A, M Minea, V Prabhu (2001). Assume-guarantee reasoning for hierarchical hybrid systems. In: Di Benedetto, Sangiovanni-Vincentelli (eds), *HSCC'01*. LNCS 2034: 275-290. DOI 10.1007/3-540-45351-2_24.

[97] Henzinger, Th A, Sl Matic (2006). An Interface algebra for real-time components. *IEEE Real-time and embedded technology and applications symposium* (RTAS'06): 253-266. DOI 10.1109/RTAS.2006.11.

[98] Hoare, C A R (1969). An axiomatic basis for computer programming. *Communications of the ACM*, 12(10): 576-583.

[99] Hoare, T (2003). The verifying compiler: a grand challenge for computing research. *Journal of the ACM*, 50(1): 63-69.

[100] Holland, I M (1992). Specifying reusable components using contracts. *ECOOP'92*. LNCS 615: 287-308.

[101] http://autosar.org/ (as of 2012-07-05).

[102] http://btc-es.de/index.php?lang=2&idcatside=5 (as of 2012-10-01).

[103] http://ecma-international.org/publications/standards/Ecma-367.htm (as of 2012-07-19).

[104] http://en.wikipedia.org/wiki/Linear_temporal_logic (as of 2012-08-23).

[105] http://en.wikipedia.org/wiki/Mealy_machine (as of 2012-10-20).

[106] http://en.wikipedia.org/wiki/Moore_machine (as of 2012-10-20).

[107] http://omg.org/corba/ (as of 2012-07-30).

[108] http://ptolemy.eecs.berkeley.edu/ (as of 2012-09-26).

[109] http://vlsi.colorado.edu/~vis/ (as of 2012-08-24).

[110] http://www.dspace.com/en/pub/home/products/sw/pcgs/targetli.cfm (as of 2012-03-02).

[111] http://www.prismmodelchecker.org/ (as of 2012-10-13).

[112] Hung, Ph Ng (2009). Assume-guarantee verification of evolving component-based software. Doctoral thesis, Japan advanced institute of science and technology (JAIST).

[113] Hung, Ph Ng, T Katayama (2008). Modular conformance testing and assume-guarantee verification of evolving component-based software. *15th Asia-Pacific software engineering conference*: 480-486. DOI 10.1109/APSEC.2008.51.

[114] Jahanian, F, A K Mok (1986). Safety analysis of timing properties in real-time systems. *IEEE Transactions on software engineering*, 12(9): 890-904.

[115] Jahanian, F, A K Mok, D A Stuart (1988). Formal specification of real-time systems. Technical report, University of Texas at Austin.

Jézéquel, J-M, B Meyer (1997). Design by contract: the lessons of Ariane. *IEEE Computer*, 30(2): 129-130.

[116] Jézéquel, J-M, O Defour, N Plouzeau (2004). An MDA approach to tame component based software development. In: de Boer & al (eds), FMCO'03.

[117] LNCS 3188: 260-275.

Jones, C B (1981). Development methods for computer programs including a notion of interference. Technical monograph PRG-25, Oxford university

[118] computing laboratory.

Jones, C B (1983). Tentative steps toward a development method for

[119] interfering programs. *ACM Transactions on programming languages and systems*, 5(4): 596-619.

Jonge, M de (2003). To reuse or be reused – techniques for component

[120] composition and construction. Doctoral thesis, Universiteit van Amsterdam.

[121] Jonsson, B (1994). Compositional specification and verification of distributed systems. *ACM Transactions on programming languages and systems*, 16(2): 259-303.

[122] Jonsson, B, Y-K Tsay (1996). Assumption/guarantee specifications in linear-

[123] time temporal logic. *Theoretical computer science*, 167: 47-72.

Kapoor, K, K Lodaya, U S Reddy (2011). Fine-grained concurrency with separation logic. Journal of philosophical logic, 40(5): 583-632. DOI

[124] 10.1007/s10992-011-9195-1.

Kelly, T (2006). Using software architecture techniques to support the

[125] modular certification of safety-critical systems. *11th Australian workshop on safety-related programmable systems* (SCS'06): 53-65.

[126] Kelly, T (1998). Arguing safety – a systematic approach to managing safety cases. PhD thesis, University of York.

[127] Kindler, E, T Vesper (1998). ESTL: a temporal logic for events and states. In: Desel, Silva (eds), *ICATPN'98*. LNCS 1420: 365-384.

[128] Kroening, D (2012). Predicate abstraction: a tutorial. Presentation slides, Oxford university. http://fm.csl.sri.com/SSFT12/predabs-SSFT12.pdf (as of 2012-10-22)

Krueger, Ch W (1992). Software reuse. *ACM Computing surveys*, 24(2): 131-183.

[129] Kupferman, O, M Y Vardi (1998). Modular model checking. In: de Roever, Langmaack, Pnueli (eds), *Compositionality: the significant difference* (COMPOS'97). LNCS 1536: 381-401. DOI 10.1007/3-540-49213-5_14.

[130] Kupferman, O, N Piterman, M Y Vardi (2009). From liveness to promptness. *Formal methods in system design*, 34(2): 83-103. DOI 10.1007/s10703-009-0067-z.

[131] Kwiatkowska, M, G Norman, D Parker, H Qu (2010). Assume-guarantee verification for probabilistic systems. In: Esparza, Majumdar (eds), *TACAS'10*. LNCS 6015: 23-37.

[132] Lamport, L (1977). Proving the correctness of multiprocess programs. *IEEE Transactions on software engineering*, SE-3(2): 125-143.

[133] Lamport, L (1983). What good is temporal logic? In: Mason (ed), *IFIP'83*: 657-668.

[134] Lamport, L (1994). The temporal logic of actions. *ACM Transactions on programming languages and systems*, 16(3): 872-923.

[135] Lamport, L (1998). Composition: a way to make proofs harder. In: de Roever, Langmaack, Pnueli (eds), Compositionality: the significant difference (COMPOS'97). LNCS 1536: 402-423. DOI 10.1007/3-540-49213-5_15.

[136] Larsen, K G (1990). Modal specifications. In: Sifakis (ed), Automatic verification methods for finite state systems. LNCS 407: 232-246. DOI 10.1007/3-540-52148-8_19.

[137] Larsen, K G, U Nyman, A Wąsowski (2006). Interface input/output automata. In: Misra, Nipkow, Sekerinski (eds), *FM'06*. LNCS 4085: 82-97.

[138] Lee, E A, Y Xiong (2002). Behavioral types for component-based design. Technical memorandum UCB/ERL M02/29. University of California, Berkeley.

[139] Liskov, B H, J M Wing (1994). A behavioral notion of subtyping. ACM Transactions on programming languages and systems, 16(6): 1811-1841.

[140] Lomuscio, A, B Strulo, N Walker, P Wu (2010). Assume-guarantee reasoning with local specifications. In: Dong, Zhu (eds), *ICFEM'10*. LNCS 6447: 204-219.

[141] Lynch, N (1988). I/O automata: a model for discrete event systems. *Annual conference on information sciences and systems*: 29-38.

Maier, P (2003). Compositional circular assume-guarantee rules cannot be sound and complete. Technischer Bericht MPI-I-2003-2-001, Max-Planck-Institut für Informatik.

[142] Maraninchi, Fl, L Morel (2004). Logical-time contracts for reactive embedded systems. *EUROMICRO'04*: 48-55.

[143] Maydl, W, L Grunske (2005). Behavioral types for embedded software – a survey. In: Atkinson & al (eds), *Component-based software development*.

[144] LNCS 3778: 82-106.

McIlroy, M D (1969). 'Mass-produced' software components. In: Naur,

[145] Randell (eds), *Software engineering: report on a conference sponsored by the NATO science committee*: 138-155.

McMillan, K L (1999). Circular compositional reasoning about liveness. In:

[146] Pierre, Kropf (eds), CHARME'99. LNCS 1703: 342-346.

[147] Meyer, B (1991). Design by contract. In: Mandrioli, Meyer (eds), *Advances in object-oriented software engineering*: 1-50. Prentice-Hall.

[148] Meyer, B (1992). Applying "Design by contract". *IEEE Computer*, October 1992: 40-51.

[149] Meyer, B (2000). Contracts for components.

http://archive.eiffel.com/doc/manuals/technology/bmarticles/sd/contracts.html (as of 2012-07-04).

[150] Meyer, B (2003). A framework for proving contract-equipped classes. In:

[151] Börger, Gargantini, Riccobene (eds), *Abstract state machines* (ASM'03).

Misra, J, K M Chandy (1981). Proofs of networks of processes. *IEEE*

[152] *Transactions on software engineering*, SE-7(4): 417-426.

Nain, S, M Y Vardi (2007). Branching vs linear time: semantical perspective.

[153] In: Namjoshi & al (eds), fault *ATVA'07*. LNCS 4762: 19-34.

[154] Necula, G C (1997). Proof-carrying code. *POPL'97*: 106-119.

[155] Necula, G C, P Lee (1998). The design and implementation of a certifying compiler. *SIGPLAN'98*: 333-344.

[156] Nguyen, V (1985). The incompleteness of Misra and Chandy's proof systems. *Information processing letters*, 21: 93-96.

[157] Ostroff, J S (1999). Composition and refinement of discrete real-time systems. *ACM Transactions on software engineering and methodology*. 8(1): 1-48.

Ouaknine, J, J Worrell (2008). Some recent results in metric temporal logic. In: Cassez, Jard (eds), *FORMATS'08*. LNCS 5215: 1-13.

[158] Ouederni, M, Gw Salaün, E Pimentel (2010). Quantifying service compatibility: a step beyond the Boolean approaches. In: Maglio & al (eds), *ICSOC'10*. LNCS 6470: 619-626.

[159] Ouederni, M, Gw Salaün (2010). Tau be or not tau be? A perspective on service compatibility and substitutability. In: Cámara, Canal, Salaün (eds), *Component and service interoperability* (WCSI'10): 57-70. DOI 10.4204/EPTCS.37.5.

[160] Owicki, S, D Gries (1976). An axiomatic proof technique for parallel programs I. *Acta informatica*, 6(4): 319-340. DOI 10.1007/BF00268134.

[161] Peng, H, S Tahar (1998). A survey on compositional verification. Technical report, Concordia university. http://hvg.ece.concordia.ca/publications/technical-reports/ (as of 2012-08-08).

[162] Pnueli, A (1979). The temporal semantics of concurrent programs. In: Kahn (ed), *Semantics of concurrent computation*. LNCS 70: 1-20. DOI 10.1007/BFb0022460.

[163] Pnueli, A, M Siegel, E Singerman (1998). Translation validation. *Tools and algorithms for the construction and analysis of systems*. LNCS 1384: 151-166.

[164] Quinton, S, S Graf, R Passerone (2010). Contract-based reasoning for component systems with complex interactions. Verimag research report no TR-2010-12.

[165] Raclet, J-B, E Badouel, A Benveniste, B Caillaud, A Legay, R Passerone (2009). Modal interfaces: unifying interface automata and modal specifications. *EMSOFT'09*: 87-96.

[166] Raclet, J-B, E Badouel, A Benveniste, B Caillaud, R Passerone (2009). Why are modalities good for interface theories? *9th International conference on application of concurrency to system design*: 119-127.

[167] Raj, R K, H M Levy (1989). A compositional model for software reuse. The computer journal, 32(4): 312-322.

[168] Reussner, R H (2001). Parametrisierte Verträge zur Protokolladaption bei Software-Komponenten. Doctoral dissertation, Technische Hochschule Karlsruhe.

[169] Reussner, R H, H W Schmidt, I H Poernomo (2003). Reliability prediction for component-based software architectures. *The journal of systems and software*, 66: 241-252.

[170] Reussner, R H, I H Poernomo, H W Schmidt (2003). Reasoning about software architectures with contractually specified components. In: Cechich & al (eds), *Component-based software quality*. LNCS 2693: 287-325.
Roever, W-P de (1997). The need for compositional proof systems: a survey. *COMPOS'97*. LNCS 1536: 1-22.
[171] Schmidt, D C. Why software reuse has failed and how to make it work for you. http://www.cs.wustl.edu/~schmidt/reuse-lessons.html (as of 2012-08-15).
[172] Schroeder, A, S S Bauer, M Wirsing (2011). A contract-based approach to adaptivity. *The journal of logic and algebraic programming*, 80: 180-193. DOI 10.1016/j.jlap.2010.09.001.
[173] Shankar, N (1993). A lazy approach to compositional verification. Technical report CSL-93-08, Computer science laboratory, SRI International.
[174] Shankar, N (1998). Lazy compositional verification. In: de Roever, Langmaack, Pnueli (eds), *COMPOS'97*. LNCS 1536: 541-564.
[175] Sinha, N (2007). Automated compositional analysis for checking component substitutability. Doctoral dissertation, Carnegie Mellon university.
[176] Stark, E W (1985). A proof technique for rely/guarantee properties. In: Maheshwari (ed), *Foundations of software technology and theoretical computer science*. LNCS 206: 369-391.
[177] Szyperski, C (2002). *Component software: beyond object-oriented programming*, 2nd edition. Boston: Addison-Wesley.
[178] Thane, H, A Wall (2000). Formal and probabilistic arguments for reuse and reverification of components in safety-critical real-time systems. Technical report MDH-MRTC-56/2000-1-SE, Mälardalen real-time research centre.
[179] Thiele, L, E Wandeler, N Stoimenov (2006). Real-time interfaces for composing real-time systems. EMSOFT'06: 34-43.
[180] Törngren, M, D Chen, I Crnkovic (2005). Component-based vs model-based development: a comparison in the context of vehicular embedded systems. 31st EUROMICRO Conference on software engineering and advanced applications (EUROMICRO-SEAA'05).
[181] Tsay, Y-K (2000). Compositional verification in linear-time temporal logic (extended abstract). In: Tiuryn (ed), *FOSSACS'00*. LNCS 1784: 344-358.

US Department of transportation, Federal aviation administration (2004). Reusable software components. Advisory circular AC 20-148.

[183] Vafeiadis, V, M Parkinson (2007). A marriage of rely/guarantee and separation logic. In: Caires, Vasconcelos (eds), *CONCUR 2007*. LNCS 4703: 256-271.

[184] Veanes, M, C Campbell, W Schulte (2007). Parallel and serial composition of model programs. Microsoft research technical report MSR-TR-2007-22.

[185] Wallnau, K C (2003). Volume III: a technology for predictable assembly from certifiable components. Technical Report CMU/SEI-2003-TR-009, Carnegie Mellon university, Software engineering institute.

[186] Wallnau, K C (2010). Predictability by construction: working the architecture/program seam. Doctoral thesis, Mälardalens högskola.

[187] Xu, D N, G Gössler, A Girault (2010). Probabilistic contracts for component-based design. In: Bouajjani, Chin (eds), *ATVA 2010*. LNCS 6252: 325-340.

[188] Xu, Q, W-P de Roever, H Jifeng (1997). The rely-guarantee method for verifying shared variable concurrent progams. *Formal aspects of computing*, 9(2): 149-174.

[189] Zschaler, St (2004). Towards a semantic framework for non-functional specifications of component-based systems. EUROMICRO'04: 92-99.

[190] Zschaler, St (2010). 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.

[191]

Impressum

Herausgeber	FAT Forschungsvereinigung Automobiltechnik e.V. Behrenstraße 35 10117 Berlin Telefon +49 30 897842-0 Fax +49 30 897842-600 www.vda-fat.de
ISSN	2192-7863

VDA | Verband der Automobilindustrie

FAT | Forschungsvereinigung Automobiltechnik

Behrenstraße 35
10117 Berlin
www.vda.de
www.vda-fat.de