Verification of Component Architectures using Mode-Based Contracts

Stefan Kugele*, Diego Marmsoler*, Núria Mata†, and Kai Werther†
*Fakultät für Informatik, Technische Universität München, Germany
Email: {stefan.kugele, diego.marmsoler}@tum.de
†ETAS GmbH, Stuttgart, Germany
Email: {Nuria.Mata, Kai.Werther}@etas.com

***Abstract*—We consider the problem of achieving a required level of confidence about safety-critical systems consisting of interacting components. Especially, we address restrictions in traditional A/G reasoning techniques which may cause false positives in contract compatibility analyses. Therefore, we introduce interface assertions, i. e., predicate logical formulae over the components' interfaces. We show how to compute interface assertions for architecture configurations based on the interface assertions of the corresponding components and show *soundness* and *relative completeness* of the method. Moreover, we introduce *mode-based contracts*, which—as a special kind of interface assertions—consist of dedicated *assume* and *guarantee* pairs. They provide a methodological guidance for developers and facilitate contract specification in contrast to e.g. traditional A/G reasoning. For this concept, we provide algorithms to check *under-specification*, *over-specification*, and the *fulfillment* of specifications. We also sketch how the checks can be operationalized using SMT solvers. Finally, an example demonstrates the approach.**

I. Introduction

During the last decades, more and more software-controlled functions were introduced in embedded systems. Many of them are characterized as safety-critical such as those found in the avionic, automotive, railway, and industry automation domains. In the future, the number of those functions [1] as well as the complexity of involved hardware topologies will grow further—especially in emerging cyber-physical systems and the Industry 4.0 sector [2]. Thus, engineers have to grasp the enormous complexity involved in the development of such systems which will dominate their daily work. New ways of system design have to be applied: (i) Model-driven development helps to reduce the complexity apparent to the developer and (ii) the introduction of different levels of abstraction facilitates to structure the model. In component-based development, engineers heavily reuse already developed and verified library components. This reduces development time and costs. Each of those components viewed in isolation fulfills its specification, however, one cannot be sure about the situation when a component is reused in a new environment. Rigor modeling approaches on their own will not reach a sufficient and also required level of confidence about the product under development. An emphasis on early development stages (keyword "front-loading") and rigor (i. e., formal and mathematical provable) specifications of requirements are considered as indispensable (e. g. ISO 26262 [3] part 6, §6.4.8 recommends to *formally* verify software safety requirements beginning with ASIL B).

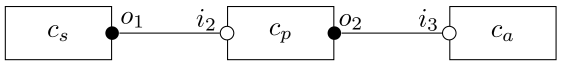

Figure 1. Example Architecture Configuration.

Design-by-Contract [4] and related techniques (A/G reasoning) aim at detecting non-obvious deficiencies and possible flaws in composed component architectures. However, sometimes traditional, pure assume/guarantee (A/G) reasoning is not well-suited and may lead to false positives in contract compatibility analyses as shown by the following introductory example.

A. Example: False Positives in Contract Analyses

Assume we have a simple architecture configuration given in Figure 1 consisting of the three components c_s (sensor), c_p (processing), and c_a (actuator). We say that component c_a is product-specific and c_p is reused in all product lines. The component c_s is in general also product-independent but can be parametrized during the calibration process.

Moreover assume, that the value of o_1 is either on or off: this is the guarantee of c_s. The component c_p assumes that i_2 is either on or off and guarantees that o_2 is either high or mid (e. g. for mid-size engines in the standard equipment). Lastly, c_a assumes that i_3 is mid. Now, we want to check whether component c_a is under-specified within this architecture configuration using A/G reasoning. In the given concrete architecture, c_s is parametrized such that it always provides the value off. You can think of a switch regulating the driving dynamics (e. g. sport mode is activated or not) available as optional equipment, but always set to off in the standard equipment, as in our case. We can rewrite it in A/G style as:

	Assumption	$\Longrightarrow$	*Guarantee*
c_s :	true	$\Longrightarrow$	$o_1 = \mathsf{off}$
c_p :	$(i_2 = \mathsf{on} \lor i_2 = \mathsf{off})$	$\Longrightarrow$	$(o_2 = \mathsf{mid} \lor o_2 = \mathsf{high})$
c_a :	$(i_3 = \mathsf{mid})$	$\Longrightarrow$	true

with additional predicates derived from the concrete parametrization and architecture:

$$\begin{aligned} (o_1, i_2): &\qquad i_2 := o_1 \\ (o_2, i_3): &\qquad i_3 := o_2 \end{aligned}$$

The component c_a is underspecified iff there is an assignment to the ports o_1, i_2, o_2, and i_3 such that i_3 is assigned a value different to mid. Of course, it is possible that c_2 provides at o_2 the value high. However, this is a *spurious counterexample* because the A/G pair $(i_2 = \mathsf{on} \vee i_2 = \mathsf{off}) \implies (o_2 = \mathsf{mid} \vee o_2 = \mathsf{high})$ is too coarse. Therefore, simple A/G reasoning leads to the assumption of under-specification. However, if the A/G pair of c_p was split into c_p' and c_p''

	Assumption $\implies$ *Guarantee*
c_p' :	$(i_2 = \mathsf{on}) \implies (o_2 = \mathsf{high})$
c_p'' :	$(i_2 = \mathsf{off}) \implies (o_2 = \mathsf{mid})$

c_a would not have been classified as underspecified. Only an *explicit* distinction in which case or system *mode* the port o_2 provides mid or high enables a meaningful analysis result. From this observation, we present *mode-based contracts* in Section III-C to address these situations.

B. Mode-Based Contracts

To address this problem, we describe an approach to compute the composition of interface assertions for a system consisting of many connected components. Related approaches—see below—mostly aim at verifying components' contracts. This can be done using common verification techniques ranging from lightweight testing methods to heavyweight model checking, theorem proving, or static analysis techniques. In contrary, the aim of this work is to use a formal component model that is enriched with specifications—interface assertions and (mode-based) contracts—to detect possible problems when already verified components are connected together in component networks. In particular, we investigate the question of interface assertion composition and verification of design properties leading to potential errors, namely *over-* and *under-specification* and the non-fulfillment of specifications. We incrementally extend the formalism and likewise discuss the expanded verification approaches in Section III.

We emphasize the division into the two parts (i) *component verification* and (ii) *architecture verification* due to the following reasons:

(i) Component contracts can be used to automatically generate test cases.
(ii) Already verified components can be reused facilitating the concept of a "*verified component library*".
(iii) Architecture verification is a means to provide support during system integration, where in fact most of the problems during system development occur.
(iv) The detection of under- and over-specified components within an architecture are indicators for architectural *flaws* and *smells*, respectively.
(v) Contracts can be used for instance to automatically generate watchdog components within an architecture that continuously monitor the system during runtime.

C. Related Work

Related work is classified into three major areas: *contract-based design*, *compositional A/G reasoning*, and *multi-contract components*.

1) Contract-Based Design: In contract-based design components are supplied by formal contracts in form of a specification formulated over their interfaces. Later on, these contracts can be used to verify whether a component fulfills its contract or not. Approaches in this area are sometimes also classified as component theories (in contrast to interface theories) [5] and do not explicitly describe a component's environment.

The basic ideas of *contract-based design* go back to Meyer [4], [6] with the EIFFEL [7] language. Other widely used approaches exist such as OCRA [8], JML [9] (and its several tools from external providers [10]), and SPEC# [11] which is an extension of the C# programming language.

While all these approaches provide an important base for contract-based design, their major focus lies on the verification of contracts, i. e., verifying that a component implementation indeed fulfills a certain contract specification. Our work, on the other hand, focuses on the compositional aspect, i. e., assuming a component fulfills its interface assertion, what can we say about the composition of them.

2) Compositional Assume/Guarantee Reasoning: Compositional A/G reasoning takes the idea of contracts one step further. A contract is split into two separate specifications, an *assumption* and a *guarantee*. Thus, they can be classified also as interface theories according to de Alfaro and Henzinger [5] and capture explicit information about a component's environment.

This additional information can then be used to perform more sophisticated analyses such as contract compatibility. That is, to check whether the guarantee of one component leads to the assumption of another connected component. Moreover, the verification of an architecture can be separated into two steps: First, components itself are verified by proving that a component fulfills its guarantee, given that the corresponding assumption holds. Second, component compatibility can be checked.

One early example of work in this area comes from Emmi et al. [12] who provide an approach for the modular verification of interface automata compatibility. Another example is the work of Damm et al. [13] in which the authors describe an approach where they use contracts in combination with rich components to perform virtual integration testing and to check a refinement relation between contracts. Moreover, Broy [14] builds upon the FOCUS theory and relates interface assertions to contracts. Finally, Chilton [15] provides an approach to A/G reasoning for safety properties. In this work, assumptions and guarantees are formulated as sets of traces.

More recently the relationship between simple contracts and A/G contracts were studied. For example, Nuzzo et al. [16]

investigate the relationship between interface theories and contract theories, their commonalities, and differences.

While all these works contribute to a better understanding of contracts, they all discuss contracts in terms of a single A/G pair. With our work we build on these fundamental approaches in that we investigate how the insights gained from them can be mapped to contracts based on multiple A/G pairs.

3) Multiple Contracts: More recently, the idea of multiple contracts emerged. It is based on the fact that a component may be deployed to different contexts which is why the need for multiple A/G pairs emerges.

To the best of our knowledge, there exists only one work which explicitly considers multiple A/G pairs for the specification of contracts which is Reussner's [17] work on *parametric contracts*. This work recognizes the need for multiple contracts and introduces the notion of parametric contracts to assign multiple contracts to a component.

However, only little is known about how these additional information can be used for more sophisticated analyses. Thus, with this work we want to address this point and investigate possible analysis techniques for those kind of contracts. Thereby we come up with the notion of under- and over-specification of contracts.

D. Contributions

This work provides the following contributions: (i) A formal component model, (ii) the notion of interface assertions and their composition, (iii) the notion of contracts and their extension to mode-based contracts, (iv) detection of *over-specification*, *under-specification*, and specification *fulfillment*, and (v) an example to demonstrate the approach.

E. Outline

Section II explains the notation used in this work and introduces a formal component model. In Section III, we present a specification formalism called *interface assertions*, which we extend step by step into *(mode-based) contracts*. Next, we discuss in Section IV analysis techniques applicable for each specification formalism. We present an example taken from the automotive domain in Section V and discuss the results. Finally, Sections VI and VII discuss and conclude this work and point out future research directions.

II. Foundations

This section provides the foundations of the presented model by introducing several concepts. We introduce the notion of components, ports, channels, and architecture configurations.

A. Ports and Components

In our model we assume that a system's functionality is specified by a set of interconnected components. Each component $c \in \texttt{COMPONENT}$ has a syntactic interface consisting of typed ports that allow interaction with its environment. Moreover, a component's semantic interface specifies its behavior.

In the following, we assume the existence of sets PORT and TYPE containing all ports and all types, respectively. A port can be simply thought of a named entity which can be used to exchange elements of a certain type. A type, on the other hand, is a representative for a certain set of elements. A simple type, for example, might be int containing all integers or string containing all character sequences, respectively. However, there might be also more complex types, such as int^* containing all sequences over int.

We further distinguish between *input* and *output* ports: IPORT and OPORT, respectively. Moreover, we assume $\texttt{IPORT} \cup \texttt{OPORT} = \texttt{PORT}$ and $\texttt{IPORT} \cap \texttt{OPORT} = \emptyset$.

To identify which elements can be exchanged through a certain port, we define a mapping which assigns a type to each port:

$$\text{type} : \texttt{PORT} \to \texttt{TYPE}. \tag{1}$$

That is, we assign a type to each port (e. g. int, bool, double, etc.). An instantiation, e. g. for AUTOSAR, needs to define the concrete types. We say that a *valuation* for a set of ports $P \subseteq \texttt{PORT}$ is a map $\mu : P \to \bigcup_{t \in \texttt{TYPE}} t$ which assigns a value of corresponding type to each port. We require $\mu(p) \in \text{type}(p)$ for all $p \in P$. The set of all valuations of $P \subseteq \texttt{PORT}$ is denoted by

$$\overline{P} \quad \overset{\text{def}}{=} \quad \left\{ \mu \colon P \to \bigcup_{t \in \texttt{TYPE}} t \mid \forall p \in P \colon \mu(p) \in \text{type}(p) \right\}. \tag{2}$$

For a better presentation, we also assume ports to be labeled with strings by an injective function

$$\ell : \texttt{PORT} \rightarrowtail \texttt{STRING} \tag{3}$$

where STRING is the set of all possible finite words. The only way to interact with components is via their interfaces.

Definition 1 (Interface). *An* interface *is a pair* (I, O) *where* $I \subseteq \texttt{IPORT}$ *and* $O \subseteq \texttt{OPORT}$.

A component consists of an interface and a behavior, i. e., a mapping from valuations of input ports to valuations of output ports.

Definition 2 (Component). *A* component c *is a pair* $((I, O), f)$ *where* (I, O) *is the component's interface and* $f : \overline{I} \to \mathcal{P}\left(\overline{O}\right)$ *its behavior function.*

Note that a component assigns a set of possible output port valuations to each input port valuation. This is due to the fact that a component's behavior may be non-deterministic in the sense that a concrete input may lead to different outputs. Moreover, note that our notion of component is rather abstract. We only require that a component's behavior can be represented as some kind of set-valued function. This is a deliberate decision which allows the corresponding theory to be applied to various contexts by providing a concrete notion of type:

- *Stateless* component models can be directly modeled using a simple notion of type such as int, double, string, etc.

- *Stateful* components can be modeled using sequences of simple types such as int^*, double^*, string^*, etc. to encode the full input history.
- *Service-oriented systems* can be modeled using services (e. g., $\mathsf{add}, \mathsf{mult}$) as types such as $\mathsf{add}(\mathsf{int}\ x, \mathsf{int}\ y)$, $\mathsf{mult}(\mathsf{int}\ x, \mathsf{int}\ y)$, etc.

By concertizing the notion of type, our results can be transferred to these kinds of system as well. For more details about the encoding we refer to [18] where an encoding of service-oriented systems is given for our model.

B. Architecture Configuration

An architecture configuration consists of a set of components, and a set of channels connecting the components' ports. Thus, an architecture configuration is modeled as a pair of a set of components and a set of channels describing the components' interactions. Note that an architecture can again be considered as a component which allows hierarchical construction of components by simply hiding the internal ports and channels.

Definition 3 (Architecture Configuration)**.** *An* architecture configuration *is a pair (C, A) where $C \subseteq \mathtt{COMPONENT}$ is a set of contained components and*
$A \subseteq \left(\bigcup_{((I,O),f)\in C} O \times \bigcup_{((I,O),f)\in C} I\right)$ *is the set of all channels such that the following constraints hold:*

(i) Different components do not share any ports, formally:

$$\forall\ ((I_1, O_1), f_1), ((I_2, O_2), f_2) \in C\colon \\ (I_1 = I_2 \wedge O_1 = O_2 \wedge f_1 = f_2) \vee \\ ((I_1 \cup O_1) \cap (I_2 \cup O_2) = \emptyset)\,. \quad (4)$$

(ii) Connected ports are compatible, i. e., $\forall\ (p_o, p_i) \in A\colon \mathrm{type}(p_o) \preceq \mathrm{type}(p_i)$, where $\preceq$ denotes the subtype relation.

(iii) To exclude undesired nondeterministic behavior, we prohibit that two channels are connected to the same target port:

$$\forall\ (o_1, i_1), (o_2, i_2) \in A\colon i_1 = i_2 \implies o_1 = o_2. \quad (5)$$

Convention. *For an architecture configuration $\Gamma = (C, A)$ we use the following abbreviations: We write $\mathsf{input}(\Gamma) = \bigcup_{((I,O),f)\in C} I$ and $\mathsf{output}(\Gamma) = \bigcup_{((I,O),f)\in C} O$ to denote input and output ports, respectively. All ports are referred to as $\mathsf{ports}(\Gamma) = \mathsf{input}(\Gamma) \cup \mathsf{output}(\Gamma)$. We distinguish between* connected *and* open *(input and output) ports. Connected:* $\mathsf{input}^{\mathsf{con}}(\Gamma) = \{p_i \mid \exists p_o\colon (p_o, p_i) \in A\}$, $\mathsf{output}^{\mathsf{con}}(\Gamma) = \{p_o \mid \exists p_i\colon (p_o, p_i) \in A\}$, *and* $\mathsf{ports}^{\mathsf{con}}(\Gamma) = \mathsf{input}^{\mathsf{con}}(\Gamma) \cup \mathsf{output}^{\mathsf{con}}(\Gamma)$. *Open:* $\mathsf{input}^{\mathsf{op}}(\Gamma) = \mathsf{input}(\Gamma) \setminus \mathsf{input}^{\mathsf{con}}(\Gamma)$, $\mathsf{output}^{\mathsf{op}}(\Gamma) = \mathsf{output}(\Gamma) \setminus \mathsf{output}^{\mathsf{con}}(\Gamma)$, *and* $\mathsf{ports}^{\mathsf{op}}(\Gamma) = \mathsf{input}^{\mathsf{op}}(\Gamma) \cup \mathsf{output}^{\mathsf{op}}(\Gamma)$. *Moreover, we denote by $\mathsf{trgt}^{\Gamma}(p_o) = \{p_i \mid (p_o, p_i) \in A\}$ the set of target ports for output port p_o and by $\mathsf{src}^{\Gamma}(p_i) = \{p_o \mid (p_o, p_i) \in A\}$ the set of source ports for input port p_i. Since $\mathsf{src}^{\Gamma}(p_i)$ is unique, with $\mathsf{src}^{\Gamma}(p_i)$ we denote the singleton set containing the unique output port.*

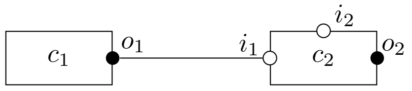

Figure 2. Architecture Configuration.

Figure 2 shows a conceptual representation of an architecture configuration $\Gamma = (C, A)$ consisting of two components $C = \{c_1, c_2\}$ and channel $(o_1, i_1) \in A$. It holds: $\mathsf{input}(\Gamma) = \{i_1, i_2\}$, $\mathsf{input}^{\mathsf{op}}(\Gamma) = \{i_2\}$, $\mathsf{input}^{\mathsf{con}}(\Gamma) = \{i_1\}$, $\mathsf{output}^{\mathsf{op}}(\Gamma) = \{o_2\}$, and $\mathsf{output}^{\mathsf{con}}(\Gamma) = \{o_1\}$.

Semantics: Now we are going to define the computational meaning of an architecture configuration. In the following, for ports P, valuation $\mu \in \overline{P}$, and $P' \subseteq P$, we write $\mu|_{P'}$ for the restriction of μ to ports P'.

Definition 4. *The semantics of an architecture configuration $\Gamma = (C, A)$ is a valuation $[\![(C, A)]\!]\colon \overline{\mathsf{input}^{\mathsf{op}}(\Gamma)} \to \overline{\mathsf{ports}(\Gamma)}$, defined as follows:*

$$[\![(C, A)]\!](\mu) = \Big\{\mu' \in \overline{\mathsf{ports}(\Gamma)} \;\Big|\; \quad (6)$$

$$\Big(\mu = \mu'|_{\mathsf{input}^{\mathsf{op}}(\Gamma)}\Big) \wedge \quad (7)$$

$$\left(\forall\, i \in \mathsf{input}^{\mathsf{con}}(\Gamma) : \mu'(i) = \mu'\left(\mathsf{src}^{\Gamma}(i)\right)\right) \wedge \quad (8)$$

$$(\forall\, o \in \mathsf{output}(\Gamma), ((I', O'), f') \in C\colon o \in O' \\ \implies \exists \xi \in f'(\mu'|_{I'})\colon \xi = \mu'|_{O'})\}\,. \quad (9)$$

Each element of the semantics $[\![(C, A)]\!](\mu)$ is created by constructing a valuation μ' of all the ports $\mathsf{ports}(\Gamma)$ of the architecture configuration. In fact, line (6) says that μ' is a valuation of all ports of the configuration and by line (7) we require that the valuation of all the open input ports of μ' are consistent with the valuation provided for the configurations open input ports. Line (8) says that if we require the value of a connected input port, then we use the value of the corresponding output port (according to channels A). Line (9) says that if we need a valuation provided by a component, then the computation proceeds according to the components behavior function.

Consider, for example, the architecture configuration given in Figure 2. Moreover, assume that component c_1 always returns a constant 2 at its output port o_1 and that component c_2 returns on its output port o_2 the product of the values at its input ports i_1 and i_2. We can now calculate port-valuations for each port (given a corresponding open-input-port valuation) according to Definition 4: Assume, for example, that $\mu(i_2) = 5$. According to (6), $\mu' \in \overline{\{i_1, i_2, o_1, o_2\}}$. Then, according to (9), $\mu'(o_1) = 2$. Thus, according to (8), $\mu'(i_1) = 2$, as well. Moreover, according to (7), $\mu'(i_2) = 5$. Thus, with (9), we have $\mu'(o_2) = \mu'(i_1) * \mu'(i_2) = 2 * 5 = 10$.

III. Specification Formalisms

In the following, we provide different approaches to specify components and architectures. First, we start by providing the notion of *interface assertion* as a simple means of specification, then we present *contracts* as an advanced mechanism

Table I
SUPPORTED CHECKS AND PROCEDURES.

Check	IA	C	MBC
Fulfillment (cf. Theorem 2)	✓	✓	✓
Under-specification (cf. Definition 9)	–	✓*	✓
Over-specification (cf. Definition 10)	–	✓*	✓

*only limited support

to specify components and also architectures. Finally, we introduce the notion of *mode-based contracts* which allows the specification of contracts depending on different system or operating modes.

For each concept we provide possible analysis techniques. Table I provides an overview of the supported specification techniques and the possible analyses. Each technique can also be applied to more elaborated concepts. Therefore, all techniques for interface assertions can also be applied to contracts and each technique discussed for contracts can be applied to mode-based contracts. On the other hand, techniques available for elaborated concepts are only limited (or not available, at all) in less elaborated concepts. Thus, under- and over-specification checks are only limited for pure A/G-contracts (i. e. they may provide false positives) and not available, at all, for plain interface assertions.

A. Interface Assertions

A very simple way to specify components is by means of so-called *interface assertions*.

Definition 5 (Interface Assertion). *An* interface assertion S *for a set of ports* $P \subseteq \mathtt{PORT}$ *is a predicate logical formula (with equality) with port labels as free variables. We denote with* $\mathtt{LOGI}(P)$ *the set of all logical formulae over port labels* $\ell\,(P)$.

Consider, for example, a simple component $inc = ((I, O), f)$ with input port $I = \{i\}$ and output port $O = \{o\}$ of type integer. Any logical formula with i and o as free variables, would be a valid interface assertion for the component. One example could be $o > i$ denoting the assertion that the output has to be greater than the corresponding input.

1) Fulfillment: Since an interface assertion is simply a logical formula over port labels, they can be used to specify both, components as well as complete architectures. An interface assertion can be used to specify the behavior of a component by a logical formula over its interface. In the following we define what it means for a component to fulfill an interface assertion.

Definition 6 (Interface Assertion Fulfillment: Component). *Given a component* $c = ((I, O), f)$, *with* $I, O \subseteq P$, *an interface assertion* $S \in \mathtt{LOGI}(I \cup O)$, *and port valuations* $x \in \overline{I}, y \in \overline{O}$, *the component* c fulfills *the interface assertion* S, *written* $c \vDash_{x,y} S$ *iff* $y \in f(x) \implies S\,[\ell\,(i) \mapsto x(i)]_{i \in I}\,[\ell\,(o) \mapsto y(o)]_{o \in O}$. *We write* $c \vDash S$ *to denote that a component* c *fulfills the interface assertion* S *for all* $x \in \overline{I}$ *and* $y \in \overline{O}$.

Considering again the example component $inc = ((I, O), f)$ introduced above, assuming that the corresponding behavior function is defined as $f(\mu)(o) = \mu(i) + 1$. That is, the component increases the input value by 1. According to Definition 6, the component fulfills the interface assertion $o > i$ since $i + 1 > i$ for each integer value i.

Similarly, interface assertions can also be used to specify the behavior of an architecture configuration by a logical formula over all ports of an architecture configuration. In the following we define what it means for an architecture configuration to fulfill an interface assertion.

Definition 7 (Interface Assertion Fulfillment: Architecture Configuration). *Given an architecture configuration* $\Gamma = (C, A)$, *an interface assertion* $S \in \underline{\mathtt{LOGI}}(\mathsf{ports}\,(\Gamma))$, *and port valuations* $x \in \overline{\mathsf{input}^{\mathsf{op}}\,(\Gamma)}, y \in \overline{\mathsf{ports}\,(\Gamma)}$, *the architecture* fulfills *the interface assertion* S, *written* $\Gamma \vDash_{x,y} S$ *iff* $y \in [\![(C, A)]\!](x) \implies S\,[\ell\,(p) \mapsto y(p)]_{p \in \mathsf{ports}(\Gamma)}$. *We write* $\Gamma \vDash S$ *to denote that an architecture configuration* Γ *fulfills the interface assertion* S *for all* $x \in \overline{\mathsf{input}^{\mathsf{op}}\,(\Gamma)}, y \in \overline{\mathsf{ports}\,(\Gamma)}$.

Note that in order for an architecture to fulfill an interface assertion, we require all ports to behave as specified by the assertion.

2) Interface Assertion Composition: Next, we present an important property of interface assertions. It allows us to calculate a *composed* interface assertion for an architecture configuration from the interface assertions of its components.

Theorem 1 (Interface Assertion Composition). *(1) Given architecture configuration* $\Gamma = (C, A)$ *and corresponding interface assertions* $(S_c)_{c \in C}$, *such that for all* $c \in C$, $c \vDash S_c$, *then,* $\Gamma \vDash \varphi(\Gamma)$, *with*

$$\varphi(\Gamma) \equiv \bigwedge_{c \in C} S_c \wedge \bigwedge_{(s,t) \in A} (\ell\,(s) = \ell\,(t))\,. \qquad (10)$$

(2) Moreover, if $(S_c)_{c \in C}$ *are the strongest interface assertions for components* $c = ((I, O), f) \in C$, *formally:*

$$\forall \alpha \in \mathtt{LOGI}(I \cup O) \colon c \vDash \alpha \implies (S_c \implies \alpha)\,, \qquad (11)$$

then, $\varphi(\Gamma)$ *is the strongest interface assertion satisfied by* Γ, *formally:*

$$\forall \alpha \in \mathtt{LOGI}(\mathsf{ports}\,(\Gamma)) \colon \Gamma \vDash \alpha \implies (\varphi(\Gamma) \implies \alpha)\,. \qquad (12)$$

Proof: (1) We show $\forall x \in \overline{\mathsf{input}^{\mathsf{op}}\,(\Gamma)}, y \in \overline{\mathsf{ports}\,(\Gamma)} \colon y \in [\![(C, A)]\!](x) \implies \varphi(\Gamma)\,[\ell\,(p) \mapsto y(p)]_{p \in \mathsf{ports}(\Gamma)}$ by contradiction and have $\Gamma \vDash \varphi(\Gamma)$ by Definition 7. Thus, we assume $\exists x \in \overline{\mathsf{input}^{\mathsf{op}}\,(\Gamma)}, y \in \overline{\mathsf{ports}\,(\Gamma)}$ such that $y \in [\![(C, A)]\!](x) \wedge \neg\varphi(\Gamma)\,[\ell\,(p) \mapsto y(p)]_{p \in \mathsf{ports}(\Gamma)}$. According to (10) we distinguish two cases:

- Case $\exists c = ((I, O), f) \in C \colon \neg S_c\,[\ell\,(p) \mapsto y(p)]_{p \in I \cup O}$: By (9) we have $y|_O \in f(y|_I)$. From $y|_O \in f(y|_I)$ and $\neg S_c\,[\ell\,(p) \mapsto y(p)]_{p \in I \cup O}$ follows by Definition 6 $\neg(c \vDash S_c)$, which is in contradiction to the assumption.
- Case $\exists (s, t) \in A$: $(\ell\,(s) \neq \ell\,(t))\,[\ell\,(s) \mapsto y(s),\ \ell\,(t) \mapsto y(t)]$:

Then, we have $t = \mathsf{trgt}^{\Gamma}(s)$ and $y(s) \neq y(t)$ which yields a contradiction to (8).

(2) Now assume that for all $c = ((I,O),f) \in C$ we have $\forall \alpha_c \in \mathtt{LOGI}(I \cup O)\colon c \vDash \alpha_c \implies (S_c \implies \alpha_c)$. Moreover, let $\alpha \in \mathtt{LOGI}(\mathsf{ports}(\Gamma))$ such that $\Gamma \vDash \alpha$ and assume $\varphi(\Gamma)$. We show α: Since $\Gamma \vDash \alpha$ holds, for all $c \in C$ there exists α_c such that $c \vDash \alpha_c$ and $\alpha = \bigwedge_{c\in C} \alpha_c$. Moreover, since $\varphi(\Gamma)$ holds, from (10) follows that S_c holds for all $c \in C$. Since $c \vDash \alpha_c$ and S_c for all $c \in C$ we have α_c for all $c \in C$ by (11). Thus, we finally have α by Definition. ■

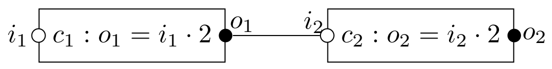

Figure 3. Two components and corresponding interface assertions.

Equation (10) consists of two parts. The first part consists of a conjunction of all the interface assertions of all the components of an architecture configuration. The second part consists of a so-called *port renaming* which is due to the fact that valuations of connected ports are the same.

Consider the simple architecture configuration Γ depicted in Figure 3 with components c_1 and c_2, satisfying interface assertions $S_1 \equiv (o_1 = i_1 \cdot 2)$ and $S_2 \equiv (o_2 = i_2 \cdot 2)$, respectively. According to Theorem 1, Γ fulfills the conjunction of S_1, S_2, and port renaming $o_1 = i_2$, respectively: $\varphi(\Gamma) \equiv (o_2 = i_2 \cdot 2 \wedge o_1 = i_1 \cdot 2 \wedge o_1 = i_2)$.

Based on Theorem 1 we can derive a method to check whether an architecture configuration fulfills a specification. Theorem 2 shows *soundness* and *relative completeness* of the method.

Theorem 2 (Fulfillment). *(1) For an architecture configuration $\Gamma = (C, A)$ with corresponding interface assertions $(S_c)_{c\in C}$, such that for all $c \in C$, $c \vDash S_c$, and an interface assertion $S \in \mathtt{LOGI}(\mathsf{ports}(\Gamma))$, such that*

$$\forall \mu \in \overline{\mathsf{ports}(\Gamma)}\colon \varphi(\Gamma)\,[\ell(p) \mapsto \mu(p)]_{p\in\mathsf{ports}(\Gamma)} \implies S\,[\ell(p) \mapsto \mu(p)]_{p\in\mathsf{ports}(\Gamma)}, \quad (13)$$

where $\varphi(\Gamma)$ is defined as in Theorem 1, the architecture configuration fulfills S, formally: $\Gamma \vDash S$.

(2) Moreover, the backward direction holds as well, if $(S_c)_{c\in C}$ are the strongest interface assertions for components $c = ((I,O),f) \in C$, formally:

$$\forall \alpha \in \mathtt{LOGI}(I \cup O)\colon c \vDash \alpha \implies (S_c \implies \alpha). \quad (14)$$

Proof: (1) We show $\forall x \in \overline{\mathsf{input}^{\mathsf{op}}(\Gamma)}, y \in \overline{\mathsf{ports}(\Gamma)}\colon y \in [\![(C,A)]\!](x) \implies S\,[\ell(p) \mapsto y(p)]_{p\in\mathsf{ports}(\Gamma)}$ and have $\Gamma \vDash S$ by Definition 7. Let $x \in \overline{\mathsf{input}^{\mathsf{op}}(\Gamma)}, y \in \overline{\mathsf{ports}(\Gamma)}$ and assume $y \in [\![(C,A)]\!](x)$. We show $\varphi(\Gamma)\,[\ell(p) \mapsto y(p)]_{p\in\mathsf{ports}(\Gamma)}$ and have $S\,[\ell(p) \mapsto y(p)]_{p\in\mathsf{ports}(\Gamma)}$ by assumption (13). By Theorem 1 we have $\Gamma \vDash \varphi(\Gamma)$ and since $y \in [\![(C,A)]\!](x)$ holds, $S\,[\ell(p) \mapsto y(p)]_{p\in\mathsf{ports}(\Gamma)}$ follows from Definition 7.

(2) Now let $\Gamma \vDash S$ and have $\forall x {\in} \overline{\mathsf{input}^{\mathsf{op}}(\Gamma)}, y {\in} \overline{\mathsf{ports}(\Gamma)}\colon y {\in} [\![(C,A)]\!](x) {\implies} S[\ell(p) {\mapsto} y(p)]_{p\in\mathsf{ports}(\Gamma)}$ by Definition 7. Furthermore, let $\mu \in \overline{\mathsf{ports}(\Gamma)}$ and assume $\varphi(\Gamma)\,[\ell(p) \mapsto \mu(p)]_{p\in\mathsf{ports}(\Gamma)}$. Since $(S_c)_{c\in C}$ are the strongest interface assertions (14) for $c = ((I,O),f) \in C$, by Theorem 1 we have $y \in [\![(C,A)]\!](x)$. Thus, by assumption we have $S\,[\ell(p) \mapsto y(p)]_{p\in\mathsf{ports}(\Gamma)}$. ■

Consider the simple example architecture configuration Γ depicted in Figure 3. Moreover, consider a specification $S \equiv (i_1 \geq 0 \implies o_2 \geq i_1)$ which requires output o_2 to be greater than the input i_1 for positive values of i_1. Note that indeed $\varphi(\Gamma) \implies S$ and it should be clear that the architecture configuration indeed fulfills S, since each component only increases its input.

B. Contracts

Sometimes interface assertion are separated into assumptions and guarantees to explicitly state the conditions under which a component is supposed to work correctly.

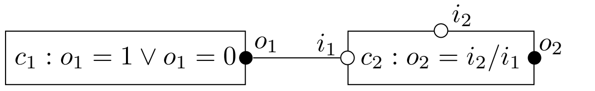

Figure 4. Component c_2: Division of two integers.

Consider, for example, component c_2 in Figure 4 with corresponding interface assertions. The component divides an integer passed through i_2 by an integer passed through i_1. The component guarantees to return the correct division only for the cases that on i_1 no 0 will appear. Therefore, one might add another interface assertion to specify under which conditions component c_2 works. The specification of component c_2 thus consists of two interface assertions, an assumption and a corresponding guarantee. Interface assertions, which are separated into assumptions and guarantees are called *contracts* and usually consist of several assumption and guarantee pairs. A useful contract K for component c_2 in Figure 4 would be $K = \left\{ (\overset{\text{assumption}}{i_1 \neq 0}, \overset{\text{guarantee}}{o_2 = i_2/i_1}) \right\}$. Contracts can be easily transferred into traditional interface assertions.

Definition 8 (Contract). *A* contract *over a set of ports $P \subseteq \mathtt{PORT}$ is a family $(A_j, G_j)_{j\in J}$ with* entries *$j \in J$ where for each $j \in J$, $A_j, G_j \in \mathtt{LOGI}(P)$. The interpretation of a contract $K = (A_j, G_j)_{j\in J}$ is defined as an interface assertion:*

$$[\![K]\!] = \bigwedge_{j\in J} (A_j \implies G_j). \quad (15)$$

The predicate logical formulae A_j and G_j, $j \in J$, are also referred to as assumptions *and* guarantees*, respectively. Given an architecture configuration $\Gamma = (C, A)$ and attached contracts K_c for each component $c \in C$, we denote the corresponding composed interface assertion with*

$$\varphi(\Gamma) \equiv \bigwedge_{c\in C} [\![K_c]\!] \wedge \bigwedge_{(s,t)\in A} (s = t). \quad (16)$$

In our simple example of Figure 4, the interface assertion for component c_2 with contract $K = \{(i_1 \neq 0, o_2 = i_2/i_1)\}$ would be $i_1 \neq 0 \implies o_2 = i_2/i_1$.

Since we can interpret each contract as interface assertions, all analyses for interface assertions presented so far, can be applied to contracts as well. Note that contracts, as opposed to interface assertions, consist of additional, structural information about an assertion. Each contract consists of an assumption and a guarantee part. Thus, the reverse is not true and the following analyses cannot be done with simple interface assertions but require contracts.

The additional information inherent in a contract specification can be exploited to perform additional analyses. Thus, in the following, we introduce two more analysis techniques for contracts: *Under-* and *over-specification*.

Definition 9 (Under-specified Component). *A component $c = ((I, O), f) \in C$ within an architecture configuration $\Gamma = (C, A)$ and contracts $(A_j, G_j)_{j \in J_d}$ for each component $d \in C$, is* under-specified *w.r.t.* (C, A) *iff*

$$\exists \mu \in \overline{\mathsf{ports}(\Gamma)} \colon \varphi(\Gamma)[\ell(p) \mapsto \mu(p)]_{p \in \mathsf{ports}(\Gamma)} \wedge \neg \bigvee_{j \in J_c} (A_j)[\ell(p) \mapsto \mu(p)]_{p \in I \cup O}. \quad (17)$$

In the architecture configuration depicted in Figure 4, component c_2 with corresponding contract $K = \{(i_1 \neq 0, o_2 = i_2/i_1)\}$ is under-specified since guarantee $o_1 = 1 \vee o_1 = 0$ holds for $\mu(o_1) = \mu(i_1) = 0$ while assumption $i_1 \neq 0$ does not.

Definition 10 (Over-specified Component). *A component $c = ((I, O), f) \in C$ within an architecture configuration $\Gamma = (C, A)$ and contracts $(A_j, G_j)_{j \in J_d}$ for each component $d \in C$, is* over-specified *w.r.t. entry $j \in J_c$ iff*

$$\forall \mu \in \overline{\mathsf{ports}(\Gamma)} \colon \varphi(\Gamma)[\ell(p) \mapsto \mu(p)]_{p \in \mathsf{ports}(\Gamma)} \implies \neg A_j[\ell(p) \mapsto \mu(p)]_{p \in I \cup O}. \quad (18)$$

Assume we want to modify component c_2 in Figure 4 such that it guarantees o_2 to be negative if i_1 is negative and i_2 is greater than 0. Therefore, we add an additional element $(i_1 < 0 \wedge i_2 > 0, o_2 < 0)$ to contract K. Then, c_2 is over-specified w.r.t. the new entry since due to the specification of c_1, i_1 will never be negative.

C. Mode-based Contracts

Technical systems such as those considered in this work can be decomposed into different *operating modes* (see e.g. Broy [19] and Vogelsang [20]). These operating modes help to specify a system behavior in a certain situation or under certain conditions. A common way to model modes is to use state diagrams or transition systems. Figure 5 depicts a—very simplified—example mode model. A vehicle can be in one of the modes sleep (ignition is off), live (ignition is on but engine is off), and drive (ignition is on and engine is on). Furthermore, in case of failures, several modes corresponding to graceful degradation levels can be active. Each level is assumed to support only degraded functionality or functions with less quality of service. In an airplane—just to mention a second domain—degraded situations are referred to as *flight control modes*. Hence, contracts may be different according to the current mode, thus they are referred to as *mode-based contracts*. In a degraded system mode, for instance, one could imagine to have less strict assumptions and qualities of service.

```
Algorithm 1: Check Fulfillment
  input  : Γ = (C, A) with corr. intf. assert. $(S_c)_{c \in C}$ and intf. asser. $S$.
  output : Returns true if $\Gamma \vDash S$ and false if $\Gamma \nvDash S$
1 if $\varphi(\Gamma) \implies S$ then return true;   /* Theorems 1 and 2 */
2 else return false;
```

Definition 11 (Mode Model). *Let* MODES *be the set of all modes. A* mode model *$\mathcal{M} = (M, m_0, \rightarrow, \Lambda)$ is a labeled transition system where $M \subseteq$ MODES is a finite, non-empty set of modes (states), $m_0 \in M$ is the initial mode, and $\rightarrow \subseteq M \times \Lambda \times M$ is the transition relation. Transitions are labeled with $\lambda \in \Lambda$ where Λ is the set of labels.*

Definition 12 (Mode-Based Contract). *A* mode-based contract *is a contract referring to a mode model $\mathcal{M} = (M, m_0, \rightarrow, \Lambda)$. For each mode $m \in M$ a set of* enabled *contracts for each component $c \in C$*

$$\mathsf{enabled}_c \colon M \rightarrow \mathcal{P}(J_c) \quad (19)$$

is given, yielding the index set $\mathsf{enabled}_c(m)$. The interpretation of a contract $K = (A_j, G_j)_{j \in J_c}$ for a mode $m \in M$ is defined as an interface assertion:

$$[\![K]\!](m) = \bigwedge_{j \in \mathsf{enabled}_c(m)} (A_j \implies G_j) \quad (20)$$

Of course, a system architect has to define $\mathsf{enabled}_c$ a priori.

IV. Analysis Techniques

This section details on specific analysis mechanisms that are applicable for the specification formalisms *interface assertions*, *contracts*, and *mode-based* contracts. The algorithms are based on theorems and definitions presented in Section III.

A. Interface Assertion Fulfillment

Algorithm 1 presents a procedure to check whether an interface assertion S is fulfilled by the composition of interface assertions of an architecture configuration Γ, i. e., $\Gamma \vDash S$ (according to Theorem 2).

Theorems 1 and 2 ensure *soundness* of Algorithm 1. Moreover, according to Theorems 1 and 2, *completeness* of the approach depends on the strength of the corresponding interface assertions $(S_c)_{c \in C}$. Completeness is only guaranteed if each contract S_c is the strongest possible contract for component c.

B. Under-Specification

As outlined in Table I, not all checks are supported for each specification mechanism. Under- and over-specified components embedded in an architecture configuration can be detected when using contracts—both standard and mode-based contracts. For the standard approach we assume that

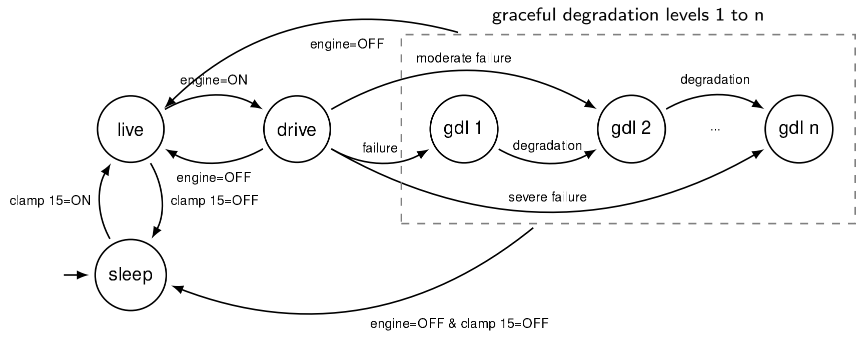

Figure 5. Simplified vehicle mode model with several levels of graceful degradation.

there exists only one "standard" mode, i. e., for mode model $\mathcal{M} = (M, m_0, \rightarrow, \Lambda)$ holds $|M| = 1$. All contracts are enabled in this mode. A component $c \in C$ of an architecture configuration (C, A) is under-specified, if there exists a port valuation under which *no* assumption $A_j, j \in \mathsf{enabled}_c(m)$ in conjunction with $\varphi(\Gamma)$ is satisfied.

Algorithm 2: Check Under-Specification

```
input  : $\Gamma = (C, A)$, $\mathcal{M} = (M, m_0, \rightarrow, \Lambda)$, and component $c \in C$
         to check. Components are attached with contract
         $(A_j, G_j)_{j \in J_c}$.
output : Returns mode $m$ if component $c$ is under-specified and false
         otherwise.
1 foreach $m \in M$ do                    /* Iterates over all modes */
2     foreach $j \in \mathsf{enabled}_c(m)$ do $\psi \leftarrow \psi \wedge \neg A_j$;   /* Def. 9 */
3     $\varphi' \leftarrow \varphi(\Gamma) \wedge \psi$;                        /* Theorem 1 */
4     if $\varphi'$ is satisfiable then return $m$;
5 od
6 return false;
```

C. Over-Specification

Besides under-specification, also over-specified components can be detected. A component $c \in C$ of an architecture configuration Γ is over-specified, if there exists *no* port valuation under which $\varphi(\Gamma) \wedge A_j$ is satisfied for an assumption $A_j, j \in \mathsf{enabled}_c(m)$. In Algorithm 3 the procedure is outlined.

Algorithm 3: Check Over-Specification

```
input  : $\Gamma = (C, A)$, $\mathcal{M} = (M, m_0, \rightarrow, \Lambda)$, and component $c \in C$
         to check. Components are attached with contract
         $(A_j, G_j)_{j \in J_c}$.
output : Returns mode $m$ and index $j$ if $c$ is over-specified and false
         otherwise.
1 foreach $m \in M$ do                    /* Iterates over all modes */
2     foreach $j \in \mathsf{enabled}_c(m)$ do
3         $\psi \leftarrow \varphi(\Gamma) \wedge A_j$;            /* Theorem 1, Def. 10 */
4         if $\psi$ is unsatisfiable then return $(m, j)$;
5     end
6 end
7 return false;
```

V. Example

We illustrate the application of the presented approach using an example developed in cooperation with our industrial partner. There we modeled an AUTOSAR software architecture

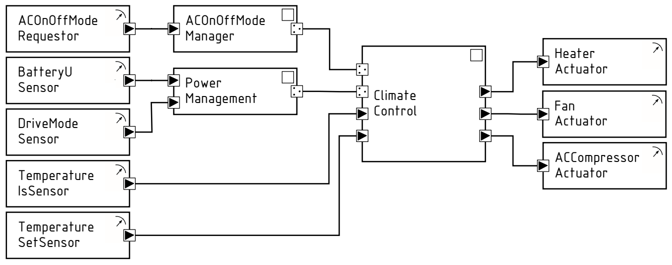

Figure 6. Part of an AUTOSAR software component architecture.

using ETAS's ISOLAR-A. The functionality encompasses the interplay of (i) the *electric break control*, (ii) the *engine control*, (iii) the *power management*, and (iv) the *climate control*. Due to place limitations, only a sufficient part to demonstrate the concepts is depicted in Figure 6 and discussed here. It provides an overview of the AUTOSAR software component architecture. Note, for the sake of clarity, we only consider a single "standard" mode, however the presented algorithms work for an arbitrary number of modes.

A. Encoding

We used the Z3 [21] SMT solver to realize the algorithms outlined in Section IV. Besides datatype (e. g. `(declare-datatypes () ((PMMODE LOW MID HIGH)))` and variable declarations (e. g. `(declare-const pm PMMODE)`), we translated the formulae into corresponding `assert` statements. Of course, they have to be transformed into equisatisfiable formulae in Łukasiewicz notation (prefix notation) before[1].

When iterating over the inner loop of Algorithm 3, we used the `push` and `pop` commands to store and restore the solver's logical context, respectively:

```
(push)
   (assert (and (< x 20)  (>  x 0)))
   (check-sat)
(pop)
(push)
   (assert (and (< x 50)  (>=  x 20)))
```

[1]The corresponding commented SMT-LIB code can be found at http://www4.in.tum.de/~marmsole/docs/2016MeMoCode/SMTLibspec.txt

```
    (check-sat)
(pop)
(push)
  (assert (and (< x 100)  (>=  x 50)))
  (check-sat)
```

This considerably improves the overall performance. The stated SMT-LIB commands encode the example of Section V-D.

B. Functional Description

The functionality of the `ClimateControl` component is to set the correct values for the `Heater`, `Fan`, and `ACCompressor` actuator software components, which are responsible to interact with corresponding hardware components. It sets these values depending on its inputs: the current temperature provided by `TemperatureIsSensor`, the set temperature provided by `TemperatureSetSensor`, the information about the air condition button state (`ACOnOffModeRequester` together with `ACOnOffModeManager`), and finally the currently available energy provided by the `PowerManagement` component. Depending on the battery voltage (`BatteryUSensor`) and the set drive mode (`DriveModeSensor`), e. g. eco or sport, it returns either `HIGH`, `MID`, or `LOW`.

Note, only relevant components are listed here. Port labels should be clear out of the context. Let $p_l \in P$ with $l = \ell(p_l)$ be a labeled port. We set the port types as follows: $\mathrm{type}(p_{dm}) = \mathrm{type}(p_{bat}) = \mathrm{type}(p_{is}) = \mathrm{type}(p_{set}) = \mathsf{int}$, for ports p_{ac} and p_{pm} we defined the new types $\mathsf{acT} = \{\mathsf{ON}, \mathsf{OFF}\}$ and $\mathsf{pmT} = \{\mathsf{LOW}, \mathsf{MID}, \mathsf{HIGH}\}$.

C. Interface Assertions and Contracts

The initial specification of the example is depicted in Table II. The object of investigation for under- and over-specification test is the `PowerManagement` component.

D. Under-Specification

Algorithm 3 returns the standard mode when applied on the `PowerManagement` component which means that it is *under-specified*. The used SMT solver returns as counterexample the assignment: $dm = 1$ and $bat = 0$. This is due to the underspecified assumption of `PowerManagement` (namely for $bat = 0$): $0 < dm \cdot bat < 20$. A more complete version is: $0 \leq dm \cdot bat < 20$.

E. Over-Specification

Next, we check on the corrected specification whether the power management component is over-specified. Therefore, we iterate (according to Algorithm 3) over all assumption parts of the given contract of component `PowerManagement`, yielding an over-specification for the assumption $50 \leq dm \cdot bat < 100$. The problem is encoded into SMT in a way such that getting a satisfiable assignment is a proof for no over-specification, whereas getting no model is a proof for over-specification. The reason for over-specification is, that due to the component's read data (dm and bat) which in their product can be at most 48, which is less than 50. Therefore, `PowerManagement` will never provide the result `HIGH`. This is not a problem in general, but it indicates a possible design flaw, which needs to be studied closely.

F. Fulfillment

Finally, we check whether *non-obvious interface assertions*—such as $S \equiv (dm = 1 \implies fan \leq 34)$—are fulfilled in an architecture configuration. This is true for the example (e. g. for $fan \leq 35$ this does not hold). The check encompasses the analysis of several connected components.

VI. Discussion

While the presented approach yields several benefits, we admit that there might be limitations. Some of which are discussed in the following:

A. Multiple Contracts vs. Interface Assertions

One possible weakness regards the need for multiple A/G contracts (as well as A/G contracts in general). Since they constitute special kinds of interface assertions, a developer might simply formulate the corresponding interface assertions in an adequate form (such as multiple implications) to allow for similar analyses. Note, however, that this requires high expertise and attention of the developer and may easily lead to erroneous specifications. Thus, the approach provides important methodological advice when designing component contracts.

B. Limited Expressiveness

Another possible weakness regards the expressiveness of the specification language. In the above example, we considered only the specification of stateless contracts, i. e., contracts which do not consider the state of a component. The reason for this was a pragmatic one since these primitive notion of contract allows for fully automated verification of multiple-contract compatibility. However, as argued in Section II-A, it is easy to extend the presented approach to stateful components.

C. Small Example Application

Another weakness concerns the simple architecture of the chosen example application. It consisted of a total of 11 components with rather simple contracts. However, the intention of the example was to ensure the feasibility of the approach. Now that this was shown, future work is needed to prove its applicability in real-life applications.

VII. Conclusion

We have provided a formal component model where components communicate via channels attached to typed ports. Each component is specified either using *interface assertions* or *(mode-based) contracts*. We have provided a method to compute composed interface assertions for an architecture configuration and have shown their *soundness* and *relative completeness*. Moreover, we showed how to check whether an architecture configuration fulfills a given interface assertion.

Table II
COMPONENT SPECIFICATIONS.

Software Component	Assumption	Guarantee
`BatteryUSensor`	true	$0 \leq bat \leq 12$
`DriveModeSensor`	true	$0 < dm < 5$
`TemperatureIsSensor`	true	$-30 \leq is \leq 50$
`TemperatureSetSensor`	true	$16 \leq set \leq 30$
`ACOnOffModeManager`	true	$ac = \texttt{ON} \vee ac = \texttt{OFF}$
`PowerManagement`	$0 < dm \cdot bat < 20$	$pm = \texttt{LOW}$
	$20 \leq dm \cdot bat < 50$	$pm = \texttt{MID}$
	$50 \leq dm \cdot bat < 100$	$pm = \texttt{HIGH}$
`ClimateControl`		

ac	pm	$is - set$	$heater$	fan	ac_comp
ON	LOW	≥ 0	0	$(is - set) \cdot 1$	$(is - set) \cdot 1$
ON	MID	≥ 0	0	$(is - set) \cdot 2$	$(is - set) \cdot 2$
ON	HIGH	≥ 0	0	$(is - set) \cdot 3$	$(is - set) \cdot 3$
OFF	LOW	≥ 0	0	$(is - set) \cdot 1$	0
OFF	MID	≥ 0	0	$(is - set) \cdot 2$	0
OFF	HIGH	≥ 0	0	$(is - set) \cdot 3$	0
ON	LOW	< 0	$(is - set) \cdot 1$	0	0
ON	MID	< 0	$(is - set) \cdot 2$	0	0
ON	HIGH	< 0	$(is - set) \cdot 3$	0	0
OFF	LOW	< 0	$(is - set) \cdot 1$	0	0
OFF	MID	< 0	$(is - set) \cdot 2$	0	0
OFF	HIGH	< 0	$(is - set) \cdot 3$	0	0

We argued that traditional A/G reasoning is in some cases insufficient and hence extended the approach to mode-based contracts that allow in addition to check whether a component is over- or under-specified. Since contracts were introduced as special kind of interface assertions, both notions can easily be combined. Algorithms for all checks have been given and an SMT-based realization has been sketched. The practical applicability depends on the decidability of the used background theory and completeness of the solver. Finally, an example demonstrated the approach.

Future work arises in two areas: (i) *Framework*: Extend towards architecture contracts, i. e., contracts over the whole architecture to allow A/G reasoning for hierarchical component models; (ii) *Evaluation*: Integrate the approach in the industry partner's tool chain and further investigate the capabilities in an industrial development project.

ACKNOWLEDGMENT

We thank Andreas Vogelsang and Vasileios Koutsoumpas for fruitful comments on earlier version of this work.

REFERENCES

[1] M. Broy, S. Kirstan, H. Krcmar, and B. Schätz, "What is the benefit of a model-based design of embedded software systems in the car industry?" *Software Design and Development: Concepts, Methodologies, Tools, and Applications: Concepts, Methodologies, Tools, and Applications*, p. 310, 2013.

[2] acatech, Ed., *Cyber-Physical Systems: Driving Force for Innovation in Mobility, Health, Energy and Production*, ser. acatech Position. Munich, Germany: acatech – National Academy of Science and Engineering, Dec 2011.

[3] ISO, "Road vehicles–Functional safety (ISO 26262)," 2011.

[4] B. Meyer, "Applying "design by contract"," *IEEE Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[5] L. De Alfaro and T. A. Henzinger, "Interface theories for component-based design," in *International Workshop on Embedded Software*. Springer, 2001, pp. 148–165.

[6] B. Meyer, "Eiffel: A language and environment for software engineering," *J. Syst. Softw.*, vol. 8, no. 3, pp. 199–246, Jun. 1988.

[7] ——, "Design by contract: The eiffel method," in *TOOLS 1998*. IEEE Computer Society, 1998, p. 446.

[8] A. Cimatti, M. Dorigatti, and S. Tonetta, "Ocra: A tool for checking the refinement of temporal contracts," in *ASE 2013*. IEEE, 2013, pp. 702–705.

[9] G. T. Leavens and Y. Cheon, "Design by Contract with JML," 2006.

[10] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll, "An Overview of JML Tools and Applications," *Int. J. Softw. Tools Technol. Transf.*, vol. 7, no. 3, pp. 212–232, Jun. 2005.

[11] M. Barnett, K. R. M. Leino, and W. Schulte, "The Spec# Programming System: An Overview," in *CASSIS 2004*, ser. LNCS, vol. 3362. Springer, 2005, pp. 49–69.

[12] M. Emmi, D. Giannakopoulou, and C. S. Păsăreanu, "Assume-guarantee verification for interface automata," in *FM 2008: Formal Methods*. Springer, 2008, pp. 116–131.

[13] W. Damm, H. Hungar, B. Josko, T. Peikenkamp, and I. Stierand, "Using contract-based component specifications for virtual integration testing and architecture design," in *DATE 2011*. IEEE, 2011, pp. 1023–1028.

[14] M. Broy, "Towards a Theory of Architectural Contracts: - Schemes and Patterns of Assumption/Promise Based System Specification," in *Software and Systems Safety - Specification and Verification*. IOS Press, 2011, vol. 30, pp. 33–87.

[15] C. Chilton, B. Jonsson, and M. Z. Kwiatkowska, "Assume-guarantee reasoning for safe component behaviours." in *FACS*, vol. 12. Springer, 2012, pp. 92–109.

[16] P. Nuzzo, A. Iannopollo, S. Tripakis, and A. Sangiovanni-Vincentelli, "Are interface theories equivalent to contract theories?" in *Formal Methods and Models for Codesign (MEMOCODE), 2014 Twelfth ACM/IEEE International Conference on*. IEEE, 2014, pp. 104–113.

[17] R. H. Reussner, S. Becker, and V. Firus, "Component composition with parametric contracts," *Tagungsband der Net. ObjectDays*, vol. 2004, pp. 155–169, 2004.

[18] A. Malkis and D. Marmsoler, "A model of service-oriented architectures," in *Components, Architectures and Reuse Software (SBCARS), 2015 IX Brazilian Symposium on*. IEEE, 2015, pp. 110–119.

[19] M. Broy, "Multifunctional software systems: Structured modeling and specification of functional requirements," *Science of Computer Programming*, vol. 75, no. 12, pp. 1193–1214, 2010.

[20] A. Vogelsang, "Model-based requirements engineering for multifunctional systems," Dissertation, Technische Universität München, München, 2015.

[21] L. De Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *TACAS 2008*. Springer-Verlag, 2008, pp. 337–340.