# Contract-Based Requirement Modularization via Synthesis of Correct Decompositions

THI THIEU HOA LE and ROBERTO PASSERONE, DISI, University of Trento, Italy
ULI FAHRENBERG and AXEL LEGAY, Inria/Irisa, Rennes, France



In distributed development of modern systems, contracts play a vital role in ensuring interoperability of components and adherence to specifications. It is therefore often desirable to verify the satisfaction of an overall property represented as a contract, given the satisfaction of smaller properties also represented as contracts. When the verification result is negative, designers must face the issue of refining the subproperties and components. This is an instance of the classical synthesis problems: "can we construct a model that satisfies some given specification?" In this work, we propose two strategies enabling designers to synthesize or refine a set of contracts so that their composition satisfies a given contract. We develop a generic algebraic method and show how it can be applied in different contract models to support top-down component-based development of distributed systems.

Categories and Subject Descriptors: I.6.4 [**Computing Methodologies**]: Model Validation and Analysis

General Terms: Design, Theory, Verification

Additional Key Words and Phrases: Contract, requirement, decomposition, synthesis, distributed systems

**ACM Reference Format:**
Thi Thieu Hoa Le, Roberto Passerone, Uli Fahrenberg, and Axel Legay. 2016. Contract-based requirement modularization via synthesis of correct decompositions. ACM Trans. Embed. Comput. Syst. 15, 2, Article 33 (February 2016), 26 pages.
DOI: http://dx.doi.org/10.1145/2885752

## 1. INTRODUCTION

Modern computing systems are increasingly being built by composing components that are developed concurrently by different design teams. In such a paradigm, the distinction between what is constrained on environments and what must be guaranteed by a system given the constraint satisfaction reflects the different roles and responsibilities in the system design procedure. Component-based and contract-based design have been shown to be a rigorous and effective approach for designing such concurrent systems [Davare et al. 2013; Bauer et al. 2012; Meyer 1992; de Alfaro and Henzinger 2001]. Different components of the same system can be developed by different teams in an independent and concurrent manner provided that their associated contracts can synchronize and satisfy predefined properties. Formally, a contract is a pair of *assumptions* and *guarantees*, which intuitively are properties that must be satisfied by all inputs and outputs of a design, respectively. Such separation between assumptions

Authors' addresses: T. T. H. Le and R. Passerone, Dipartimento di Ingegneria e Scienza dell'Informazione, University of Trento, via Sommarive 9, 38123 Trento, Italy; emails: lethithieuhoa@gmail.com, roberto.passerone@unitn.it; U. Fahrenberg and A. Legay, Campus Universitaire de Beaulieu, 263 Avenue du General Leclerc, 35042, IRISA Rennes, France; emails: {ulrich.fahrenberg, axel.legay}@irisa.fr.

DOI: http://dx.doi.org/10.1145/2885752

and guarantees allows an efficient reuse of already-designed components, thereby supporting the distributed development of complex systems effectively.

Components can be formed by a bottom-up composition of simpler predefined components. They can alternatively be formed by a top-down decomposition into subcomponents defined by a set of subcontracts, as long as the composition of the subcontracts satisfies or refines the contract of the system as a whole. This approach is most appropriate when a design needs to be distributed among several design teams or contractors, since it clearly establishes the responsibilities and decomposes the issue of correctness into smaller local verification subproblems that can be addressed before system integration [Sangiovanni-Vincentelli et al. 2012]. The main hurdle with this method is how to budget the system specification across the different components, by strengthening and weakening their respective assumptions and guarantees while achieving overall correctness with respect to the system contract. When this condition is not satisfied, i.e., the subcontract composition does not refine the overall contract, designers must refine the subcontract specifications until the system is proved correct. This is an instance of the classical synthesis problems: "can we construct a model that satisfies some given specification?" In this article, we deal with the problem of checking if a contract $\mathcal{C}$ can be decomposed into a set of contracts $\{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$ and that of synthesizing the contract set to make their composition refine $\mathcal{C}$ when necessary. In particular, we study *decomposing conditions* under which the contract decomposition can be verified and thereby propose a generic *synthesis strategy* for fixing wrong decompositions. Our conditions and synthesis strategy can be applied to generic contract frameworks equipped with specification operators (e.g., composition, refinement) including popular frameworks such as trace-based or modal contract frameworks. This article extends our previous results obtained in this area [Le and Passerone 2014; Le et al. 2016]. In particular, we describe in detail and compare two synthesis strategies in terms of usability and computational complexity. We also formally analyze the issues of soundness and completeness of the synthesis strategies. Finally, we frame our techniques in the context of an overall design methodology.

The rest of the article is organized as follows. In Section 3, we recall basic notions of contracts and how a generic contract can be derived from a specification theory equipped with composition, quotient, refinement, and conjunction. Section 4 describes how our techniques fit into an overall design methodology. Then, based on the decomposition into a set of *two* contracts [Bauer et al. 2012], we propose decomposing conditions for a set of $n$ contracts in Section 5. Section 6 presents our two synthesis strategies to make the set of contracts satisfy a predefined contract, together with theoretical results. We then demonstrate our strategy for synthesizing trace-based contracts in Section 7 and modal contracts in Section 8. We conclude in Section 9.

## 2. RELATED WORK

Contracts were first introduced in Bertrand Meyer's design-by-contract method [Meyer 1992], based on ideas by Dijkstra [1975] and Lamport [1990], and others, where systems are viewed as abstract boxes achieving their common goal by verifying specified contracts. Such technique guarantees that methods of a class provide some postconditions at their termination, as long as the preconditions under which they operate are satisfied. De Alfaro and Henzinger subsequently introduced interface automata [de Alfaro and Henzinger 2001] for documenting components, thereby enabling them to be reused effectively. This formalism establishes a more general notion of contract, where preconditions and postconditions, which originally appeared in the form of predicates, are generalized to behavioral interfaces. The central issues when introducing the formalism of interface automata are compatibility, composition, and refinement. Separating assumptions from guarantees, which was somewhat implicit in interface automata,

has then been made explicit in the contract framework of the SPEEDS HRC model [Benveniste et al. 2008; Benvenuti et al. 2008]. A separation between specifying assumptions on expected behaviors and guarantees to achieve them at runtime has recently been applied to the handling of synchronization requirements to improve the component-based development of real-time high-integrity systems [López Martínez and Vardanega 2012].

The relationship between specifications of component behaviors and contracts is further studied by Bauer et al. [2012], where a contract framework can be built on top of any *specification theory* equipped with a composition operator and a refinement relation that satisfy certain properties. Trace-based [Benveniste et al. 2008] and modal contract [Raclet et al. 2011] theories are also demonstrated to be instances of such a framework. This formalization enables verifying if a contract can be decomposed into two other contracts by checking if that contract can *dominate* the others. In this work, we take advantage of such a dominating notion and generalize it to a set of $n$ contracts and construct generic decomposing conditions for the contract set.

Several methods have been reported in the literature for checking contract refinement. Quinton et al. have developed a hierarchical approach that deals with the problem of checking contract refinement by decomposing a large verification task into smaller problems that involve a limited number of assumptions and guarantees, and then relying on compositional methods and circular reasoning to deduce the global result [Graf et al. 2014]. The tool is part of the BIP framework and uses a model based on modal transition systems (MTS). Later, Raclet et al. developed a theory of contracts based on modal specifications implemented in the InterSMV toolset, dedicated to checking dominance and refinement between contracts [Raclet et al. 2011]. Contract-based specification methods were extended to timed models by David et al. [2010], using timed I/O automata and constructs for refinement and consistency checking. The theory is implemented on top of the Uppaal-tiga engine for timed games. Benvenuti et al. extend refinement checking to hybrid automata to account for continuous time components, supported by the Ariadne tool [Benvenuti et al. 2012]. To improve scalability, Iannapollo et al. propose a library-based approach to refinement checking of contracts expressed in LTL [Iannopollo et al. 2014]. Our approach is complementary. In fact, although these methods may be effective in determining when refinement holds, they do not provide guidance as to how the decomposition needs to be changed when this is not the case. The verification problem of decomposing a contract into a set of contracts was also studied by Cimatti and Tonetta [2012] and was addressed by property-based proof systems with SMT-based model checking techniques, and supported by the OCRA tool. The contract specifications allowed in such systems, however, are trace-based only. Our decomposing conditions can instead deal with generic contract specifications including both trace-based and modal ones.

Assume-guarantee reasoning (AGR) has also been applied extensively in declarative compositional reasoning [de Roever 1985] to help prove properties by decomposing them into simpler and more manageable steps. The classical AGR uses assumptions as hypotheses to establish whether a generic property holds. Naturally, this technique can be used in contract models as well, with possibly nontrivial transformation and formalization. In case of unsuccessful termination, AGR can also provide a counterexample showing how the property can be violated. Such a counterexample can then be used to synthesize the model so as to satisfy a given property [Lin and Hsiung 2011]. However, this synthesis strategy is only applicable for systems with trace-based semantics. Viewing the same assume-guarantee synthesis problem as a game, Chatterjee et al. solve it by finding a winning strategy on the global system state graph, but the method does not guarantee the inclusion of all traces satisfying the specification [Chatterjee and Henzinger 2007]. The synthesized model was shown to be a subset of that

synthesized by counterexample-based synthesis [Lin and Hsiung 2011]. Unlike these concrete notions of synthesis, ours is more generic since it is not tied to the system semantics. Moreover, whereas the application of our synthesis strategy to generic contract-based systems is direct and straightforward, the generalization of the previous approaches has not been studied and would require a conversion process from normalized contracts to unnormalized ones.

## 3. BACKGROUND: SPECIFICATION AND CONTRACT THEORIES

In this section, we briefly review the specification theory that we adopt to represent components and describe how this is lifted to a contract model. In particular, we will define the notions of component, contract, and its environment and implementations.

### 3.1. Specification Theory

For our formalization, we follow the notation introduced by Bauer et al. [2012], which is built on top of a specification theory of components equipped with a refinement ($\leq$) and a composition ($\|$) operator. Components are generic entities that represent objects that can be hierarchically combined to compose systems. Composition is an operator that takes two components $S$ and $T$ and produces an overall specification $S \parallel T$ for a component that behaves according to their interaction. Refinement, on the other hand, can be used to compare components. Intuitively, a component $S'$ refines a component $S$ whenever $S'$ can replace $S$ in all of its contexts of use. The theory operators are *metatheoretical* or uninterpreted operators, meaning that we do not need to know their exact semantics as long as they satisfy certain properties [Bordin and Vardanega 2007]. This ensures that our results can be applied indifferently to several concrete specification theories. To make the approach generic, the properties required of the model are not particularly demanding and are met by most compositional models. In particular, we require that parallel composition of components be both commutative and associative. Similarly, the refinement relation must be a preorder, and therefore it must be reflexive and transitive. Conversely, antisymmetry, thus a partial order, is not strictly required, although several specification theories have this characteristic. To support a compositional approach, we also require that composition be monotonic relative to refinement, i.e., that

$$(S' \leq S) \wedge (T' \leq T) \Rightarrow (S' \parallel T') \leq (S \parallel T).$$

Two other operators that can be defined on top of composition and refinement are quotient ($/$) and conjunction ($\curlywedge$). The quotient between specifications $T$ and $S$, written $T/S$, is a specification $R$ such that its composition with $S$ can concretize or refine $T$, i.e.,

$$S \parallel R \leq T.$$

As there may exist many such specifications, the quotient is defined to be the greatest specification in the refinement order of all such $R$:

$$((S \parallel (T/S)) \leq T) \wedge ((S \parallel R) \leq T \Rightarrow R \leq T/S).$$

The quotient $T/S$ is the most permissive specification that, when composed with $S$, satisfies $T$. Thus, intuitively, we can use the quotient to "correct" the specification $S$ using the component $R$.

Likewise, the conjunction operator computes the greatest lower bound in the refinement order of the original specification:

$$\begin{gathered}((S \curlywedge T) \leq S) \wedge \\ ((S \curlywedge T) \leq T) \wedge \\ (R \leq S \wedge R \leq T \Rightarrow R \leq (S \curlywedge T)).\end{gathered}$$

Conjunction is used to combine specifications for the *same* component. These specifications typically pertain to different *aspects* or *viewpoints* of the component and are given separately to simplify the design process. In the rest of the article, we assume that quotient and conjunction exist for all specifications. On the other hand, the refinement order does not need to be complete to apply the techniques described in this work.

In most concrete models, specifications are expressed in terms of a certain *alphabet* of actions, ports, or variables, which the components use to interface to the rest of the system. For instance, state-based models employ variables to exchange data and control information, whereas process-based models typically make use of actions. Data-flow models, on the other hand, communicate through ports carrying tokens. By resting on a generic notion of composition and refinement, our methodology is substantially independent of these aspects, which will be introduced only with the concrete examples. In general, however, we assume that the operators and the relations, such as composition, quotient, and conjunction, defined on the model provide ways to handle these alphabets and to compute the alphabet of the composite structures.

### 3.2. Contract Theory

Assuming the existence of such underlying specification theory, a contract of a component can be defined formally as a pair of specifications, called the *assumptions* and the *guarantees*:

$$\mathcal{C} = (\mathcal{A}, \mathcal{G}).$$

The assumption $\mathcal{A}$ is a specification that expresses how the environment is allowed to handle the components. To put it another way, the component can only be used in an environment that satisfies the assumption specification. This clearly defines the context of use of the component. Conversely, the guarantee $\mathcal{G}$ is a specification that describes what the component is allowed to do or what it guarantees. The semantics of the contract is such that the component must satisfy the guarantees, but only in the context of an environment that satisfies the assumptions. An implementation of the component thus satisfies its contract whenever it satisfies the contract guarantee, subject to the contract assumption. The contract semantics is therefore defined through the notions of such environments and implementations. We will now make these notions more precise.

An environment $\mathcal{E}$ satisfies contract $\mathcal{C} = (\mathcal{A}, \mathcal{G})$ whenever $\mathcal{E} \leq \mathcal{A}$. Let $[\![\mathcal{C}]\!]_{\mathsf{e}}$ be the set of environments of $\mathcal{C}$. We say that an implementation $\mathcal{I}$ satisfies contract $\mathcal{C}$ if

$$\forall \mathcal{E} \in [\![\mathcal{C}]\!]_{\mathsf{e}} : \mathcal{I} \parallel \mathcal{E} \leq \mathcal{G} \parallel \mathcal{E}$$

holds. We denote the set of all possible implementation similarly by $[\![\mathcal{C}]\!]_{\mathsf{p}}$. Two contracts $\mathcal{C}_1$ and $\mathcal{C}_2$ have identical semantics and are *equivalent* if they possess the same set of environments and implementations, i.e.,

$$([\![\mathcal{C}_1]\!]_{\mathsf{e}} = [\![\mathcal{C}_2]\!]_{\mathsf{e}}) \wedge ([\![\mathcal{C}_1]\!]_{\mathsf{p}} = [\![\mathcal{C}_2]\!]_{\mathsf{p}}).$$

The implementation semantics of a contract, namely its sets of implementations $[\![\mathcal{C}]\!]_{\mathsf{p}}$, generally depends on both the assumption $\mathcal{A}$ and the guarantee $\mathcal{G}$. Without loss of generality [Bauer et al. 2012], we assume that for every contract $\mathcal{C} = (\mathcal{A}, \mathcal{G})$, there exists contract $\mathcal{C}^{\mathsf{n}} = (\mathcal{A}, \mathcal{G}^{\mathsf{n}})$, which is equivalent to $\mathcal{C}$ and where the implementation semantics is independent of the assumption presence. This happens when there is a way to incorporate the original assumption and guarantee into the new guarantee $\mathcal{G}^{\mathsf{n}}$. We call $\mathcal{C}^{\mathsf{n}}$ the normalized form of $\mathcal{C}$ and derive $\mathcal{G}^{\mathsf{n}}$ using the normalization operator $\rhd$:

$$\mathcal{G}^{\mathsf{n}} = \mathcal{G} \rhd \mathcal{A}.$$

The normalization operator can generally be defined on top of the basic operators $\leq$, $\|$, $/$, $\curlywedge$. Its precise expression, however, depends on the specific concrete model in use. For this reason, we defer examples of normalization to later parts of our article, namely Examples 7.1 and 8.2 (see Table I). On the other hand, one can implicitly define the normalized form by recalling its semantics as follows.

*Definition* 3.1. A contract $\mathcal{C} = (\mathcal{A}, \mathcal{G})$ is in normalized form if and only if

$$\mathcal{I} \in [\![\mathcal{C}]\!]_{\mathsf{p}} \Leftrightarrow \mathcal{I} \leq \mathcal{G}.$$

A refinement relation between contracts can then be established based on that between their environment sets and implementation sets.

*Definition* 3.2. Contract $\mathcal{C}$ is said to refine $\mathcal{C}'$, written $\mathcal{C} \preceq \mathcal{C}'$, when it can accept more environments and fewer implementations than contract $\mathcal{C}'$:

$$[\![\mathcal{C}']\!]_{\mathsf{e}} \subseteq [\![\mathcal{C}]\!]_{\mathsf{e}} \wedge [\![\mathcal{C}]\!]_{\mathsf{p}} \subseteq [\![\mathcal{C}']\!]_{\mathsf{p}}.$$

Contract refinement can be seen as a notion of substitutability. In particular, by accepting *more* environments, the concrete contract can be used in all of those contexts in which the abstract contract can be used. Similarly, by having fewer implementations, the concrete contracts provide stronger guarantees, which are required to discharge the assumptions of the other components in the system.

## 4. DESIGN METHODOLOGIES

The contract framework supports several different design methodologies that are important in practice. These should be seen as prototype work flows, which in a concrete development process can be freely combined. We refer to David et al. [2009] for more background on design methodologies.

*Stepwise refinement*. Contract refinement $\mathcal{C} \preceq \mathcal{C}'$, as defined earlier, can be used to gradually refine a contract specification until a desired level of detail has been reached. As per the definition of refinement, application of $\preceq$ increases permissibility toward the environment and decreases the number of implementation choices; hence, iterated refinement allows one to reach any desired level of permissibility and any desired level of implementation detail. This is closely related to *behavioral subtyping* [Liskov and Wing 1994].

Stepwise refinement is hence suitable for a top-down design paradigm. First, a high-level contract is developed that represents an overall view of the system to be implemented, and then this contract is successively refined to devise an implementation. This is especially useful in conjunction with compositionality as follows.

*Compositionality*. We will see that contract refinement is preserved under contract composition (Theorem 5.4 (iiib), shown later); hence, stepwise refinement can be applied freely in subcomponents of a given contract: by Theorem 5.4, if a subcontract in a composition is refined to increase environmental permissibility and decrease the number of implementation choices, then this automatically induces a refinement between the global combined contracts. Therefore, compositions of contracts can be refined individually without breaking global refinement.

Within a top-down design paradigm, this means that when design requirements have been modularized, i.e., split into contracts for components, then each component can be refined and implemented independently, even by independent development teams.

Compositionality guarantees that the implementations can be composed and that the composition satisfies the original global contract.

*Modularization of requirements*. The central contribution of this work is to provide support for modularization of requirements, i.e., for splitting a global contract into contracts for components. This generally cannot be done in a completely automated manner and hence requires (human) domain expertise and may introduce errors in the process. These errors are easy enough to detect, as one can simply check whether the composition of the component contracts refines the original global contract; what we propose here is a methodology for *correcting* such errors automatically.

Note how stepwise refinement, compositionality, and modularization combine to support a complete top-down design process. The process begins with a high-level contract that represents global assumptions and guarantees of the system to be implemented; then, the global requirements are modularized into contracts for components; these contracts are further refined (and perhaps further modularized) and eventually implemented, possibly by independent teams; compositionality then guarantees that the system composed of all components' implementations automatically satisfies the original global contract.

In some what more detail, our methodology supports modularization in the following way (we shall give precise formal definitions of the involved concepts later in the article). Suppose that a designer has developed a global contract $\mathcal{C} = (\mathcal{A}, \mathcal{G})$ for a system to be implemented, and that she has a rough idea that the system falls into a number of $n$ different components that can be developed independently. The developer then goes on to develop a set of contracts $\{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$ for these components. Checking whether the decomposition is right (using a tool that can do so automatically), the developer notices that the composition $\bigodot_{1 \leq i \leq n} \mathcal{C}_i$ (which we shall define precisely in the next section) does not refine the global contract $\mathcal{C}$. Our work now provides a method, fully automatic, to adapt some of the components' contracts so that this refinement is enforced. In other words, the method returns a set $\{\mathcal{C}'_1, \ldots, \mathcal{C}'_n\}$ of new contracts, with the properties that (1) $\mathcal{C}'_i \prec \mathcal{C}_i$ for each $i$, and (2) $\bigodot_{1 \leq i \leq n} \mathcal{C}'_i \prec \mathcal{C}$.

Hence, the component contracts that the developer proposed have been altered so that (1) they are more permissive toward their environment and have fewer implementation choices, and (2) their composition is refined by the original global contract. These component contracts can now safely be implemented, possibly by independent teams, and the composition of the implementations is guaranteed to satisfy the global contract.

## 5. CONTRACT COMPOSITION AND DECOMPOSITION

Contract composition is formalized so that the compositionality between their implementations can be respected, i.e., composing such implementations results in an implementation of the composite contract. Because contracts include assumptions, every environment of the composite contract should also be able to work with any implementation of an individual contract in a way that their composition does not violate the other contract assumption. In fact, there exists a class of contracts, including the composite contract, able to provide such desirable consequences. These are referred to as *dominating* contracts [Bauer et al. 2012], and the composite contract is the least in the refinement order of all dominating contracts, as we shall see in Section 5.1.

This notion of dominance thus enables the compositionality of the implementation relation, an important principle in reusing components and decomposing systems into existing components. Before studying contract decomposition (Section 5.2), we first generalize the notion of dominance and composition from two contracts [Bauer et al. 2012] to a set of $n$ contracts.

### 5.1. Contract Composition

*Definition* 5.1. A contract $\mathcal{C} = (\mathcal{A}, \mathcal{G})$ *dominates* the contract set $\{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$ whenever

(i) $\forall \mathcal{I}_1 \in [\![\mathcal{C}_1]\!]_\mathsf{p}, \ldots, \forall \mathcal{I}_n \in [\![\mathcal{C}_n]\!]_\mathsf{p} : \; \|_{1 \le i \le n} \; \mathcal{I}_i \in [\![\mathcal{C}]\!]_\mathsf{p}$,

(ii) $\forall \mathcal{E} \in [\![\mathcal{C}]\!]_\mathsf{e}, \forall \mathcal{I}_1 \in [\![\mathcal{C}_1]\!]_\mathsf{p}, \ldots, \forall \mathcal{I}_n \in [\![\mathcal{C}_n]\!]_\mathsf{p}, \forall 1 \le i \le n : \mathcal{E} \parallel \|_{1 \le j \ne i \le n} \; \mathcal{I}_j \le \mathcal{A}_i$.

The first condition formalizes the idea that the composition of any arbitrary set of implementations of the individual contracts in the set must be an implementation of the overall contract. The second condition ensures that any environment of the overall contract can be used as an environment of any implementation of an individual contract, given any arbitrary implementation of the other contracts.

Because the set of implementations of a contract are fully characterized by the normalized guarantees of the contract, these can be conveniently used to give an alternate formulation of the preceding conditions. Specifically, the following theorem reduces checking the two conditions in Definition 5.1 to checking simpler formulas.

THEOREM 5.2. *Condition* (i) *is equivalent to*

$$\|_{1 \le i \le n} \; \mathcal{G}_i^\mathsf{n} \in [\![\mathcal{C}]\!]_\mathsf{p}.$$

*Condition* (ii) *is equivalent to*

$$\forall 1 \le i \le n : \mathcal{A} \parallel \|_{1 \le j \ne i \le n} \; \mathcal{G}_j^\mathsf{n} \le \mathcal{A}_i.$$

PROOF. For each condition, we will prove equivalence by showing that one condition implies the other:

(i) $\Rightarrow$: Consider $\mathcal{I}_i = \mathcal{G}_i^\mathsf{n}$.
$\Leftarrow$: $\mathcal{I}_i \le \mathcal{G}_i^\mathsf{n} \Rightarrow \forall \mathcal{E} \in [\![\mathcal{C}]\!]_\mathsf{e} : (\mathcal{E} \parallel \|_{1 \le i \le n} \; \mathcal{I}_i) \le (\mathcal{E} \parallel \|_{1 \le i \le n} \; \mathcal{G}_i^\mathsf{n}) \Rightarrow \|_{1 \le i \le n} \; \mathcal{I}_i \in [\![\mathcal{C}]\!]_\mathsf{p}$.

(ii) $\Rightarrow$: Consider $\mathcal{E} = \mathcal{A}, \mathcal{I}_j = \mathcal{G}_j^\mathsf{n}$.
$\Leftarrow$: $(\mathcal{E} \parallel \|_{1 \le j \ne i \le n} \; \mathcal{I}_j) \le (\mathcal{A} \parallel \|_{1 \le j \ne i \le n} \; \mathcal{G}_j^\mathsf{n}) \le \mathcal{A}_i$. □

The composition of a set of contracts is the least contract that dominates the set. In the following, we provide the exact formulation and then prove its properties. Our definition is an extension of the one proposed by Bauer et al. [2012], which defines composition for a pair of contracts only.

*Definition* 5.3. The composition of a set of contracts $\{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$, written $\bigodot_{1 \le i \le n} \mathcal{C}_i$, is the contract

$$\mathcal{C} = (\mathcal{A}, \mathcal{G}) = \left( \bigwedge_{1 \le i \le n} \left( \mathcal{A}_i / \|_{1 \le k \ne i \le n} \; \mathcal{G}_k^\mathsf{n} \right), \|_{1 \le j \le n} \; \mathcal{G}_j^\mathsf{n} \right).$$

Let contracts $\mathcal{C}_i, \mathcal{C}_i'$ be such that $\mathcal{C}_i' \prec \mathcal{C}_i$. The following theorem generalizes several important results that were established for $n = 2$ [Bauer et al. 2012]:

—The composition of a set of contracts dominates the individual contracts (Theorem 5.4 (i)) and is the *least*, in the refinement order, of all contracts dominating them (Theorem 5.4 (ii)).

—Dominance is preserved under the refinement operation of contracts (Theorem 5.4 (iiia)).
—Contract refinement is preserved under contract composition (Theorem 5.4 (iiib)).

As a result, we can derive a generic contract theory with an $n$-ary composition for contracts, lifting the compositional design to a set of $n$ components.

THEOREM 5.4. *Let $\mathcal{C}$ be the composition of $\{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$, then*

(*i*) *$\mathcal{C}$ dominates the contract set $\{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$.*
(*ii*) *$\forall \mathcal{C}' : \mathcal{C}'$ dominates $\{\mathcal{C}_1, \ldots, \mathcal{C}_n\} \Leftrightarrow \mathcal{C} \prec \mathcal{C}'$.*
(*iii*) *If $\mathcal{C}'$ dominates $\{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$, then*
  (*a*) *it dominates also $\{\mathcal{C}'_1, \ldots, \mathcal{C}'_n\}$ where $\mathcal{C}'_i \prec \mathcal{C}_i$,*
  (*b*) $(\bigodot_{1\leq i\leq n} \mathcal{C}'_i) \prec (\bigodot_{1\leq i\leq n} \mathcal{C}_i)$.

PROOF. Let $\mathcal{A}'_h$ be defined as follows:

$$\mathcal{A}'_h \stackrel{\text{def}}{=} \mathcal{A}_h / \Big|\Big|_{1\leq k\neq h\leq n} \mathcal{G}^{\mathfrak{n}}_k,$$

then $\mathcal{A} = \curlywedge_{1\leq h\leq n} \mathcal{A}'_h$ and it follows that $\mathcal{A} \leq \mathcal{A}'_h$.

(i) $\mathcal{C}$ dominates $\{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$ because
  (a) $||_{1\leq i\leq n} \mathcal{I}_i \in [\![\mathcal{C}]\!]_{\mathsf{p}}$, by Theorem 5.2 and $\mathcal{G} \in [\![\mathcal{C}]\!]_{\mathsf{p}}$.
  (b) By Theorem 5.2 and by the quotient property,

$$\mathcal{A} \parallel \Big|\Big|_{1\leq j\neq i\leq n} \mathcal{G}^{\mathfrak{n}}_j \leq \mathcal{A}'_i \parallel \Big|\Big|_{1\leq j\neq i\leq n} \mathcal{G}^{\mathfrak{n}}_j \leq \left(\mathcal{A}_i / \Big|\Big|_{1\leq k\neq i\leq n} \mathcal{G}^{\mathfrak{n}}_k\right) \parallel \Big|\Big|_{1\leq j\neq i\leq n} \mathcal{G}^{\mathfrak{n}}_j \leq \mathcal{A}_i.$$

(ii) —⇒: By the dominance of $\mathcal{C}'$ over $\{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$ and by Theorem 5.2, we have

$$\mathcal{A}' \parallel \Big|\Big|_{1\leq j\neq i\leq n} \mathcal{G}^{\mathfrak{n}}_j \leq \mathcal{A}_i \Rightarrow \mathcal{A}' \leq \mathcal{A}_i / \Big|\Big|_{1\leq j\neq i\leq n} \mathcal{G}^{\mathfrak{n}}_j.$$

$$\Rightarrow \mathcal{A}' \leq \curlywedge_{1\leq i\leq n} \left(\mathcal{A}_i / \Big|\Big|_{1\leq j\neq i\leq n} \mathcal{G}^{\mathfrak{n}}_j\right).$$

$$\Rightarrow \mathcal{A}' \leq \curlywedge_{1\leq i\leq n} \mathcal{A}'_i.$$

This means that $\mathcal{A}' \leq \mathcal{A}$ and implies that $[\![\mathcal{C}']\!]_{\mathsf{e}} \subseteq [\![\mathcal{C}]\!]_{\mathsf{e}}$, which in turn implies that

$$\forall \mathcal{E}' \in [\![\mathcal{C}']\!]_{\mathsf{e}} : \mathcal{E}' \in [\![\mathcal{C}]\!]_{\mathsf{e}}.$$

We also have $\mathcal{I} \in [\![\mathcal{C}]\!]_{\mathsf{p}}$, which means that

$$\forall \mathcal{E} \in [\![\mathcal{C}]\!]_{\mathsf{e}} : \mathcal{I} \parallel \mathcal{E} \leq \mathcal{G} \parallel \mathcal{E}.$$

By the dominance of $\mathcal{C}'$ over $\{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$, the following is true:

$$\mathcal{G} \in [\![\mathcal{C}']\!]_{\mathsf{p}} \Rightarrow \forall \mathcal{E}' \in [\![\mathcal{C}']\!]_{\mathsf{e}} : \mathcal{G} \parallel \mathcal{E}' \leq \mathcal{G}' \parallel \mathcal{E}'.$$

Combining all of the preceding together, we have

$$\begin{aligned}
&\forall \mathcal{E} \in [\![\mathcal{C}]\!]_{\mathsf{e}} : \mathcal{I} \parallel \mathcal{E} \leq \mathcal{G} \parallel \mathcal{E} \\
&\quad \Rightarrow \ \forall \mathcal{E}' \in [\![\mathcal{C}']\!]_{\mathsf{e}} : \mathcal{I} \parallel \mathcal{E}' \leq \mathcal{G} \parallel \mathcal{E}' \\
&\quad \Rightarrow \ \forall \mathcal{E}' \in [\![\mathcal{C}']\!]_{\mathsf{e}} : \mathcal{I} \parallel \mathcal{E}' \leq \mathcal{G}' \parallel \mathcal{E}' \\
&\quad \Rightarrow \ \mathcal{I} \in [\![\mathcal{C}']\!]_{\mathsf{p}}.
\end{aligned}$$

This implies that $[\![\mathcal{C}]\!]_{\mathsf{p}} \subseteq [\![\mathcal{C}']\!]_{\mathsf{p}}$, and therefore $\mathcal{C} \prec \mathcal{C}'$.
—⇐: The refinement relation $\mathcal{C} \prec \mathcal{C}'$ means that

$$([\![\mathcal{C}]\!]_{\mathsf{p}} \subseteq [\![\mathcal{C}']\!]_{\mathsf{p}}) \wedge ([\![\mathcal{C}']\!]_{\mathsf{e}} \subseteq [\![\mathcal{C}]\!]_{\mathsf{e}}).$$

Since $\mathcal{G} \in [\![\mathcal{C}]\!]_{\mathsf{p}}$ and $[\![\mathcal{C}]\!]_{\mathsf{p}} \subseteq [\![\mathcal{C}']\!]_{\mathsf{p}}$, we then have

$$\Big|\Big| \limits_{1 \leq i \leq n} \mathcal{G}_i^{\mathfrak{n}} \in [\![\mathcal{C}']\!]_{\mathsf{p}}.$$

In addition,

$$[\![\mathcal{C}']\!]_{\mathsf{e}} \subseteq [\![\mathcal{C}]\!]_{\mathsf{e}} \Rightarrow \mathcal{A}' \leq \mathcal{A} \Rightarrow \mathcal{A}' \leq \mathcal{A}_i^{/} \Rightarrow \mathcal{A}' \leq \mathcal{A}_i / \Big|\Big| \limits_{1 \leq k \neq i \leq n} \mathcal{G}_k^{\mathfrak{n}}$$

$$\Rightarrow \left( \mathcal{A}' \parallel \Big|\Big| \limits_{1 \leq k \neq i \leq n} \mathcal{G}_k^{\mathfrak{n}} \right) \leq \mathcal{A}_i.$$

By Theorem 5.2, $\mathcal{C}'$ then dominates $\{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$.

(iii) (a) First,

$$\mathcal{C}_i' \prec \mathcal{C}_i \Rightarrow [\![\mathcal{C}_i']\!]_{\mathsf{p}} \subseteq [\![\mathcal{C}_i]\!]_{\mathsf{p}} \Rightarrow \mathcal{I}_i' \in [\![\mathcal{C}_i]\!]_{\mathsf{p}} \Rightarrow \Big|\Big| \limits_{1 \leq i \leq n} \mathcal{I}_i' \in [\![\mathcal{C}']\!]_{\mathsf{p}}$$

(the last implication is because of the dominance of $\mathcal{C}'$ over $\{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$). Second,

$$\begin{aligned}
\mathcal{C}_i' \prec \mathcal{C}_i &\Rightarrow \mathcal{G}_i'^{\mathfrak{n}} \leq \mathcal{G}_i^{\mathfrak{n}} \\
&\Rightarrow \mathcal{A}' \parallel \Big|\Big| \limits_{1 \leq j \neq i \leq n} \mathcal{G}_j'^{\mathfrak{n}} \leq \mathcal{A}' \parallel \Big|\Big| \limits_{1 \leq j \neq i \leq n} \mathcal{G}_j^{\mathfrak{n}} \\
&\Rightarrow \mathcal{A}' \parallel \Big|\Big| \limits_{1 \leq j \neq i \leq n} \mathcal{G}_j'^{\mathfrak{n}} \leq \mathcal{A}_i \leq \mathcal{A}_i'
\end{aligned}$$

(the last implication is because of $\mathcal{C}_i' \prec \mathcal{C}_i$).
By Theorem 5.2, $\mathcal{C}'$ thus dominates $\{\mathcal{C}_1', \ldots, \mathcal{C}_n'\}$.
(b) A direct consequence of items (i), (ii), and (iiia) of Theorem 5.4. □

### 5.2. Contract Decomposition

*Definition* 5.5. A set of contracts $\{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$ is said to be a *decomposition* of a contract $\mathcal{C}$ if and only if the composition of the contract set $\bigodot_{1 \leq i \leq n} \mathcal{C}_i$ refines the contract $\mathcal{C}$.

As a direct consequence of Theorem 5.4 (ii), contract $\mathcal{C}$ can be decomposed into the set of contracts $\{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$ if and only if $\mathcal{C}$ dominates the contract set. This dominance subsequently provides desirable compositional features for decomposing contract-based systems. These features have been formalized in items (i) and (ii) of Definition 5.1. Item (i), in particular, describes the feature of being able to replace any existing component implementing contract $\mathcal{C}_i$ with another component satisfying the same contract

in a system required to implement contract $\mathcal{C}$. This compositional feature indeed enables such replacement of components to take place without breaking the contract satisfaction of the overall system.

Verifying if $\mathcal{C}$ can be decomposed into $\{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$ is therefore equivalent to checking the dominance of $\mathcal{C}$ over $\{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$, which, by Theorem 5.2, corresponds to the two decomposing conditions (**DC**s):

$$\textbf{(DC-1)} \quad \Big|\Big| \limits_{1 \le i \le n} \mathcal{G}_i^{\mathrm{n}} \in [\![\mathcal{C}]\!]_{\mathsf{p}}, \text{ or equivalently } \Big|\Big| \limits_{1 \le i \le n} \mathcal{G}_i^{\mathrm{n}} \le \mathcal{G}^{\mathrm{n}}$$

$$\textbf{(DC-2)} \; \forall 1 \le i \le n : \mathcal{A} \parallel \Big|\Big| \limits_{1 \le j \ne i \le n} \mathcal{G}_j^{\mathrm{n}} \le \mathcal{A}_i.$$

Moreover, our extension on the dominance notion is more generic than that of Cimatti and Tonetta [2012] and can support the construction of property-based proof systems such as that proposed by the same authors. In fact, we built our system in a generic way using a set of metatheoretical operators including composition, refinement, quotient, and conjunction. Our extension can therefore be applied to build proof systems of different contract frameworks where these operators are explicitly instantiated. For example, trace-based contract system development [Cimatti and Tonetta 2012] can be derived by instantiating the composition and refinement between specifications as the intersection and set inclusion as follows:

(i) $\bigcap \limits_{1 \le i \le n} \mathcal{G}_i^{\mathrm{n}} \in [\![\mathcal{C}]\!]_{\mathsf{p}}$, or equivalently $\bigcap \limits_{1 \le i \le n} \mathcal{G}_i^{\mathrm{n}} \subseteq \mathcal{G}^{\mathrm{n}}$

(ii) $\forall 1 \le i \le n : \mathcal{A} \cap \bigcap \limits_{1 \le j \ne i \le n} \mathcal{G}_j^{\mathrm{n}} \subseteq \mathcal{A}_i.$

Likewise, modal contract system development can be based on the modal alternating refinement $\le_m$ and the modal composition $\|_m$ on shared actions [Bauer et al. 2012]. These concepts will be restated and elaborated in more detail later in Examples 7.1 and 8.2.

At the same time, our method is not limited to these models. In particular, type systems or more complex architecture-oriented models where components are mappings from a service to a service, and where composition is defined simply as function composition [Marmsoler et al. 2015], can be adapted to satisfy our assumptions. These extensions are part of our future work.

## 6. CONTRACT SYNTHESIS

When a set of contracts does not satisfy the decomposing conditions established in Section 5.2, we must adjust the specification of some of them. We propose a synthesis strategy based on the following assumption, which says that the conjunction operator can be distributed over the normalizing operator $\triangleright$ as follows:

$$(\mathcal{G} \triangleright \mathcal{A}) \curlywedge X = (\mathcal{G} \curlywedge X) \triangleright (\mathcal{A} \triangleright X) \tag{1}$$

Although this condition does not hold in general and as a result poses certain limitations on contract systems, it is a desirable property because it shows that the semantics of a model is invariant when commuting (appropriately) normalization $\triangleright$ and conjunction $\curlywedge$. Better flexibility in the design process can also be gained when these operators can somehow be interchanged. Since conjunction and normalization amount to strengthening and weakening operations, respectively, strengthening $X$ causes a semantic reduction in the two sides of Equation (1). Thus, when this property does not hold, we can keep strengthening $X$ until we reach a fixed point in semantic equivalence, as we shall see later in Section 8.

Contract synthesis consists of finding suitable refinements for the individual contracts. Our synthesis strategy is based on strengthening the normalized guarantees, which can be reduced to strengthening the unnormalized guarantees and weakening the corresponding assumptions. Because such operations either strengthen the left sides or weaken the right sides of the decomposing conditions, their refinement relation are either maintained or changed from false to true.

### 6.1. Synthesis of Decomposing Conditions

*6.1.1. Condition 1.* Assume that **DC-1** is not satisfied. Then, according to our formulation, the composition of the normalized guarantees of the components does not refine the normalized guarantee of the specification. This condition can be corrected by selecting and strengthening any of the guarantee $\mathcal{G}_k^{\mathrm{n}}$ that occur in **DC-1**. To do this, we take advantage of the quotient operator, which, by virtue of its definition, gives us the least constrained specification $X$:

$$X = \mathcal{G}^{\mathrm{n}} / \left( \Big\| \limits_{1 \le i \ne k \le n} \mathcal{G}_i^{\mathrm{n}} \right),$$

which ensures the satisfaction of **DC-1**. At the same time, $X$ is not necessarily a valid replacement for $\mathcal{G}_k^{\mathrm{n}}$, as it does not take the component specification into account. Instead, the newly strengthened normalized guarantee $\bar{\mathcal{G}}_k^{\mathrm{n}}$, must be computed as the weakest specification that satisfies both $X$ and the original guarantee, i.e., it is the conjunction:

$$\bar{\mathcal{G}}_k^{\mathrm{n}} = \mathcal{G}_k^{\mathrm{n}} \curlywedge X = (\mathcal{G}_k \rhd \mathcal{A}_k) \curlywedge X = (\mathcal{G}_k \curlywedge X) \rhd (\mathcal{A}_k \rhd X). \tag{2}$$

Since conjunction and normalization amount to strengthening and weakening operations, respectively, the preceding equation shows that strengthening a normalized guarantee amounts to strengthening its unnormalized version and weakening its coupled assumption. Overall, it amounts to refining the contract $\mathcal{C}_k$.

*6.1.2. Condition 2.* To satisfy the $i$-th clause of **DC-2**, we select a guarantee $\mathcal{G}_{k_i}^{\mathrm{n}}$ to be strengthened where $\mathcal{G}_{k_i}^{\mathrm{n}}$ can be any of the guarantees $\mathcal{G}_j^{\mathrm{n}}$ composing the $i$-th clause of **DC-2** and $k_i \ne i$. Similar to the synthesis of the first condition, we need to find the least constrained specification $Y_i$:

$$Y_i = \mathcal{A}_i / \left( \mathcal{A} \parallel \Big\| \limits_{1 \le j \ne i, j \ne k_i \le n} \mathcal{G}_j^{\mathrm{n}} \right),$$

which ensures the satisfaction of the $i$-th clause. As done for condition 1, $\mathcal{G}_{k_i}^{\mathrm{n}}$ is then strengthened to $\bar{\mathcal{G}}_{k_i}^{\mathrm{n}} \stackrel{\mathrm{def}}{=} \mathcal{G}_{k_i}^{\mathrm{n}} \curlywedge Y_i$:

$$\bar{\mathcal{G}}_{k_i}^{\mathrm{n}} = \mathcal{G}_{k_i}^{\mathrm{n}} \curlywedge Y_i = (\mathcal{G}_{k_i} \rhd \mathcal{A}_{k_i}) \curlywedge Y_i = (\mathcal{G}_{k_i} \curlywedge Y_i) \rhd (\mathcal{A}_{k_i} \rhd Y_i). \tag{3}$$

### 6.2. Synthesis Strategy

Based on Equation (1) and the preceding analysis, we can apply two different synthesis strategies. The first, in Algorithm 1, is called *aggressive* since it aggressively fixes a false decomposition clause as soon as possible, which can be very helpful in a distributed context. In fact, each team can try to synthesize their own component contract without waiting for other teams' synthesis update. This capability of performing independent synthesis is indeed enabled by basing the synthesis on the original status of other contracts in the system, namely $\mathcal{C}$ and $\{\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_n\}$.

**ALGORITHM 1:** Aggressive Synthesis Strategy

**Input**: Contracts $\{\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_n\}$ not being a decomposition of contract $\mathcal{C}$.
**Output**: Contracts $\{\bar{\mathcal{C}}_1, \bar{\mathcal{C}}_2, \ldots, \bar{\mathcal{C}}_n\}$ being a decomposition of contract $\mathcal{C}$.

```
// Checking the condition DC-1...
```
$k = -1$;
**if** $\Big( \big\|_{1 \le j \le n} \mathcal{G}_j^{\mathrm{n}} \not\leq \mathcal{G}^{\mathrm{n}} \Big)$ **then**
  $k = 1$; `// alternatively` $k$ `=Random(`$1, n$`) if some random function is available`
  $X = \mathcal{G}^{\mathrm{n}} / \big( \big\|_{1 \le j \ne k \le n} \mathcal{G}_j^{\mathrm{n}} \big)$;
**end**
```
// Checking the condition DC-2...
```
**for** $i = 1$ *to* $n$ **do**
  $p = (i + 1) \bmod n$;
  **if** $\big(\mathcal{A} \parallel \big\|_{1 \le j \ne i \le n} \mathcal{G}_j^{\mathrm{n}} \not\leq \mathcal{A}_i\big)$ **then**
    `// Synthesize contract` $\mathcal{C}_p$
    $Y_i = \mathcal{A}_i / \big(\mathcal{A} \parallel \big\|_{1 \le j \ne i, j \ne p \le n} \mathcal{G}_j^{\mathrm{n}}\big)$;
    **if** $(p == k)$ **then**
      $\bar{\mathcal{G}}_p = ((\mathcal{G}_p \curlywedge X) \curlywedge Y_i)$;
      $\bar{\mathcal{A}}_p = ((\mathcal{A}_p \rhd X) \rhd Y_i)$;
    **else**
      $\bar{\mathcal{G}}_p = (\mathcal{G}_p \curlywedge Y_i)$;
      $\bar{\mathcal{A}}_p = (\mathcal{A}_p \rhd Y_i)$;
    **end**
  **else**
    `// No synthesis effort is necessary, copy contract` $\mathcal{C}_p$
    **if** $(p == k)$ **then**
      $\bar{\mathcal{G}}_p = \mathcal{G}_p \curlywedge X$;
      $\bar{\mathcal{A}}_p = \mathcal{A}_p \rhd X$;
    **else**
      $\bar{\mathcal{G}}_p = \mathcal{G}_p$; $\bar{\mathcal{A}}_p = \mathcal{A}_p$;
    **end**
  **end**
**end**

The idea of the strategy is to select a different contract at every step. Since our strategy may go through $n + 1$ synthesis steps in the worst case, while there are $n$ contracts to be synthesized, some contract may be synthesized twice. At first, a contract $\mathcal{C}_k$ is chosen for synthesis of **DC-1**, i.e., the value of $k$, which was initialized to $-1$, turns positive if **DC-1** is not met. In this phase, we compute the set $X$; however, the effective synthesis on $\mathcal{C}_k$ is deferred to a later step. Next, every contract $\mathcal{C}_{(i+1) \bmod n}$ is to be synthesized if the $i$-th clause of **DC-2** is not met. If $(i + 1) \bmod n$ is exactly $k$, then the synthesis deferred earlier must also be performed. All other contracts may be synthesized at most once.

The second strategy is a less aggressive or *incremental* version of the strategy where false decomposition clauses are fixed one after another, as shown in Algorithm 2. After a contract has been synthesized, it is used to check whether the other false decomposition clauses have been rectified. Therefore, a contract can be synthesized as many times as required. The strategy keeps going on as long as there is still a false decomposition clause that needs rectifying. This algorithm does not have to keep a reminder of a

**ALGORITHM 2:** Incremental Synthesis Strategy

**Input**: Contracts $\{\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_n\}$ not being a decomposition of contract $\mathcal{C}$.
**Output**: Contracts $\{\bar{\mathcal{C}}_1, \bar{\mathcal{C}}_2, \ldots, \bar{\mathcal{C}}_n\}$ being a decomposition of contract $\mathcal{C}$.

// Copy all contracts $\bar{\mathcal{C}}_{0t} = \mathcal{C}_t \ldots$
**for** $t = 1$ *to* $n$ **do**
  $\bar{\mathcal{G}}_{0t} = \mathcal{G}_t$; $\bar{\mathcal{A}}_{0t} = \mathcal{A}_t$;
**end**
// Checking the condition **DC-1**...
**if** $(\ \|_{1\le j\le n} \mathcal{G}_j^{\mathrm{n}} \not\leq \mathcal{G}^{\mathrm{n}})$ **then**
  $k = 1$; // alternatively $k$ =Random$(1, n)$ if some random function is available
  $X = \mathcal{G}^{\mathrm{n}} / (\ \|_{1\le j\neq k\le n} \mathcal{G}_j^{\mathrm{n}})$;
  $\bar{\mathcal{G}}_{0k} = (\mathcal{G}_k \swarrow X)$;
  $\bar{\mathcal{A}}_{0k} = (\mathcal{A}_k \triangleright X)$;
**end**
// Checking the condition **DC-2**...
**for** $i = 1$ *to* $n$ **do**
  $p = (i + 1) \bmod n$;
  // Copy all contracts $\bar{\mathcal{C}}_{it} = \bar{\mathcal{C}}_{(i-1)t} \ldots$
  **for** $t = 1$ *to* $n$ **do**
    $\bar{\mathcal{G}}_{it} = \bar{\mathcal{G}}_{(i-1)t}$; $\bar{\mathcal{A}}_{it} = \bar{\mathcal{A}}_{(i-1)t}$;
  **end**
  **if** $(\mathcal{A} \parallel \|_{1\le j\neq i\le n} \bar{\mathcal{G}}_{ij}^{\mathrm{n}} \not\leq \bar{\mathcal{A}}_{ii})$ **then**
    // Synthesize contract $\mathcal{C}_p$
    $Y_i = \bar{\mathcal{A}}_{ii} / (\mathcal{A} \parallel \|_{1\le j\neq i, j\neq p\le n} \bar{\mathcal{G}}_{ij}^{\mathrm{n}})$;
    $\bar{\mathcal{G}}_{ip} = (\bar{\mathcal{G}}_{ip} \swarrow Y_i)$;
    $\bar{\mathcal{A}}_{ip} = (\bar{\mathcal{A}}_{ip} \triangleright Y_i)$;
  **else**
    // No synthesis effort is necessary, copy contract $\mathcal{C}_p$
    $\bar{\mathcal{G}}_{ip} = \bar{\mathcal{G}}_{ip}$; $\bar{\mathcal{A}}_{ip} = \bar{\mathcal{A}}_{ip}$;
  **end**
**end**
// Copy all contracts $\bar{\mathcal{C}}_t = \bar{\mathcal{C}}_{nt} \ldots$
**for** $t = 1$ *to* $n$ **do**
  $\bar{\mathcal{G}}_t = \bar{\mathcal{G}}_{nt}$; $\bar{\mathcal{A}}_t = \bar{\mathcal{A}}_{nt}$;
**end**

deferred synthesis as its counterpart, as every synthesis is done immediately upon detecting dissatisfaction of any **DC**. The strategy is useful in the synchronous context where distributed teams need to synchronize with each other at every synthesis step. The purpose of synchronization is to avoid or reduce two problems:

—Unnecessary synthesis effort since the decomposition conditions may be all satisfied with less than $m$ synthesis steps, where $m$ is the number of false decomposition conditions.
—Quotient operations, which can be costly in some contract systems. For example, in modal contract systems, the quotient involves computing the complement of the active alphabet for every transition in modal contract systems. In trace-based contract systems, the quotient involves the complement set operation.

*Computation complexity*. Although both strategies perform $n+1$ satisfaction checks of decomposition conditions and $m$ quotient operations in the worst case, the incremental strategy can show better performance with fewer satisfaction checks and quotient operations than the aggressive one, in general. This is because the latter performs *exactly* $n+1$ satisfaction checks and *exactly* $m$ quotient operations, whereas the former performs the same number of condition verification and *possibly* less than $m$ quotients, where $m$ is the number of false decomposition conditions.

### 6.3. Soundness and Completeness of Synthesis Strategy

The synthesis strategies proposed in Section 6.2 are *sound*, meaning that the set of newly refined contracts is always a valid decomposition of contract $\mathcal{C}$. We show the soundness of the aggressive strategy in Theorem 6.1. The incremental strategy's soundness will then follow from this theorem.

Because we compute quotients when determining the stronger contracts, our strategies are able to determine the *most general* specification that satisfies the decomposition conditions. Therefore, in some sense, the algorithms achieve a certain degree of optimality. Nevertheless, since our strategy proposes some heuristic directions to synthesize the contract set, it is not *complete* in general, meaning that it does not explore all possible decompositions of contract $\mathcal{C}$. To achieve completeness, one would have to systematically examine the design space. This is made difficult by at least two factors. First, the algorithm would need to permute the order in which contracts in a set are considered for strengthening. In the second place, one would have to budget assumptions and guarantees differently across the components. Whereas the first factor may result in a combinatorial explosion, making the strategies not effective in practical contexts, the second could potentially lead to nontermination for all but overly simplistic models. For these reasons, and given the complexity of the issues, completeness is still the subject of our current research.

In the rest of the section, we prove the soundness of our synthesis algorithms.

THEOREM 6.1. *Let* $\{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$ *be a set of contracts that is not a decomposition of contract* $\mathcal{C}$. *Let* $\{\bar{\mathcal{C}}_1, \ldots, \bar{\mathcal{C}}_n\}$ *denote the synthesized version of the contract set in the aggressive strategy where for any* $1 \leq i \leq n$, *either* $\bar{\mathcal{C}}_i \equiv \mathcal{C}_i$ *(i.e., the contract* $\mathcal{C}_i$ *is not synthesized) or* $\bar{\mathcal{C}}_i \leq \mathcal{C}_i$ *(i.e., the contract is synthesized).*

*Then, the newly synthesized contract set* $\{\bar{\mathcal{C}}_1, \ldots, \bar{\mathcal{C}}_n\}$ *is a decomposition of contract* $\mathcal{C}$.

PROOF. When a contract is synthesized, it can affect the logical values of the predicates specified in the decomposing conditions where the contract is involved. The effect is positive in the sense that it helps rectifying some false predicates while maintaining the true ones. Therefore, to prove the theorem, we show that:

—a predicate that is falsified by the original contract set $\{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$ is rectified by the synthesized one $\{\bar{\mathcal{C}}_1, \ldots, \bar{\mathcal{C}}_n\}$, and
—a predicate that is certified by the original contract set remains certified by the synthesized one.

This is indeed equivalent to showing that the two decomposing conditions hold for the synthesized contract set $\{\bar{\mathcal{C}}_1, \ldots, \bar{\mathcal{C}}_n\}$.

(a) If condition **DC-1** holds for the contract set $\{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$, i.e.,

$$\left( \Big\| \underset{1 \leq i \leq n}{} \mathcal{G}_i^{\mathrm{n}} \right) \leq \mathcal{G}^{\mathrm{n}}, \tag{4}$$

then it holds also for the synthesized contract set $\{\bar{\mathcal{C}}_1, \ldots, \bar{\mathcal{C}}_n\}$ because

$$\left(\bigparallel_{1\le i\le n} \bar{\mathcal{G}}_i^{\mathfrak{n}}\right) \le \left(\bigparallel_{1\le i\le n} \mathcal{G}_i^{\mathfrak{n}}\right) \le \mathcal{G}^{\mathfrak{n}}.$$

The first refinement is because guarantees have always been strengthened, i.e.,

$$\bar{\mathcal{G}}_i^{\mathfrak{n}} \le \mathcal{G}_i.$$

The second refinement is because of the assumption (4).

(b) If condition **DC-1** is falsified by the contract set $\{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$, then it is rectified by the contract set $\{\bar{\mathcal{C}}_1, \ldots, \bar{\mathcal{C}}_n\}$. Assuming that $\mathcal{C}_k$ is synthesized,

$$\begin{aligned}
\bigparallel_{1\le i\le n} \bar{\mathcal{G}}_i^{\mathfrak{n}} = \left(\bigparallel_{1\le i\ne k\le n} \bar{\mathcal{G}}_i^{\mathfrak{n}}\right) \parallel \bar{\mathcal{G}}_k^{\mathfrak{n}} &\le \left(\bigparallel_{1\le i\ne k\le n} \bar{\mathcal{G}}_i^{\mathfrak{n}}\right) \parallel \left(\mathcal{G}_k^{\mathfrak{n}} \between X\right) \\
&\le \left(\bigparallel_{1\le i\ne k\le n} \mathcal{G}_i^{\mathfrak{n}}\right) \parallel X \\
&\le \left(\bigparallel_{1\le i\ne k\le n} \mathcal{G}_i^{\mathfrak{n}}\right) \parallel \left(\mathcal{G}^{\mathfrak{n}} \Big/ \left(\bigparallel_{1\le i\ne k\le n} \mathcal{G}_i^{\mathfrak{n}}\right)\right) \\
&\le \mathcal{G}^{\mathfrak{n}}.
\end{aligned}$$

The first refinement is because guarantees have always been strengthened and the second by definition of conjunction:

$$\begin{aligned}
\bar{\mathcal{G}}_i^{\mathfrak{n}} &\le \mathcal{G}_i, \\
\left(\mathcal{G}_k^{\mathfrak{n}} \between X\right) &\le X.
\end{aligned}$$

The third refinement is because by construction,

$$X = \mathcal{G}^{\mathfrak{n}} / \left(\bigparallel_{1\le i\ne k\le n} \mathcal{G}_i^{\mathfrak{n}}\right).$$

The last refinement is because of the definition of quotient.

(c) Any condition **DC-2*i*** that is falsified by the contract set $\{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$ can now be rectified by the contract set $\{\bar{\mathcal{C}}_1, \ldots, \bar{\mathcal{C}}_n\}$. This is because

$$\begin{aligned}
\mathcal{A} \parallel \left(\bigparallel_{1\le j\ne i\le n} \bar{\mathcal{G}}_j^{\mathfrak{n}}\right) &\le \mathcal{A} \parallel \left(\bigparallel_{1\le j\ne i, j\ne k_i\le n} \bar{\mathcal{G}}_j^{\mathfrak{n}}\right) \parallel \left(\mathcal{G}_{k_i}^{\mathfrak{n}} \between Y_i\right) \\
&\le \mathcal{A} \parallel \left(\bigparallel_{1\le j\ne i, j\ne k_i\le n} \mathcal{G}_j^{\mathfrak{n}}\right) \parallel Y_i \\
&\le \mathcal{A} \parallel \left(\bigparallel_{1\le j\ne i, j\ne k_i\le n} \mathcal{G}_j^{\mathfrak{n}}\right) \parallel \left(\mathcal{A}_i \Big/ \left(\mathcal{A} \parallel \bigparallel_{1\le j\ne i, j\ne k_i\le n} \mathcal{G}_j^{\mathfrak{n}}\right)\right) \\
&\le \mathcal{A}_i \\
&\le \bar{\mathcal{A}}_i,
\end{aligned}$$

where $k_i = (i+1) \bmod n$. The first refinement is because of guarantees have always been strengthened and the second by definition of conjunction. The third refinement is because by construction,

$$Y_i = \mathcal{A}_i / \left( \mathcal{A} \parallel \Big\| _{1 \le j \ne i, j \ne k_i \le n} \mathcal{G}_j^{\mathfrak{n}} \right).$$

The fourth refinement is because of the definition of quotient. The last refinement is because assumptions have always been maintained or weakened.

(d) Any condition **DC-2***i* enabled by the contract set $\{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$, i.e.,

$$\mathcal{A} \parallel \left( \Big\| _{1 \le j \ne i \le n} \mathcal{G}_j^{\mathfrak{n}} \right) \le \mathcal{A}_i, \tag{5}$$

is also enabled by the contract set $\{\bar{\mathcal{C}}_1, \ldots, \bar{\mathcal{C}}_n\}$ since

$$\mathcal{A} \parallel \left( \Big\| _{1 \le j \ne i \le n} \bar{\mathcal{G}}_j^{\mathfrak{n}} \right) \le \mathcal{A} \parallel \left( \Big\| _{1 \le j \ne i \le n} \mathcal{G}_j^{\mathfrak{n}} \right) \le \mathcal{A}_i \le \bar{\mathcal{A}}_i.$$

The first refinement is because guarantees have always been strengthened and the second refinement is due to the refinement (5). The last refinement is true since assumptions have always been maintained or weakened. □

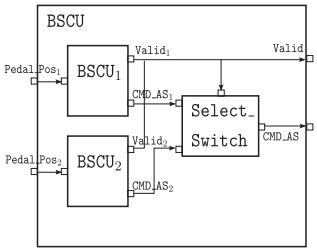
To prove the soundness of the incremental strategy, we use a line of reasoning similar to the preceding proof where items (a), (b), and (d) are in the exact same way. For item (c), we must introduce a minor modification to contract index, as follows:

$$\begin{aligned}
\mathcal{A} \parallel \left( \Big\| _{1 \le j \ne i \le n} \bar{\mathcal{G}}_j^{\mathfrak{n}} \right) &= \mathcal{A} \parallel \left( \Big\| _{1 \le j \ne i, j \ne k_i \le n} \bar{\mathcal{G}}_j^{\mathfrak{n}} \right) \parallel \bar{\mathcal{G}}_{k_i}^{\mathfrak{n}} \\
&= \mathcal{A} \parallel \left( \Big\| _{1 \le j \ne i, j \ne k_i \le n} \bar{\mathcal{G}}_{ij}^{\mathfrak{n}} \right) \parallel \bar{\mathcal{G}}_{ik_i}^{\mathfrak{n}} \\
&\le \mathcal{A} \parallel \left( \Big\| _{1 \le j \ne i, j \ne k_i \le n} \bar{\mathcal{G}}_{ij}^{\mathfrak{n}} \right) \parallel Y_i \\
&\le \mathcal{A} \parallel \left( \Big\| _{1 \le j \ne i, j \ne k_i \le n} \bar{\mathcal{G}}_{ij}^{\mathfrak{n}} \right) \parallel \left( \bar{\mathcal{A}}_{ii} / \left( \mathcal{A} \parallel \Big\| _{1 \le j \ne i, j \ne k_i \le n} \bar{\mathcal{G}}_{ij}^{\mathfrak{n}} \right) \right) \\
&\le \bar{\mathcal{A}}_{ii} \\
&\le \bar{\mathcal{A}}_i.
\end{aligned}$$

We next demonstrate our strategy in synthesizing trace-based and modal contract sets.

## 7. TRACE-BASED CONTRACT SYNTHESIS

In trace-based contract systems, assumptions and guarantees are considered as sets of traces (or behaviors) defined over a set of system ports (or variables). Our technique does not depend on the specific nature of the traces, which can vary according to the particular model of computation in use. In essence, every trace must assign a history




(a) High-level view of the BSCU

$$\mathcal{A} \stackrel{\text{def}}{=} \texttt{always}\,(\neg\texttt{fault\_Monitor}_1 \wedge \neg\texttt{fault\_Command}_1 \wedge \neg\texttt{fault\_Monitor}_2) \vee$$
$$\texttt{always}\,(\neg\texttt{fault\_Monitor}_1 \wedge \neg\texttt{fault\_Command}_1 \wedge \neg\texttt{fault\_Command}_2) \vee$$
$$\texttt{always}\,(\neg\texttt{fault\_Monitor}_1 \wedge \neg\texttt{fault\_Command}_2 \wedge \neg\texttt{fault\_Monitor}_2) \vee$$
$$\texttt{always}\,(\neg\texttt{fault\_Command}_1 \wedge \neg\texttt{fault\_Command}_1 \wedge \neg\texttt{fault\_Monitor}_2)$$
$$\mathcal{G} \stackrel{\text{def}}{=} \texttt{always}\,(\texttt{Valid}_1 \vee \texttt{Valid}_2)$$
$$\mathcal{A}_i \stackrel{\text{def}}{=} \texttt{always}\,(\neg\texttt{fault\_Monitor}_i) \wedge \texttt{always}\,(\neg\texttt{fault\_Command}_i)$$
$$\mathcal{G}_i \stackrel{\text{def}}{=} \texttt{always}\,(\neg\texttt{fault\_Monitor}_i) \wedge \texttt{always}\,(\neg\texttt{fault\_Command}_i) \Longrightarrow \texttt{always}\,(\texttt{Valid}_i)$$

(b) Contract specification of the BSCU and $\texttt{BSCU}_i$

Fig. 1. Structure and contract models of the BSCU.

of values to ports. In the particular case of the following example, traces are *sequences* of simultaneous assignments of values to the set of ports, resulting in a synchronous model. For a finite representation, these sets can be conveniently specified as the traces that satisfy a logical formula or are recognized by an automaton. In this set-theoretic setting, refinement is defined as set inclusion, composition and conjunction are set intersection, and normalization, which is identical to quotient, is defined as

$$\mathcal{G} \rhd \mathcal{A} = \mathcal{G}/\mathcal{A} = \mathcal{G} \cup \neg\mathcal{A},$$

and it is easy to verify that Equation (1) is satisfied:

$$(\mathcal{G} \cup \neg\mathcal{A}) \cap X = (\mathcal{G} \cap X) \cup \neg(\mathcal{A} \cup \neg X).$$

Therefore, we can apply the preceding synthesis strategy proposed directly. It is also interesting to notice that for trace-based models, to satisfy the $i$-th clause of **DC-2**, an alternative is to weaken $\mathcal{A}_i$ to $\bar{\mathcal{A}}_i$:

$$\bar{\mathcal{A}}_i \stackrel{\text{def}}{=} \mathcal{A}_i \cup Z_i,$$

where $\mathcal{A} \cap \bigcap_{1 \le j \ne i \le n} \mathcal{G}_j^{\text{n}} \subseteq Z_i$. This operation has a nice consequence in strengthening the corresponding normalized guarantee, which is

$$\bar{\mathcal{G}}_i^{\text{n}} = \mathcal{G}_i \cup \neg(\mathcal{A}_i \cup Z_i) = \mathcal{G}_i \cup (\neg\mathcal{A}_i \cap \neg Z_i)$$

since $(\neg\mathcal{A}_i \cap \neg Z_i) \subseteq \neg\mathcal{A}_i \Rightarrow \bar{\mathcal{G}}_i^{\text{n}} \subseteq \mathcal{G}_i^{\text{n}}$.

*Example* 7.1. We consider a variant of the contract model of the Brake System Control Unit (BSCU) described in Damm et al. [2011] and shown in Figure 1(a). The BSCU controls the operation of the hydraulic braking system on the basis of the positions of the two brake pedals $\texttt{Pedal\_Pos}_1$ and $\texttt{Pedal\_Pos}_2$, as measured by a pair of sensors, and outputs the signal CMD_AS to control the braking process of a wheel-brake system. The signal Valid denotes when the control unit is malfunctioning, triggering the activation of a mechanical backup system.

The BSCU component is further decomposed into two redundant control units: a primary $\texttt{BSCU}_1$ and a backup $\texttt{BSCU}_2$, and a selector Select_Switch. When $\texttt{BSCU}_1$ fails, the Select_Switch puts the backup signal from $\texttt{BSCU}_2$ through. The signal failure in a control unit $\texttt{BSCU}_i$ is indicated by its signal $\texttt{Valid}_i$ going down and is caused by a basic fault that is either a monitor fault $\texttt{fault\_Monitor}_i$ or a command fault $\texttt{fault\_Command}_i$ with $i \in \{1, 2\}$.

A safety requirement on the BSCU is to ensure that $\texttt{Valid}_1 \vee \texttt{Valid}_2$ is always true when at most one of the basic faults $\texttt{fault\_Monitor}_i$ or $\texttt{fault\_Command}_i$ can occur [Cimatti and Tonetta 2012]. This is specified as contract $\mathcal{C} = (\mathcal{A}, \mathcal{G})$ in Figure 1(b).

The safety contracts specification [Cimatti and Tonetta 2012] on $\texttt{BSCU}_i$ make no assumptions and guarantees that signal $\texttt{Valid}_i$ remains true when neither of its basic faults occurs. In this example, we strengthen the assumption of the original safety contracts on the $\texttt{BSCU}_i$ and present them as contract $\mathcal{C}_i = (\mathcal{A}_i, \mathcal{G}_i)$ in Figure 1(b)

The contracts can be specified in symbolic logic [Cimatti and Tonetta 2012], where sets of traces are represented by logical formulas. Thus, checking the two **DC**s amounts to checking the following formulas in symbolic logic:

(i) $\bigwedge_{1 \le i \le n} \mathcal{G}_i^{\mathrm{n}} \Rightarrow \mathcal{G}^{\mathrm{n}}$

(ii) $\forall 1 \le i \le n : \mathcal{A} \wedge \bigwedge_{1 \le j \ne i \le n} \mathcal{G}_j^{\mathrm{n}} \Rightarrow \mathcal{A}_i,$

where $\mathcal{G}^{\mathrm{n}} = \mathcal{G} \vee \neg\mathcal{A}$ and $\mathcal{G}_i^{\mathrm{n}} = \mathcal{G}_i \vee \neg\mathcal{A}_i$.

To reuse the contract specification of the subcomponents $\texttt{BSCU}_1$ and $\texttt{BSCU}_2$, we verify if $\mathcal{C}$ can be decomposed into $\mathcal{C}_1$ and $\mathcal{C}_2$, which amounts to verifying the satisfaction of the two **DC**s. Although $\mathcal{C}$ can be decomposed into the subcomponents' original contracts [Cimatti and Tonetta 2012], it cannot be decomposed into $\mathcal{C}_1$ and $\mathcal{C}_2$ without refining $\mathcal{C}_i$, as we shall show next.

It is obvious that the contracts $\mathcal{C}_i$ are in normal form, and thus $\mathcal{G}_i^{\mathrm{n}} \equiv \mathcal{G}_i$. We observe that **DC-1** is satisfied because $\mathcal{G}_1 \wedge \mathcal{G}_2 \Rightarrow \mathcal{G}^{\mathrm{n}}$ is true. However, **DC-2** is not satisfied because $\mathcal{A} \wedge \mathcal{G}_1 \Rightarrow \mathcal{A}_2$ is not true. Applying our incremental synthesis strategy, we first refine $\mathcal{C}_1$ with respect to $Y_2$ into $\mathcal{C}_1' = (\mathcal{A}_1', \mathcal{G}_1')$, where

$$\begin{aligned}
\mathcal{G}_1' &= (\mathcal{G}_1 \between Y_2) = (\mathcal{G}_1 \cap Y_2),\\
\mathcal{A}_1' &= (\mathcal{A}_1 \rhd Y_2) = (\mathcal{A}_1 / Y_2) = (\mathcal{A}_1 \cup \neg Y_2),\\
Y_2 &= (\mathcal{A}_2 / \mathcal{A}) = (\mathcal{A}_2 \cup \neg\mathcal{A}) = (\mathcal{A} \Rightarrow \mathcal{A}_2).
\end{aligned}$$

**DC-2** is still not satisfied after the first synthesis because $\mathcal{A} \wedge \mathcal{G}_2 \Rightarrow \mathcal{A}_1'$ is not true. Continuing our incremental synthesis strategy, we refine $\mathcal{C}_2$ into $\mathcal{C}_2' = (\mathcal{A}_2', \mathcal{G}_2')$, where

$$\begin{aligned}
\mathcal{G}_2' &= (\mathcal{G}_2 \between Y_1) = (\mathcal{G}_2 \cap Y_1),\\
\mathcal{A}_2' &= (\mathcal{A}_2 \rhd Y_1) = (\mathcal{A}_2 / Y_1) = (\mathcal{A}_2 \cup \neg Y_1),\\
Y_1 &= (\mathcal{A}_1' / \mathcal{A}) = (\mathcal{A}_1 \cup \neg Y_2) \cup \neg\mathcal{A} = ((\mathcal{A} \wedge \mathcal{A}_2) \Rightarrow \mathcal{A}_1).
\end{aligned}$$

Alternatively, we can weaken $\mathcal{A}_i$ with respect to any $Z_i$ such that

$$(\mathcal{A} \wedge \mathcal{G}_{3-i}) \Rightarrow Z_i$$

is correct. The simplest option could be $Z_i = \texttt{TRUE}$, and this derives the original safety contracts [Cimatti and Tonetta 2012]. Our approach therefore provides a wider set of options, which allows designers to explore the refinement space.

## 8. MODAL CONTRACT SYNTHESIS

Modal contracts are defined over MTS, where transitions are annotated with action labels and with *may* or *must* modalities modeling behaviors that can be (optionally) or must be (compulsorily) implemented, respectively. Formally, an MTS is a tuple $M = (S, s_0, \Sigma, \dashrightarrow, \rightarrow)$, where $S$ is the set of states, $s_0 \in S$ is the initial state, $\Sigma$ is the set of actions, and $\dashrightarrow, \rightarrow \subseteq S \times \Sigma \times S$ are the *may*,*must* transition relation, respectively, such that $\rightarrow \subseteq \dashrightarrow$ [Raclet et al. 2011].

For the sake of comprehension, we use our notations with $m$-subscripts when referring to modal operators. The modal operators for combining modal transitions are described in Table I, where ⓤ denotes a new state in which there is a looping *may* transition for every action. This state is referred to as the *universal* state. Let $may(s_i)$ and $must(s_i)$ denote the set of *may* actions and *must* actions respectively allowed at

Table I. Rules for Combining Modal Specifications $S_1$ and $S_2$ Using Modal Operators $\rhd_m$, $\|_m$, $/_m$, $\curlywedge_m$

| $S_1 \rhd_m S_2$ | $s_2 \overset{\alpha}{\dashrightarrow} s_2'$ | $s_2 \overset{\alpha}{\dashrightarrow\!\!\!\!\!\!\rightarrow}$ |
|---|---|---|
| $s_1 \overset{\alpha}{\rightarrow} s_1'$ | $(s_1, s_2) \overset{\alpha}{\rightarrow} (s_1', s_2')$ | $(s_1, s_2) \overset{\alpha}{\dashrightarrow} ⓤ$ |
| $s_1 \overset{\alpha}{\dashrightarrow} s_1'$ | $(s_1, s_2) \overset{\alpha}{\dashrightarrow} (s_1', s_2')$ | $(s_1, s_2) \overset{\alpha}{\dashrightarrow} ⓤ$ |
| $s_1 \overset{\alpha}{\dashrightarrow\!\!\!\!\!\!\rightarrow}$ | | $(s_1, s_2) \overset{\alpha}{\dashrightarrow} ⓤ$ |

| $S_1 \|_m S_2$ | $s_2 \overset{\alpha}{\rightarrow} s_2'$ | $s_2 \overset{\alpha}{\dashrightarrow} s_2'$ |
|---|---|---|
| $s_1 \overset{\alpha}{\rightarrow} s_1'$ | $(s_1, s_2) \overset{\alpha}{\rightarrow} (s_1', s_2')$ | $(s_1, s_2) \overset{\alpha}{\dashrightarrow} (s_1', s_2')$ |
| $s_1 \overset{\alpha}{\dashrightarrow} s_1'$ | $(s_1, s_2) \overset{\alpha}{\dashrightarrow} (s_1', s_2')$ | $(s_1, s_2) \overset{\alpha}{\dashrightarrow} (s_1', s_2')$ |

| $S_1/_m S_2$ | $s_2 \overset{\alpha}{\rightarrow} s_2'$ | $s_2 \overset{\alpha}{\dashrightarrow} s_2'$ | $s_2 \overset{\alpha}{\dashrightarrow\!\!\!\!\!\!\rightarrow}$ |
|---|---|---|---|
| $s_1 \overset{\alpha}{\rightarrow} s_1'$ | $(s_1, s_2) \overset{\alpha}{\rightarrow} (s_1', s_2')$ | $(s_1, s_2)$ is inconsistent | $(s_1, s_2)$ is inconsistent |
| $s_1 \overset{\alpha}{\dashrightarrow} s_1'$ | $(s_1, s_2) \overset{\alpha}{\dashrightarrow} (s_1', s_2')$ | $(s_1, s_2) \overset{\alpha}{\dashrightarrow} (s_1', s_2')$ | $(s_1, s_2) \overset{\alpha}{\dashrightarrow} ⓤ$ |
| $s_1 \overset{\alpha}{\dashrightarrow\!\!\!\!\!\!\rightarrow}$ | | | $(s_1, s_2) \overset{\alpha}{\dashrightarrow} ⓤ$ |

| $S_1 \curlywedge_m S_2$ | $s_2 \overset{\alpha}{\rightarrow} s_2'$ | $s_2 \overset{\alpha}{\dashrightarrow} s_2'$ | $s_2 \overset{\alpha}{\dashrightarrow\!\!\!\!\!\!\rightarrow}$ |
|---|---|---|---|
| $s_1 \overset{\alpha}{\rightarrow} s_1'$ | $(s_1, s_2) \overset{\alpha}{\rightarrow} (s_1', s_2')$ | $(s_1, s_2) \overset{\alpha}{\rightarrow} (s_1', s_2')$ | $(s_1, s_2)$ is inconsistent |
| $s_1 \overset{\alpha}{\dashrightarrow} s_1'$ | $(s_1, s_2) \overset{\alpha}{\rightarrow} (s_1', s_2')$ | $(s_1, s_2) \overset{\alpha}{\dashrightarrow} (s_1', s_2')$ | |
| $s_1 \overset{\alpha}{\dashrightarrow\!\!\!\!\!\!\rightarrow}$ | $(s_1, s_2)$ is inconsistent | | |

state $s_i$. State $s_i$ is *consistent* when $must(s_i) \subseteq may(s_i)$. Combining modal systems using operators presented in Table I may introduce inconsistent states. A pruning procedure [Raclet et al. 2011] is therefore required to remedy such a problem. For the sake of completeness, we shall recall briefly this procedure.

Let $M = (S, s_0, \Sigma, \dashrightarrow, \rightarrow)$ be the newly combined system containing inconsistent states. Let $M_0 = (S, s_0, \Sigma, \dashrightarrow_0, \rightarrow_0)$ be a copy of $M$, i.e., $\dashrightarrow_0 \equiv \dashrightarrow$ and $\rightarrow_0 \equiv \rightarrow$, and let $k = 0$; we obtain the pruning of $M$ through the following steps:

(1) Let $M_{k+1} = (S, s_0, \Sigma, \dashrightarrow_{k+1}, \rightarrow_{k+1})$ be a copy of $M_k$.
(2) For each run $r = s_0 \overset{\sigma_0}{\dashrightarrow} s_1 \ldots \overset{\sigma_{n-1}}{\dashrightarrow} s_n$ from the initial state $s_0$ to state $s_n$ in $M_k$ where $must_k(s_n) \not\subseteq may_k(s_n)$,
   (i) set $may_{k+1}(s_n) = \Sigma$ and $must_{k+1}(s_n) = \emptyset$,
   (ii) set $may_{k+1}(s_{n-1}) = may_k(s_{n-1}) \setminus \{\sigma_{n-1}\}$.
(3) Set $k = k + 1$. If $M_k$ still contains inconsistent states, repeat the preceding steps. Otherwise, the procedure terminates.

The modal refinement is defined as follows [Raclet et al. 2011]. An MTS $M_1 = (S_1, s_{01}, \Sigma_1, \dashrightarrow_1, \rightarrow_1)$ refines another MTS $M_2 = (S_2, s_{02}, \Sigma_2, \dashrightarrow_2, \rightarrow_2)$, written $M_1 \leq_m M_2$, if there exists a relation $R \subseteq S_1 \times S_2$ such that $(s_{01}, s_{02}) \in R$ and for all $(s_1, s_2) \in R$ and $\alpha \in \Sigma$,

$$((s_1, \alpha, s_1') \in \dashrightarrow_1 \Rightarrow \exists (s_2, \alpha, s_2') \in \dashrightarrow_2 : (s_1', s_2') \in R) \wedge$$
$$((s_2, \alpha, s_2') \in \rightarrow_2 \Rightarrow \exists (s_1, \alpha, s_1') \in \rightarrow_1 : (s_1', s_2') \in R).$$

Consider a simple modal contract $\mathcal{C} = (\mathcal{A}, \mathcal{G})$ specified in Figure 2(a) and (b) and a specification $X$ in Figure 2(d) where the initial states are marked by bold circles. Equation (1) is shown to be violated as demonstrated in Figure 2(g) and (h). The reason is that normalization may introduce a universal state with a looping *may* transition for every action, whereas during conjunction, such universal state could be pruned away.

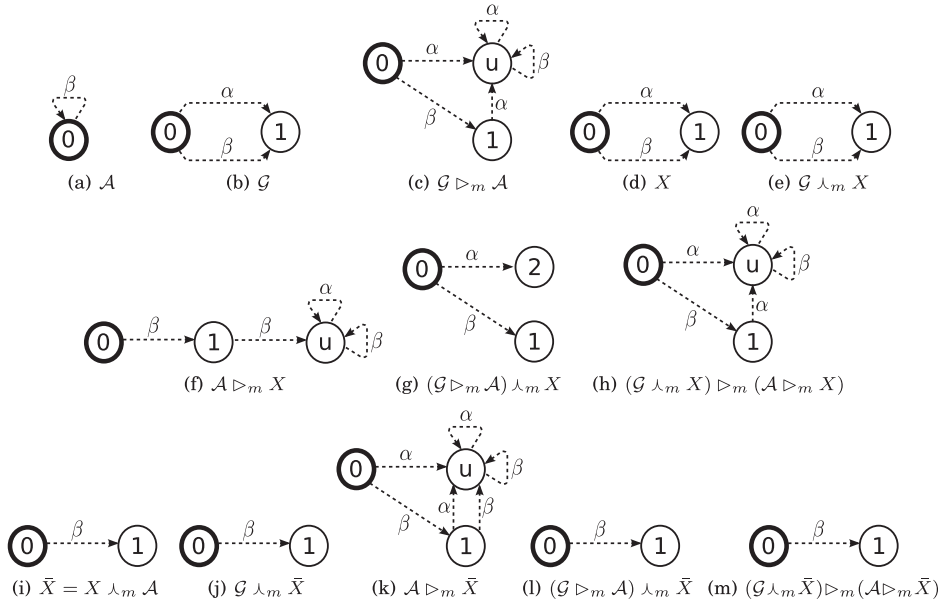



Fig. 2. A modal contract over the set of action $\Sigma = \{\alpha, \beta\}$.

To avoid such inconsistency, $\mathcal{A} \rhd_m X$ should contain all *may* transitions appearing in $X$, and we observe that it can be obtained by tightening $X$ to

$$\bar{X} \stackrel{\text{def}}{=} X \swarrow_m \mathcal{A}$$

as shown in Figure 2(l) and (m).

The following theorem affirms our preceding observation and provides a way to synthesize modal contracts. Note that as mentioned previously, a pruning procedure is invoked after every combining operation, e.g., it is invoked two times on the left side and three times on the right side of the theorem.

THEOREM 8.1. $(\mathcal{G} \rhd_m \mathcal{A}) \swarrow_m \bar{X} = (\mathcal{G} \swarrow_m \bar{X}) \rhd_m (\mathcal{A} \rhd_m \bar{X})$.

PROOF. To prove the satisfaction of Equation (1), we show that every path in $(\mathcal{G} \rhd_m \mathcal{A}) \swarrow_m \bar{X}$ can be simulated by $(\mathcal{G} \swarrow_m \bar{X}) \rhd_m (\mathcal{A} \rhd_m \bar{X})$ and vice versa.

—Let $p_l$ be a path in $(\mathcal{G} \rhd_m \mathcal{A}) \swarrow_m \bar{X}$:

$$p_l : ((g_0, a_0), \bar{x_0}) \stackrel{\alpha_0}{\dashrightarrow} ((g_1, a_1), \bar{x_1}) \ldots \stackrel{\alpha_n}{\dashrightarrow} ((g_n, a_n), \bar{x_n}).$$

Then by definition of $\swarrow_m$, there exist $p_{ga}$ in $(\mathcal{G} \rhd_m \mathcal{A})$, $p_{\bar{x}}$ in $\bar{X}$, and $p_a$ in $\mathcal{A}$:

$$p_{ga} : (g_0, a_0) \stackrel{\alpha_0}{\dashrightarrow} (g_1, a_1) \ldots \stackrel{\alpha_n}{\dashrightarrow} (g_n, a_n)$$
$$p_{\bar{x}} : \bar{x}_0 \stackrel{\alpha_0}{\dashrightarrow} \bar{x}_1 \ldots \stackrel{\alpha_n}{\dashrightarrow} \bar{x}_n$$
$$p_a : a_0 \stackrel{\alpha_0}{\dashrightarrow} a_1 \ldots \stackrel{\alpha_n}{\dashrightarrow} a_n.$$

By definition of $\rhd_m$, the existence of $p_{ga}$ and $p_a$ implies that of path $p_g$ in $\mathcal{G}$:

$$p_g : g_0 \stackrel{\alpha_0}{\dashrightarrow} g_1 \ldots \stackrel{\alpha_n}{\dashrightarrow} g_n.$$

Next, $p_g$, $p_{\bar{x}}$, and $p_a$ implies the existence of path $p_r$ in $(\mathcal{G} \curlywedge_m \bar{X}) \rhd_m (\mathcal{A} \rhd_m \bar{X})$:

$$p_r : ((g_0, \bar{x}_0), (a_0, \bar{x}_0)) \xrightarrow{\alpha_0} \ldots \xrightarrow{\alpha_n} ((g_n, \bar{x}_n), (a_n, \bar{x}_n)).$$

In addition, assume that there is a *must* transition

$$((g_i, a_i), \bar{x}_i) \xrightarrow{\alpha_i} ((g_{i+1}, a_{i+1}), \bar{x}_{i+1})$$

somewhere in $p_l$. By definition of $\curlywedge_m$, either

$$(g_i, a_i) \xrightarrow{\alpha_i} (g_{i+1}, a_{i+1}) \text{ holds or.}$$
$$\bar{x}_i \xrightarrow{\alpha_i} \bar{x}_{i+1} \text{ holds.}$$

implying that $(g_i, \bar{x}_i) \xrightarrow{\alpha_i} (g_{i+1}, \bar{x}_{i+1})$. Thus, there is the following *must* transition in $p_r$:

$$((g_i, \bar{x}_i), (a_i, \bar{x}_i)) \xrightarrow{\alpha_i} ((g_{i+1}, \bar{x}_{i+1}), (a_{i+1}, \bar{x}_{i+1})).$$

—Let $p_r$ be a path in $(\mathcal{G} \curlywedge_m \bar{X}) \rhd_m (\mathcal{A} \rhd_m \bar{X})$:

$$p_r : ((g_0, \bar{x}_0), (a_0, \bar{x}'_0)) \xrightarrow{\alpha_0} \ldots \xrightarrow{\alpha_n} ((g_n, \bar{x}_n), (a_n, \bar{x}'_n)).$$

By induction, we prove that

$$\forall 0 \leq i \leq n : ((g_i, \bar{x}_i), (a_i, \bar{x}'_i)) \text{ is not universal, and } \bar{x}_i \equiv \bar{x}'_i :$$

—Base case $i = 0$: Trivial.

—Step case: Assume that the induction holds up to the $i$-th state of $p_r$. By contraposition, assume that the $(i + 1)$-th state, i.e., $((g_{i+1}, \bar{x}_{i+1}), (a_{i+1}, \bar{x}'_{i+1}))$, is universal. Then by definition of $\rhd_m$, the following must hold:

$$(a_i, \bar{x}_i) \xrightarrow{\alpha_i} \not\rightarrow$$

which implies that $a_i \xrightarrow{\alpha_i} \not\rightarrow$ and $\bar{x}_i \xrightarrow{\alpha_i} \bar{x}_{i+1}$.

As $\bar{X} = X \curlywedge_m \mathcal{A}$, the latter then implies that $a_i \xrightarrow{\alpha_i} a_{i+1}$ by definition of $\curlywedge_m$, contradicting with the former. Thus, the $(i + 1)$-th state of $p_r$ is not universal, and this implies, by definition of $\rhd_m$, that

$$(a_i, \bar{x}_i) \xrightarrow{\alpha_i} (a_{i+1}, \bar{x}'_{i+1}),$$
$$(g_i, \bar{x}_i) \xrightarrow{\alpha_i} (g_{i+1}, \bar{x}_{i+1}),$$

which then implies that $(a_i, \bar{x}_i) \xrightarrow{\alpha_i} (a_{i+1}, \bar{x}_{i+1})$. Hence, $\bar{x}_{i+1} \equiv \bar{x}'_{i+1}$ by the deterministic assumption on modal automata.

The induction also infers the existence of $p_g$ in $\mathcal{G}$, $p_{\bar{x}}$ in $\bar{X}$, $p_a$ in $\mathcal{A}$:

$$p_g : g_0 \xrightarrow{\alpha_0} \ldots \xrightarrow{\alpha_n} g_n,$$
$$p_{\bar{x}} : \bar{x}_0 \xrightarrow{\alpha_0} \ldots \xrightarrow{\alpha_n} \bar{x}_n,$$
$$p_a : a_0 \xrightarrow{\alpha_0} \ldots \xrightarrow{\alpha_n} a_n,$$

which together implies that of $p_l$ in $(\mathcal{G} \rhd_m \mathcal{A}) \curlywedge_m \bar{X}$:

$$p_l : ((g_0, a_0), \bar{x_0}) \xrightarrow{\alpha_0} \ldots \xrightarrow{\alpha_n} ((g_n, a_n), \bar{x_n}),$$

In addition, if there is a *must* transition somewhere in $p_r$:

$$((g_i, \bar{x}_i), (a_i, \bar{x}_i)) \xrightarrow{\alpha_i} ((g_{i+1}, \bar{x}_{i+1}), (a_{i+1}, \bar{x}_{i+1})),$$

then by definition of $\rhd_m$, there must be

$$(g_i, \bar{x}_i) \xrightarrow{\alpha_i} (g_{i+1}, \bar{x}_{i+1}).$$

Thus, either $g_i \xrightarrow{\alpha_i} g_{i+1}$ or $\bar{x}_i \xrightarrow{\alpha_i} \bar{x}_{i+1}$ holds, implying the following transition in $p_l$:

$$((g_i, a_i), \bar{x}_i) \xrightarrow{\alpha_i} ((g_{i+1}, a_{i+1}), \bar{x}_{i+1}). \quad \square$$

With this theorem, our synthesis strategy for modal contracts needs only a minor modification. In other words, we compute and use $\bar{X}$ and $\bar{Y}_i$ instead of $X$ and $Y_i$ where,

$$\bar{X} \stackrel{\text{def}}{=} X \between_m \mathcal{A}_k,$$

$$\bar{Y}_i \stackrel{\text{def}}{=} Y_i \between_m \mathcal{A}_{k_i},$$

in applying Equations (2) and (3).

*Example* 8.2. We consider the simple message system `System` studied by Bauer et al. [2012]. The system consists of two components: component `Server` and component `User`. Their contracts are defined over the action set

$$\Sigma = \{\texttt{msg}, \texttt{secret\_msg}, \texttt{auth}, \texttt{send}\}$$

and are shown in Figures 3(a) through (h), where *may* transitions underlying *must* transitions are not drawn for simplicity.

The contract $\mathcal{C}_{Server} = (\mathcal{A}_{Server}, \mathcal{G}_{Server})$ models a simple protocol of sending (`send`) a message (`msg`) or secret message (`secret_msg`) from the `Server` to the `User`. In addition, the `Server` waits for an authentication code (`auth`) from the `User` before sending a secret message to it. The authentication code, however, is not required for sending a normal message. The contract $\mathcal{C}_{User} = (\mathcal{A}_{User}, \mathcal{G}_{User})$ then guarantees that the messages can always be received but does not ensure that authentication codes can be sent. The contracts described in this example are almost identical to those provided by Bauer et al. [2012] except that of component `Server`, where we make a minor modification to the assumption. In other words, authentication codes can only be received before messages are sent to the `User` while they are also allowed after such message sending in the original contract. Therefore, our modified assumptions is stronger than the original one. Decomposing the message system into these two components is then only possible when the system contract $\mathcal{C}_{System} = (\mathcal{A}_{System}, \mathcal{G}_{System})$ can also be decomposed into the component contracts $\mathcal{C}_{Server}$ and $\mathcal{C}_{User}$.

To verify the decomposition, we first normalize all guarantees as in Figure 3(c), (e), and (h). We next observe that the composition of the `Server` and `User` normalized guarantees, i.e., $\mathcal{G}^{\mathrm{n}}_{Server} \parallel_m \mathcal{G}^{\mathrm{n}}_{User}$, does not refine $\mathcal{G}^{\mathrm{n}}_{System}$ since the authentication code reception is allowed by the former and not allowed by the latter. In fact, the modification that we made to the contract assumption of component `Server` is the main reason for the failure of this decomposition. Thereby decomposing the message system into the two components would not be possible without performing some corrective synthesis.

We then apply our incremental synthesis strategy in Section 6 to synthesize the `Server` contract with respect to

$$\bar{X} = \left(\mathcal{G}^{\mathrm{n}}_{System} /_m \mathcal{G}^{\mathrm{n}}_{User}\right) \between_m \mathcal{A}_{Server},$$

which is shown in Figure 3(i). The newly synthesized `Server` contract provides the same guarantee under a more general assumption (Figure 3(j)). This new assumption corrects our wrong modification and allows authentication codes to always be received. We can now verify easily that the composition of the new `Server` contract and the `User` contract refines the overall `System` contract. As a result, the message system can be obtained by composing components `Server` and `User`.

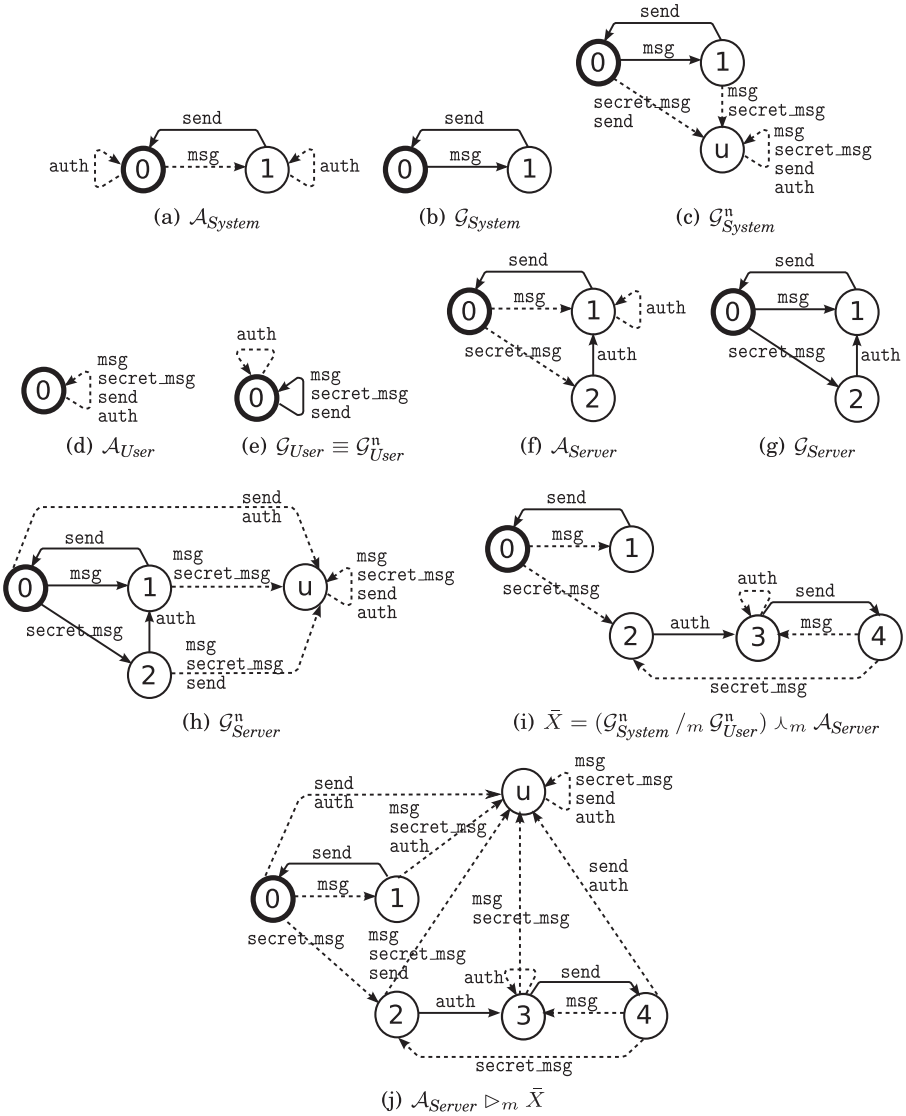



Fig. 3. Modal contracts for a simple message system.

## 9. CONCLUSIONS

In contract-based design, the top-down decomposition of a system into subcomponents is possible when the system general requirement $\mathcal{C}$ can be decomposed into requirements $\{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$ of the subcomponents. To support this top-down design procedure, we have presented a set of decomposing conditions for verifying the decomposition of a contract into a set of contracts. The conditions are defined on top of specifications operators such as normalization, composition, and refinement and work for any generic contract framework equipped with such operators. To provide for a complete design methodology, we have also proposed two synthesis strategies that can correct wrong contracts causing the condition failure. Our synthesis strategies can be applied to contract frameworks under the assumption that normalization and conjunction can be

interchanged. Although such assumption appears to be a limitation, it is a desirable property for flexible design methodologies, as it comes with a synthesis strategy for fixing wrong decompositions. The assumption can be made satisfied by continuously strengthening the core operand.

Our future work includes the implementation of our decomposition and synthesis strategy and the evaluation on verification performance. In particular, one essential step will be the integration of our strategy with several of the contract refinement checking methods that have been proposed in the literature.

## REFERENCES


Sebastian Bauer, Alexandre David, Rolf Hennicker, Kim Guldstrand Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. 2012. Moving from specifications to contracts in component-based design. In *Fundamental Approaches to Software Engineering*. Lecture Notes in Computer Science, Vol. 7212. Springer, 43–58.

Albert Benveniste, Benot Caillaud, Alberto Ferrari, Leonardo Mangeruca, Roberto Passerone, and Christos Sofronis. 2008. Multiple viewpoint contract-based specification and design. In *Formal Methods for Components and Objects*. Lecture Notes in Computer Science, Vol. 5382. Springer, 200–225.

Luca Benvenuti, Davide Bresolin, Pieter Collins, Alberto Ferrari, Luca Geretti, and Tiziano Villa. 2012. Ariadne: Dominance checking of nonlinear hybrid automata using reachability analysis. In *Reachability Problems*. Lecture Notes in Computer Science, Vol. 7550. Springer, 79–91.

L. Benvenuti, A Ferrari, L. Mangeruca, E. Mazzi, R. Passerone, and C. Sofronis. 2008. A contract-based formalism for the specification of heterogeneous systems. In *Proceedings of the Forum on Specification, Verification, and Design Languages (FDL'08)*. 142–147.

Matteo Bordin and Tullio Vardanega. 2007. Correctness by construction for high-integrity real-time systems: A metamodel-driven approach. In *Reliable Software Technologies—Ada Europe 2007*. Lecture Notes in Computer Science, Vol. 4498. Springer, 114–127.

Krishnendu Chatterjee and Thomas A. Henzinger. 2007. Assume-guarantee synthesis. In *Tools and Algorithms for the Construction and Analysis of Systems*. Lecture Notes in Computer Science, Vol. 4424. Springer, 261–275.

A. Cimatti and S. Tonetta. 2012. A property-based proof system for contract-based design. In *Proceedings of the 2012 38th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA'12)*. 21–28.

W. Damm, H. Hungar, B. Josko, T. Peikenkamp, and I. Stierand. 2011. Using contract-based component specifications for virtual integration testing and architecture design. In *Proceedings of the Design, Automation, and Test in Europe Conference and Exhibition (DATE'11)*. 1–6.

Abhijit Davare, Douglas Densmore, Liangpeng Guo, Roberto Passerone, Alberto L. Sangiovanni-Vincentelli, Alena Simalatsar, and Qi Zhu. 2013. METROII: A design environment for cyber-physical systems. *ACM Transactions on Embedded Computing Systems* 12, 1s, 49:1–49:31.

Alexandre David, Kim G. Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. 2009. Methodologies for specification of real-time systems using timed I/O automata. In Formal Methods for Components and Objects. Lecture Notes in Computer Science, Vol. 6286. Springer, 290–310. DOI:http://dx.doi.org/10.1007/978-3-642-17071-3

Alexandre David, Kim G. Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. 2010. Timed I/O automata: A complete specification theory for real-time systems. In *Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control (HSCC'10)*. 10.

Luca de Alfaro and Thomas A. Henzinger. 2001. Interface automata. In *Proceedings of the 8th European Software Engineering Conference Held Jointly with the 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-9)*. ACM, New York, NY, 109–120.

William P. de Roever. 1985. The quest for compositionality. In *Proceedings of the IFIP Working Conference on the Role of Abstract Models in Computer Science*.

Edsger W. Dijkstra. 1975. Guarded commands, non-determinacy and a calculus for the derivation of programs. In *Proceedings of the International Conference on Reliable Software*. ACM, New York, NY, 2–2.13.

Susanne Graf, Roberto Passerone, and Sophie Quinton. 2014. Contract-based reasoning for component systems with rich interactions. In *Embedded Systems Development: From Functional Models to Implementations*. Vol. 20. Springer, New York, NY, 139–154.

A. Iannopollo, P. Nuzzo, S. Tripakis, and A. Sangiovanni-Vincentelli. 2014. Library-based scalable refinement checking for contract-based design. In *Proceedings of the Design, Automation, and Test in Europe Conference and Exhibition (DATE'14)*. 1–6.

Patricia López Martínez and Tullio Vardanega. 2012. Handling synchronization requirements under separation of concerns in model-driven component-based development. In *Reliable Software Technologies—Ada Europe 2012*. Lecture Notes in Computer Science, Vol. 7308. Springer, 89–104.

Leslie Lamport. 1990. Win and sin: Predicate transformers for concurrency. *ACM Transactions on Programming Languages and Systems* 12, 3, 396–428.

Thi Thieu Hoa Le and Roberto Passerone. 2014. Refinement-based synthesis of correct contract model decompositions. In *Proceedings of the 12th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE'14)*. 134–143.

Thi Thieu Hoa Le, Roberto Passerone, Uli Fahrenberg, and Axel Legay. 2016. A tag contract framework for modeling heterogeneous systems. *Science of Computer Programming* 115–116, 225–246.

Shang-Wei Lin and Pao-Ann Hsiung. 2011. Counterexample-guided assume-guarantee synthesis through learning. *IEEE Transactions on Computers* 60, 5, 734–750.

Barbara Liskov and Jeannette M. Wing. 1994. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems* 16, 6, 1811–1841.

Diego Marmsoler, Alexander Malkis, and Jonas Eckhardt. 2015. A model of layered architectures. In *Proceedings of the 12th International Workshop on Formal Engineering Approaches to Software Components and Architectures (FESCA'15)*. 47–61.

Bertrand Meyer. 1992. Applying "design by contract." *Computer* 25, 10, 40–51.

Jean-Baptiste Raclet, Eric Badouel, Albert Benveniste, Benoît Caillaud, Axel Legay, and Roberto Passerone. 2011. A modal interface theory for component-based design. *Fundamenta Informaticae* 108, 1–2, 119–149.

Alberto Sangiovanni-Vincentelli, Werner Damm, and Roberto Passerone. 2012. Taming Dr. Frankenstein: Contract-based design for cyber-physical systems. *European Journal of Control* 18, 3, 217–238.

Received March 2015; revised November 2015; accepted December 2015