CrossMark

Tightening the contract refinements of a system architecture

Alessandro Cimatti[1] · Ramiro Demasi[1] ·
Stefano Tonetta[1]

Abstract Contract-based design is an emerging paradigm for correct-by-construction hierarchical systems: components are associated with assumptions and guarantees expressed as formal properties; the architecture is analyzed by verifying that each contract of composite components is correctly refined by the contracts of its subcomponents. The approach is very efficient, because the overall correctness proof is decomposed into proofs local to each component. However, the process for the contract specification and refinement is quite expensive because the requirements are formalised into formal properties, where part of the complexity is delegated to the designer, who has the burden of specifying the contracts. Typical problems include understanding which contracts are necessary, and how they can be simplified without breaking the correctness of the refinement and other refinements in case some subcontracts are shared. In this paper, we tackle these problems by proposing a technique to understand and simplify the contract refinements of a system architecture during the development process for the contract specification and refinement. The technique, called tightening, is based on parameter synthesis. The idea is to generate a set of parametric proof obligations, where each parameter evaluation corresponds to a variant of the original(s) contract refinement(s), and to search for tighter variants of the contracts that still ensure the correctness of the refinement(s). We cast this approach in the OCRA framework, where contracts are expressed with LTL formulas, and we evaluate its performance and effectiveness on a number of benchmarks.

Keywords Contract-based design · OCRA · Temporal logic · Parameter synthesis

✉ Ramiro Demasi
  demasi@fbk.eu

  Alessandro Cimatti
  cimatti@fbk.eu

  Stefano Tonetta
  tonettas@fbk.eu

[1] Fondazione Bruno Kessler, Trento, Italy

⚛ Springer

1 Introduction

Embedded-software systems are growing in number and technical complexity. They are becoming more and more sophisticated towards open, interconnected and networked systems. Due to the complexity of the context in which they operate, their defects can cause life-threatening situations and may have a huge economical impact [19]. In order to improve their efficiency and cost, their quality must be ensured with a rigorous analysis especially for those functions that have safety-critical requirements. Formal architectural models provide an important means to guarantee the correct refinement of system requirements along the design development and decomposition of the system.

Contract-based design, first conceived for software specification by Meyer [26] and nowadays also applied to embedded systems (cfr. e.g., [3–5,15,17,18,22,29]), is an emerging paradigm for correct-by-construction systems which structures components properties into contracts. A contract specifies the properties assumed to be satisfied by the component environment (assumptions), and the properties guaranteed by the component in response (guarantees). The architecture is analyzed by verifying that each contract of composite components is correctly refined by the contracts of its subcomponents.

In the contract framework proposed in [15,16], assumptions and guarantees are specified as temporal formulas. Checking the correctness of contracts refinement is supported by generating a set of necessary and sufficient conditions. These proof obligations are temporal formulas obtained from assumptions and guarantees, which are valid if and only if the refinement is correct. The approach is implemented in the OCRA tool [11,27] and is parametrized by a linear-time temporal logic, either propositional LTL [28], or LTL with SMT predicates [14], or HRELTL [13,14], a variant of LTL where formulas represent sets of hybrid traces, mixing discrete- and continuous-time steps, and therefore amenable to model properties of hybrid systems. The approach has been used in several contexts and domains. For example, in the FoReVer [2,20] and AutoFocus [8] framework, the OCRA approach was adopted to formalize requirements into contracts and to specify their refinements along the architecture decomposition in a top-down fashion during the development process. Moreover, a significant case study is presented in [6], where different variants of an industrial-size architectural model of a wheel braking system are analyzed, following the example outlined in the avionic AIR6110 standard.

The approach is very efficient, because the overall correctness proof is decomposed into proofs local to each component. However, the development process for the contract specification and refinement is quite expensive because the requirements are formalised into formal properties, where part of the complexity is delegated to the designer, who has the burden of specifying the contracts. Typical problems include understanding which contracts are necessary for satisfying each contract refinement starting from the system component to the leaf components in a top-down fashion, and how they can be simplified without breaking the correctness of the refinement.

In this article,[1] we aim to support the development process for the contracts specification and top-down refinement of a system architecture by proposing a technique to simplify the contract refinements during this process. Generally speaking, our approach takes as input a valid contract refinement and attempt to produce a simplified version: first, it finds which subconstracts are relevant for the refinement; second, it relaxes the contracts involved in the refinement by weakening the assumption of the parent component and the guarantees of the

---

[1] This is a revised and expanded version of a conference paper presented at *SEFM* 2016 [10]. A detailed description of the extension is given in Sect. 7.

subcomponents. This simplification is really useful at the initial stage of the formalization of the contracts both to validate the specification highlighting the relevant parts that are necessary to make the refinement correct and to relax the requirements for subcomponents increasing the design freedom for different implementation solutions. The general technique is called *tightening* and is based on parameter synthesis. The idea is to generate a set of parametric proof obligations for a single contract refinement, where each parameter evaluation corresponds to a variant of the original contract refinement, and to search for tighter variants of the contracts that still ensure the correctness of the refinement. In more details, the procedure first injects a set of parameters in the contract specification to create a search space of weakened assumptions of the parent contract and guarantees of the subcontracts. We have defined pattern-based functions that take as input a formula and they return a parametric formula and a set of injected parameter. Parameters are injected so that every parameter evaluation yields a respectively weaker or stronger formula. Moreover, we introduce parameters on the contracts in order to extend the parametric problem generated above to determine additionally which subcontracts can be removed from the contract refinement. Then, the intention is to apply this technique step by step on each contract refinement along with the successive decomposition of the system architecture. Note that, it could be the case that diverse contract refinements share some subcontracts which means that the tightened variants of a single contract refinement could possible break the correctness of the others. Consequently, we have developed a robust tightening technique called *parallel tightening* in order to deal with the problem of sharing of contracts by adding the proof obligations of the other contracts' refinements that have subcontracts in common. Thereby, we can obtain simplification of the contracts and keep the correctness of all involved contracts refinements.

We cast this approach in the OCRA framework and we evaluate its performance and effectiveness on a number of benchmarks, including the industrial-size architectures described in [6].

Outline The remainder of the paper is structured as follows. In Sect. 2 we introduce some notions used throughout the paper. In Sect. 3, we describe the motivation of using the tightening techniques for supporting the development process for the contracts specification and refinement of a system architecture. We present in Sect. 4 the formal definition and main algorithm for tightening a single contract refinement. In Sect. 5 we describe how tightening of a single contract refinement can be applied iteratively on the different levels of a system architecture and we solve the related issue of tightening contract refinements that share some contracts. We describe the experimental evaluation performed in Sect. 6. In Sect. 7 we discuss the related work. Finally, we discuss in Sect. 8 some conclusions and directions for further work.

2 Background

2.1 Transition systems

Given a finite set V of variables with a (potentially infinite) domain D, we denote with $\Sigma(V)$ the set of assignments to V, i.e. mapping from V to D. Let V' denote a copy of the variables V, which are used to represent the values of V after a transition.

A *transition system* (TS) S is a tuple $S = \langle V, I, T \rangle$, where V is a set of (state) variables, I is a formula over V representing the set of initial states, and T is a formula over $V \cup V'$

representing the set of transitions. A state $s \in \Sigma(V)$ of S is an assignment to the variables V.

With abuse of notation, we identify a formula ϕ over V with the set of states that satisfy ϕ. Thus, for example, the formula $\top$ ("true") represents the set $\Sigma(V)$. Similarly, we do not distinguish between a formula ϕ over $V \cup V'$ and the pairs of states (transitions) that satisfy ϕ.

A trace σ of S is an infinite sequence of states $\sigma = s_0, s_1, \ldots$ such that $s_0 \in I$ and for all $i \geq 0$, $\langle s_i, s_{i+1} \rangle \in T$. Given two transition systems $S_1 = \langle V_1, I_1, T_1 \rangle$ and $S_2 = \langle V_2, I_2, T_2 \rangle$, we define the synchronous product $S_1 \times S_2$ as $\langle V_1 \cup V_2, I_1 \wedge I_2, T_1 \wedge T_2 \rangle$. Since the product is commutative and associative, it can be generalized to a set of transitions systems.

2.2 LTL

Given a set of variables V, we assume to be given a set $Expr(V)$ of Boolean expressions over V as in [25]. In particular, in this paper we consider standard arithmetic predicates $(<, \leq, >, \geq, \ldots)$ and functions $(+, -, \ldots)$ over integer and real variables, although the proposed methods can be applied to more general settings.

We define the set of LTL formulas over the variables V with the following grammar rule:

$$\phi := p \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg \phi \mid X\phi \mid \phi U\phi \mid \phi R\phi$$

where p ranges in $Expr(V)$. We use the following standard abbreviations: $\top := p \vee \neg p$, $\bot := \neg\top$, $\phi \rightarrow \psi := (\neg\phi) \vee \psi$, $F\phi := \top U\phi$, $G\phi := \neg F\neg\phi$.

Traces over V are infinite sequences of assignments to V. Given a trace $\sigma = s_0, s_1, \ldots$, we denote with $\sigma[i]$ the $i+1$-th state s_i and with σ^i the suffix trace starting from $s[i]$.

Given a trace σ and an LTL formula ϕ over V, we define $\sigma \models \phi$ as follows:

– $\sigma \models p$ iff p evaluates to true given the assignment $\sigma[0]$
– $\sigma \models \phi \vee \psi$ iff $\sigma \models \phi$ or $\sigma \models \psi$
– $\sigma \models \phi \wedge \psi$ iff $\sigma \models \phi$ and $\sigma \models \psi$
– $\sigma \models \neg\phi$ iff $\sigma \not\models \phi$
– $\sigma \models X\phi$ iff $\sigma^1 \models \phi$
– $\sigma \models \phi U\psi$ iff there exists $i \geq 0$ s.t. $\sigma^i \models \psi$ and for all $j, 0 \leq j < i, \sigma^j \models \phi$
– $\sigma \models \phi R\psi$ iff for all $i \geq 0$ $\sigma^i \models \psi$ or there exists $j, 0 \leq j < i$, s.t. $\sigma^j \models \phi$

The satisfiability problem is the problem of checking if for a given LTL formula ϕ there exists a trace σ such that $\sigma \models \phi$.

Given a TS $S = \langle V, I, T \rangle$ and an LTL formula ϕ over V, $S \models \phi$ if for all trace σ of S, $\sigma \models \phi$. The satisfiability problem of an LTL formula over V can be reduced to model checking by considering the universal model as transition system: i.e., ϕ is satisfiable iff $\langle V, \top, \top \rangle \not\models \neg\phi$.

Note that we are considering in general infinite-state transition systems for which these problems are undecidable. Our methods are based on SMT-based algorithms as those implemented in nuXmv [9].

2.3 Parameter synthesis

The goal of parameter synthesis is to find the set of all parameter evaluations for which a given property is satisfied. Let S be a transition system and let U be a set of parameters, we define the parametric transition system $P = (V, U, I_U, T_U)$, where I_U is a formula over $V \cup U$ and T_U is a formula over $V \cup U \cup V'$. We define the parameters as frozen, i.e., we set their value

in the initial state and preserve it during the execution of the system. Given a valuation for the parameters ($\gamma \in \Sigma(U)$), and a formula ψ we write $\gamma(\psi) = \psi[u/\gamma(u)]_{u \in U}$, to indicate that each parameter has been substituted with its value. Given a parametric transition system P and a valuation for the parameters γ, we can compute the *induced* transition system, by replacing the parameters with their valuation: $P_\gamma = (V, \gamma(I_U), \gamma(T_U))$. Given an LTL property ϕ expressed over the state variables and parameters, the parameter region ρ is the set of assignments to the parameters such that the property is satisfied by every trace of the induced system, formally: $\rho = \{\gamma \mid P_\gamma \models \gamma(\phi)\}$.

In this paper, we consider Boolean parameters and, with abuse of notation, we identify a parameter evaluation γ with the set $\{p \mid p \in U, \gamma(p) = \top\}$. The parameter region is monotonic iff whenever $\gamma \subseteq \gamma'$, if $\gamma \in \rho$ then $\gamma' \in \rho$. The monotonicity of the parameter region is typically exploited by parameter synthesis algorithms that enumerate the parameter evaluations γ such that $P_\gamma \not\models \gamma(\phi)$. In fact, one can proceed with γ of increasing cardinality and as soon as $P_\gamma \models \gamma(\phi)$ all γ' with $\gamma \subseteq \gamma'$ can be included in ρ. Thus, in case of monotonicity, the parameter region can be represented by the set of minimal sets γ such that $P_\gamma \models \gamma(\phi)$ (taking implicitly the upward closure).

2.4 Contract refinement

In order to simplify the presentation, in this paper, we define a contract refinement independently from the component interfaces. In practice, in the tool support we consider, contracts are specified in terms of component input/output ports and the refinement has to take into account the connections among ports in component decomposition.

A contract C over the variables V is a pair $\langle \mathsf{A}, \mathsf{G} \rangle$ of LTL formulas over V representing respectively an *assumption* and a *guarantee*. We also denote A by $\mathcal{A}(C)$, G by $\mathcal{G}(C)$, and $\neg \mathsf{A} \vee \mathsf{G}$ by $nf(C)$. Let $C = \langle \mathsf{A}, \mathsf{G} \rangle$ be a contract over V. Let I and E be TS over V. We say that I is a correct implementation of C iff $I \models \mathsf{A} \rightarrow \mathsf{G}$. We say that E is a correct environment of C iff $E \models \mathsf{A}$. We denote by $\mathcal{I}(C)$ and $\mathcal{E}(C)$, respectively, the set of correct implementations and the set of correct environments of C. Given two contracts C and C' over V, we say that C refines C' (denoted by $C \preceq C'$) iff $\mathcal{I}(C') \subseteq \mathcal{I}(C)$ and $\mathcal{E}(C) \subseteq \mathcal{E}(C')$.

In a system architecture, each contract is associated to a component. If a component is decomposed into subcomponents, the associated contracts of the parent component are implemented by the composition of the subcomponents' implementations. Similarly, the environment of the contract of a subcomponent is given by the composition of the environment of the composite component and the implementations of the other subcomponents. In order to prove that such decomposition is correct, we generalize the refinement notion to a set of contracts.

Given a contract C and a set of contracts $Sub = \{C_1, \ldots, C_n\}$, we say that Sub is a refinement of C, written $Sub \preceq C$, iff the following conditions hold:

1. the correct implementations of the sub-contracts form a correct implementation of C: let V be the variables of C,

$$\{S_1 \times \cdots \times S_n \mid S_1 \in \mathcal{I}(C_1), \ldots, S_n \in \mathcal{I}(C_n)\} \subseteq \mathcal{I}(C)$$

2. for every $C_i \in Sub$, the correct implementation of the other sub-contracts and a correct environment of C form a correct environment of C_i: let V_i be the variables of C_i,

$$\begin{aligned}&\{E \times S_1 \times \cdots \times S_{j \neq i} \times \cdots \times S_n \mid \\ &E \in \mathcal{E}(C), \quad \text{for all} \quad j, 1 \leq j \leq n, j \neq i, S_j \in \mathcal{I}(C_j)\} \subseteq \mathcal{E}(C_i)\end{aligned}$$

In [15,16], we proved that the refinement is correct if and only if the following proof obligations ($PO(Sub, C)$) are valid temporal formulas:

$$nf(C_1) \wedge \cdots \wedge nf(C_n) \rightarrow nf(C)$$

$$\mathsf{A} \wedge \bigwedge_{1 \leq j \leq n, j \neq i} nf(C_j) \rightarrow \mathsf{A}_i \text{ (for every } i, 1 \leq i \leq n)$$

3 Motivation

3.1 Contract-based design

The contract-based design flow considered in this paper is depicted in Fig. 1, using the example of a control system with a redundant sensor. The example is taken from a case study developed in the FoReVer project [2,20]. This simple control example takes as input a value, speed, representing the physical speed of the system and returns a brake signal, brake. The system is decomposed into a RedundantSensor that reads the physical value and a ControlUnit that reads the speed from the sensor and produces corresponding control signals to the brakes. As the name suggests, the RedundantSensor is implemented with two redundant Sensors, so that the system works even if one of the sensor fails. In order to analyze
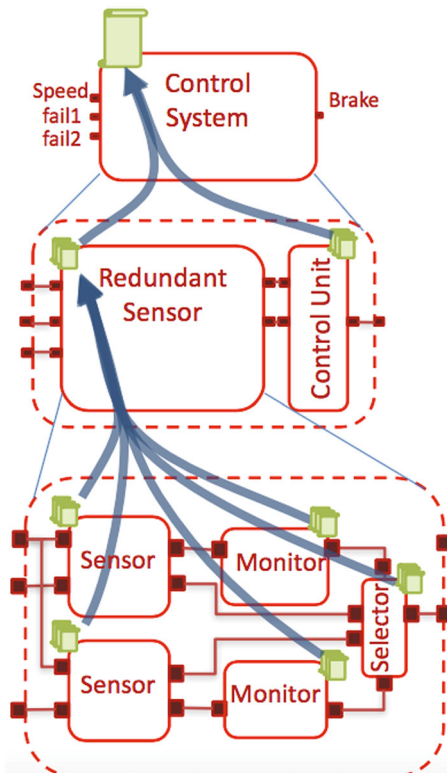

Fig. 1 Contract-based design flow

the behavior of the system in the presence of failures, we introduce two inputs, `fail1` and `fail2`, which represent the failures of the two sensors.

The design starts with the view of the `ControlSystem` as a whole black box with ports to interact with its environment. Then, it is decomposed into `RedundantSensor` and `ControlUnit` components. The `RedundantSensor` is in turn decomposed into two redundant `Sensor` subcomponents, two `Monitor` subcomponents, and a `Selector`. The decomposition also defines how the ports of the component being decomposed are mapped down into the decomposition. For example, the "left" ports of the `ControlSystem` are mapped onto the "left" ports of the `RedundantSensor`. At the lower level, `speed` and `fail1` are mapped to the input of first sensor, while the inputs of the second sensor are given by `speed` and `fail2`.

Each component in the hierarchy is associated with a set of *contracts*, depicted in green, specifying the acceptable behaviors for the component and its environment. So, for example, `ControlSystem` has a contract `speed_control` consisting of the guarantee $G(\text{speed} \geq th \rightarrow \text{brake})$, where th is a threshold parameter of the system, and the assumption $\neg\text{fail1} \land \neg\text{fail2} \land G(\neg\text{fail1} \lor \neg\text{fail2})$, i.e., initially none of the sensors is failing while in every future state it sufficient that one of the two is not failing.

Contracts are refined, following the decomposition of components. For example, the contracts of the `ControlSystem` are refined by some contracts of the `RedundantSensor` and the `ControlUnit` subcomponents. The framework guarantees that, under specific conditions (corresponding to correct contract refinement), if the contracts of the subcomponents hold, then the contract of the parent component also holds.

The contract-based top-down development process can be schematized with the following loop (as reported in the report [2] of the FoReVer project):

1. select a component (let us call it S) without subcomponents and decompose it specifying the children subcomponents;
2. for each type of the defined subcomponents, declare the component and its input and output ports;
3. detail the decomposition of the component S chosen at step 1 defining the connections among the subcomponents and the delegation connections between S and the subcomponents;
4. for each type of the subcomponents, specify the contracts that are necessary to fulfil the contracts of S;
5. for each contract C of S specify the refinement relationship, i.e., which contracts of the subcomponents refine C;
6. check the refinement with OCRA and, if not correct, adjust the contracts in order to remove all issues in the refinement;
7. if there are no more components to be refined, terminate successfully; otherwise, go to 1.

3.2 Need of tightening

The general development process for the contracts specification and refinement has been successfully applied in several frameworks [2,8] and on realistic case studies like in [6]. However, the process is quite expensive because the requirements are formalized into formal properties, where part of the complexity is delegated to the designer, who has the burden of specifying the contracts. It is clear that there is a missing of analysis methods for the improvement of contract understanding during the development process for the contracts specification and refinement. Automatic synthesis of the contract specification would be

ideal for this purpose, for example, if we have a partial specification of a contract refinement, we can automatically synthesize the assumption/guarantee of one of the subcontracts and thus understand in more details the contract specification. However, automatic synthesis is not in general feasible especially in the case of first-order formulas. Therefore, we aim at providing at least support when the contract refinement is manually specified.

Consider the `ControlSystem` example above. The `Selector` component has a contract with assumption "true" and guarantee:

$$\text{current\_use} = 1 \wedge$$
$$G((\text{current\_use} = 1 \wedge \text{switch\_current\_use}) \rightarrow X(\text{current\_use} = 2)) \wedge$$
$$G((\text{current\_use} = 2 \wedge \text{switch\_current\_use}) \rightarrow X(\text{current\_use} = 1))$$

where `current_use` is 1 or 2 depending on whether the output is linked to the output of sensor1 or sensor2 respectively, and `switch_current_use` triggers the change of `current_use` when the monitor of the current sensor detects a failure. Notice, however, that initially the current sensor is sensor1 and therefore there no need to assume (in the top-level assumption) that ¬`fail2` (i.e., that initially sensor2 does not fail).

As another example, consider a different system assumption on the occurrence of sensor faults:

$$\neg\texttt{fail1} \wedge (G(\neg\texttt{fail1}) \vee G(\neg\texttt{fail2}))$$

i.e., we assume that we cannot have both sensors eventually failing. Since initially the selector is using sensor1 and `switch_current_use` only triggers the change of `current_use` to sensor2 when the monitor of sensor1 detects a failure, given the stronger assumption, it will never switch back to sensor1. Thus, the part $G((\text{current\_use} = 2 \wedge \text{switch\_current\_use}) \rightarrow X(\text{current\_use} = 1))$ of the above guarantee is not necessary.

In general, it may happen to specify contracts on the subcomponents that are more demanding than necessary or that contain unwanted redundancies. It may happen also that, the designer specifies a very strong assumption on the system to make the refinement correct and later wants to relax the assumption while keeping the design correct. More generally, given a correct contract refinement, we would like to understand if the guarantees of subcomponents and assumption on the composite component can be weakened. Finally, the designer could include irrelevant subcontracts on a contract refinement that can be removed. Then, we would also like to provide the designer with information regarding which contracts are sufficient for (or prevent) correct refinement. We address all these issues by defining a problem called *top-down tightening* of a contract refinement.[2]

4 Tightening of a single contract refinement

In this section, we present the formal definition of local tightening for a single contract refinement and the algorithms for computing a top-down tightening of a given contract refinement.

---

[2] In [10], we also considered a bottom-up tightening but did not prove to be useful in practice for the top-down process described above, and thus we are omitting it here.

4.1 Formal definition

We now define formally the problem of tightening a contract refinement as follows. Given a contract C, and a set of contracts Sub such that $Sub \preceq C$, a *tightening* of this contract refinement is given by a contract C', a subset $\underline{Sub}$ of Sub, and a set of contracts $\underline{Sub'} = \{C'_S \mid C_S \in \underline{Sub}\}$ such that:

- $\underline{Sub'} \preceq C'$
- $C' \preceq C$ and, for every $C_S \in \underline{Sub}$, $C_S \preceq C'_S$.

A *top-down tightening* is a tightening as defined above such that $\mathcal{G}(C) = \mathcal{G}(C')$ and, for all $C_S \in \underline{Sub}$, $\mathcal{A}(C_S) = \mathcal{A}(C'_S)$. We can easily prove that, equivalently, a top-down tightening is given by a contract C', a subset $\underline{Sub}$ of Sub, and a set of contracts $\underline{Sub'} = \{C'_S \mid C_S \in \underline{Sub}\}$ such that:

- $\underline{Sub'} \preceq C'$
- $\mathcal{A}(C) \models \mathcal{A}(C')$ and, for every $C_S \in \underline{Sub}$, $\mathcal{G}(C_S) \models \mathcal{G}(C'_S)$.

4.2 The algorithm

We present now the main algorithm for top-down tightening of a contract refinement which takes as input a valid contract refinement $Sub \preceq C$ and produces as output a number of tightened versions of the given contract and its subcontracts. The procedure first injects a set P of parameters in the contract specification to create a search space of weakened assumptions of the parent contract C and guarantees of the subcontracts in Sub. Second, it creates the related proof obligations that are now parametrized by P. Additionally, we inject extra parameters on each contract in order to detect which contracts are sufficient for correct refinement. Then, we want to find for which configurations of the parameters the contract refinement holds. This is a multiple parameter synthesis problem, because we have to search for the assignment to P such that all proof obligations are valid. Thus, as third step, we propose two alternative ways to solve this problem (see Sect. 4.4). In the first step, we make sure that the injection of parameters creates a monotonic parameter region by construction, which can be exploited by the synthesis algorithm.

Algorithm 1 Tightening a contract refinement

Require: a contract C and a set of subcontracts Sub such that $Sub \preceq C$
Ensure: a set of $\langle \underline{Sub'}, C' \rangle$ such that $\underline{Sub} \subseteq Sub$ and $\underline{Sub'} \preceq C'$ and $C' \preceq C$ and, for every $C_S \in \underline{Sub}$,
$\quad C_S \preceq C'_S$.
1: {Calling top-down algorithm on Sub and C}
2: $\langle \langle Sub^P, C^P \rangle, P \rangle = TopDownTightening(Sub, C)$
3: {Extended Proof Obligations with extra parameters on the contracts}
4: $P_{sub} = \{p_1, \ldots, p_n\}$ with $n = |Sub|$ {one new parameter for each subcontract}
5: $PO = ExtendedPO(Sub^P, C^P, P_{sub})$
6: $P = P \cup P_{sub}$
7: {Solve multiple parameter synthesis problem}
8: $param\_region = SolveMultipleParamSyntProblem(PO, P)$
9: {Generate output}
10: $GenerateTightenedContractRef(PO, param\_region)$

These steps are formalized as follows, while the pseudo-code is shown in Algorithm 1. Suppose we want to obtain a top-down tightening of $Sub \preceq C$.

1. We transform C and Sub (line 2) into parametrized versions C^P and $Sub^P = \{C_S^P \mid C_S \in Sub\}$ with parameters P such that for every evaluation γ of P, if $\gamma(Sub^P) \preceq \gamma(C^P)$, then $\langle \gamma(C^P), \gamma(Sub^P) \rangle$ is a top-down tightening of $\langle C, Sub \rangle$.
2. We extend the construction of the proof obligations (line 5) of $\gamma(Sub^P) \preceq \gamma(C^P)$ by injecting further parameters $P_{Sub} = \{p_{C_S} \mid C_S \in Sub\}$, one for each subcontract $C_S \in Sub$, such that for every evaluation γ of $P \cup P_{Sub}$, $\gamma(\underline{Sub}_\gamma^P) \preceq \gamma(C^P)$ where $\underline{Sub}_\gamma^P = \{C_S^P \mid \gamma(p_{C_S}) = \top\}$.
3. We solve the multiple parameter synthesis problem (line 8) by either:

   (a) finding the region W_ϕ for each proof obligation and then intersecting such sets of parameter, i.e., $W = \{\gamma \in \Sigma(P) \text{ s.t. } \models \gamma(\phi(V, P)) \text{ for all } \phi \in PO(V, P)\} = \bigcap_{\phi \in PO} W_\phi$; or
   (b) encoding it into an equivalent proof obligation ϕ_{PO} so that $\{\gamma \in \Sigma(P) \text{ s.t. } \models \gamma(\phi) \text{ for every } \phi \in PO(V, P)\} = \{\gamma \in \Sigma(P) \text{ s.t. } \models \gamma(\phi_{PO})\}$.

Note that the output of the algorithm produces a set of tightened variants of the given contract refinement. The designer has to analyze these results and may decide to substitute the current specification with one of the tightened versions. Moreover, providing the different ways in which the specification can be simplified may improve the understanding of why the contract refinement is correct.

4.3 Generation of the parametric problem

4.3.1 Injecting parameters in the formulas

In this subsection, we describe how we introduce parameters in the formulas that define the assumption and guarantee of the contracts and generate a monotonic parameter synthesis problem. The high-level transformation is described in Algorithm 2 for top-down tightening of $Sub \preceq C$ where the assumption of the parent contract C and guarantees of the subcontracts in Sub are weakened.

Algorithm 2 Top-down tightening ($TopDownTightening(Sub, C)$)

Require: a contract C and a set of contracts $Sub = \{C_1, \ldots, C_n\}$
Ensure: $\langle \langle Sub^P, C^P \rangle, P \rangle$
1: $P = \emptyset$ {Set of parameters}
2: $Sub^P = \emptyset$
3: **for all** $C_S \in Sub$ **do**
4:     $\langle \phi, P' \rangle = Weaken(\mathcal{G}(C_S))$
5:     $P = P \cup P'$
6:     $Sub^P = Sub^P \cup \{\langle \mathcal{A}(C_S), \phi \rangle\}$
7: **end for**
8: $\langle \phi, P' \rangle = Weaken(\mathcal{A}(C))$
9: $P = P \cup P'$
10: $C^P = \langle \phi, \mathcal{G}(C) \rangle$
11: **return** $\langle \langle Sub^P, C^P \rangle, P \rangle$

The *Weaken* and *Strengthen* functions are described respectively in Algorithms 3 and 4. They take as input a formula and they return a parametric formula and a set of injected parameters. The definition assumes that every new parameter p is a fresh symbol. The number of parameters is linear in the size of the formula.

Parameters are injected so that every parameter evaluation yields a respectively weaker or stronger formula.

We remark that we do not aim to obtain the weakest or strongest version of a formula. In our approach, the definition of *Weaken* and *Strengthen* functions is pattern-based where new patterns can be investigated to complement or improve the current ones.

In the following, we use ϕ^W to denote the formula injected with parameters by *Weaken* (i.e., $Weaken(\phi) = \langle \phi^W, P \rangle$) and with ϕ^S the formula injected with parameters by *Strengthen* (i.e., $Strengthen(\phi) = \langle \phi^S, P \rangle$)

Theorem 1 *For any parameter evaluation γ, $\phi \rightarrow \gamma(\phi^W)$ and $\gamma(\phi^S) \rightarrow \phi$.*

Proof We prove the theorem by induction on the structure of the formula. The result of *Weaken* and *Strengthen* is outlined in Tables 1 and 2. It is routine to check line by line on the fourth column of the tables that, for every γ, $\phi \rightarrow \gamma(\phi^W)$ and $\gamma(\phi^S) \rightarrow \phi$, based on the inductive hypothesis that $\phi_1 \rightarrow \gamma(\phi_1^W)$, $\phi_2 \rightarrow \gamma(\phi_2^W)$, $\gamma(\phi_1^S) \rightarrow \phi_1$, and $\gamma(\phi_2^S) \rightarrow \phi_2$. Note, in particular, that the weakening of $\neg \phi_1$ is given by $\neg \phi_1^S$ and thus $\neg \phi_1 \rightarrow \neg \gamma(\phi_1^S)$ because by induction $\gamma(\phi_1^S) \rightarrow \phi_1$. Similarly for strengthening $\neg \phi_1$. $\square$

It follows immediately that Algorithm 2 yields a correct top-down tightening, as stated in the following corollary.

Corollary 1 *Let C be a contract and Sub a set of contracts. Let $\langle \langle Sub^P, C^P \rangle, P \rangle$ be the result of $TopDownTightening(Sub, C)$. Then, for any evaluation γ of the parameters P, if $\gamma(Sub^P) \preceq \gamma(C^P)$ then $\langle \gamma(Sub^P), \gamma(C^P) \rangle$ is a top-down tightening of $\langle Sub, C \rangle$.*

Moreover, the parameter injection is designed so that the semantics of the parametric formulas is monotonic with respect to the parameter evaluations.

Table 1 Simplification table for *Weaken(φ)*

Formula ϕ	$Weaken(\phi) = \langle \phi^W, P \rangle$	Evaluation γ	$\gamma(\phi^W)$
$a < b$	$p_1 \rightarrow (a < b) \wedge p_2 \rightarrow (a \leq b)$	$\{p_1, p_2\}$	$a < b$
		$\{p_1\}$	$a < b$
		$\{p_2\}$	$a \leq b$
		$\emptyset$	$\top$
$\phi_1 \wedge \phi_2$	$p_1 \rightarrow \phi_1^W \wedge p_2 \rightarrow \phi_2^W$	$\{p_1, p_2\}$	$\gamma(\phi_1^W) \wedge \gamma(\phi_2^W)$
		$\{p_1\}$	$\gamma(\phi_1^W)$
		$\{p_2\}$	$\gamma(\phi_2^W)$
		$\emptyset$	$\top$
$\phi_1 \vee \phi_2$	$\phi_1^W \vee \phi_2^W$	NA	$\gamma(\phi_1^W) \vee \gamma(\phi_2^W)$
$\phi_1 \mathcal{R} \phi_2$	$p_1 \rightarrow (\phi_1^W \wedge \phi_2^W) \wedge p_2 \rightarrow (\phi_1^W \mathcal{R} \phi_2^W)$	$\{p_1, p_2\}$	$\gamma(\phi_1^W) \wedge \gamma(\phi_2^W)$
		$\{p_2\}$	$\gamma(\phi_1^W) \mathcal{R} \gamma(\phi_2^W)$
		$\{p_1\}$	$\gamma(\phi_1^W) \wedge \gamma(\phi_2^W)$
		$\emptyset$	$\top$
$\phi_1 \mathcal{U} \phi_2$	$\phi_1^W \mathcal{U} \phi_2^W$	NA	$\gamma(\phi_1^W) \mathcal{U} \gamma(\phi_2^W)$
$\neg \phi_1$	$\neg \phi_1^S$	NA	$\neg \gamma(\phi_1^S)$

Table 2 Simplification table for $Strengthen(\phi)$

Formula ϕ	$Strengthen(\phi) = \langle \phi^S, P \rangle$	Evaluation γ	$\gamma(\phi^S)$
$a \leq b$	$\neg p_1 \rightarrow (a < b) \wedge \neg p_2 \rightarrow$ $(a = b) \wedge (p_1 \wedge p_2) \rightarrow$ $(a \leq b)$	$\{p_1, p_2\}$	$a \leq b$
		$\{p_2\}$	$a < b$
		$\{p_1\}$	$a = b$
		$\emptyset$	$\bot$
$\phi_1 \vee \phi_2$	$\neg p_1 \rightarrow \phi_1^S \wedge \neg p_2 \rightarrow$ $\phi_2^S \wedge (p_1 \wedge p_2) \rightarrow (\phi_1^S \vee \phi_2^S)$	$\{p_1, p_2\}$	$\gamma(\phi_1^S) \vee \gamma(\phi_2^S)$
		$\{p_2\}$	$\gamma(\phi_1^S)$
		$\{p_1\}$	$\gamma(\phi_2^S)$
		$\emptyset$	$\gamma(\phi_1^S) \wedge \gamma(\phi_2^S)$
$\phi_1 \wedge \phi_2$	$\phi_1^S \wedge \phi_2^S$	NA	$\gamma(\phi_1^S) \wedge \gamma(\phi_2^S)$
$\phi_1 \, \mathcal{U} \, \phi_2$	$\neg p \rightarrow \phi_2^S \wedge p \rightarrow \phi_1^S \, \mathcal{U} \, \phi_2^S$	$\{p\}$	$\gamma(\phi_1^S) \, \mathcal{U} \, \gamma(\phi_2^S)$
		$\emptyset$	$\gamma(\phi_2^S)$
$\phi_1 \, \mathcal{R} \, \phi_2$	$\phi_1^S \, \mathcal{R} \, \phi_2^S$	NA	$\gamma(\phi_1^S) \, \mathcal{R} \, \gamma(\phi_2^S)$
$\neg \phi_1$	$\neg \phi_1^W$	NA	$\neg \gamma(\phi_1^W)$

Theorem 2 *If $\gamma \subseteq \gamma'$, then $\gamma'(\phi^W) \rightarrow \gamma(\phi^W)$ and $\gamma(\phi^S) \rightarrow \gamma'(\phi^S)$.*

Proof Looking again at Tables 1 and 2, one can check the monotonicity case by case. In fact, for each type of formula, the lines report the result of *Weaken* and *Strengthen* sorted according to the strength of the parameter evaluation (third column). More precisely, if γ is below γ', then either they are incomparable or $\gamma \subset \gamma'$. Therefore it is routine to prove that, in the second case, $\gamma'(\phi^W) \rightarrow \gamma(\phi^W)$ and $\gamma(\phi^S) \rightarrow \gamma'(\phi^S)$ (fourth column). □

Note that parameters are introduced per contract, so they are shared by different occurrences of the assumption/guarantee in the proof obligations. It is immediate to show that, thanks to the structured way in which formulas are either strengthened or weakened, the resulting synthesis problem is monotonic, as stated in the following corollary.

Corollary 2 *Let C be a contract and Sub a set of contracts. Let $\langle \langle Sub^P, C^P \rangle, P \rangle$ be the result of $TopDownTightening(Sub, C)$. Then, for any evaluation γ, γ' of the parameters P such that $\gamma \subseteq \gamma'$, if $\gamma(Sub^P) \preceq \gamma(C^P)$ then $\gamma'(Sub^P) \preceq \gamma'(C^P)$.*

4.3.2 Injecting parameters on the contracts

In this subsection, we describe how we introduce parameters on the contracts in order to extend the parametric problem generated above to determine additionally which subcontracts can be removed from the contract refinement. This is done as part of the procedure $ExtendedPO$ called in Algorithm 1.

The main goal here is to generate a parametric problem to be able to find a possibly smaller subset of subcontracts that still refines the parent contract, that is, given a contract C, and a set of contracts Sub such that $Sub \preceq C$, we want to find a set $\underline{Sub} \subseteq Sub$ such that $\underline{Sub} \preceq C$.

Algorithm 3 *Weaken(ϕ)*

Require: a formula ϕ
Ensure: $\langle \phi^W, P \rangle$
1: **if** $\phi = a > b$ (similar for $\phi = a < b$) **then**
2:    $\phi^W = p_1 \rightarrow (a > b) \wedge p_2 \rightarrow (a \geq b)$
3:    **return** $\langle \phi^W, \{p_1, p_2\} \rangle$
4: **else if** $\phi = \phi_1 \wedge \phi_2$ **then**
5:    $\langle \phi_1^W, P_1 \rangle = Weaken(\phi_1), \langle \phi_2^W, P_2 \rangle = Weaken(\phi_2)$
6:    $\phi^W = p_1 \rightarrow \phi_1^W \wedge p_2 \rightarrow \phi_2^W$
7:    **return** $\langle \phi^W, P_1 \cup P_2 \cup \{p_1, p_2\} \rangle$
8: **else if** $\phi = \phi_1 \vee \phi_2$ **then**
9:    $\langle \phi_1^W, P_1 \rangle = Weaken(\phi_1), \langle \phi_2^W, P_2 \rangle = Weaken(\phi_2)$
10:    $\phi^W = \phi_1^W \vee \phi_2^W$
11:    **return** $\langle \phi^W, P_1 \cup P_2 \rangle$
12: **else if** $\phi = \phi_1 \mathcal{R} \phi_2$ **then**
13:    $\langle \phi_1^W, P_1 \rangle = Weaken(\phi_1), \langle \phi_2^W, P_2 \rangle = Weaken(\phi_2)$
14:    $\phi^W = p_1 \rightarrow (\phi_1^W \wedge \phi_2^W) \wedge p_2 \rightarrow (\phi_1^W \mathcal{R} \phi_2^W)$
15:    **return** $\langle \phi^W, P_1 \cup P_2 \cup \{p_1, p_2\} \rangle$
16: **else if** $\phi = \phi_1 \mathcal{U} \phi_2$ **then**
17:    $\langle \phi_1^W, P_1 \rangle = Weaken(\phi_1), \langle \phi_2^W, P_2 \rangle = Weaken(\phi_2)$
18:    $\phi^W = \phi_1^W \mathcal{U} \phi_2^W$
19:    **return** $\langle \phi^W, P_1 \cup P_2 \rangle$
20: **else if** $\phi = \neg \phi_1$ **then**
21:    $\langle \phi_1^S, P_1 \rangle = Strengthen(\phi_1)$
22:    **return** $\langle \neg \phi_1^S, P_1 \rangle$
23: **else**
24:    **return** $\langle p \rightarrow \phi, \{p\} \rangle$
25: **end if**

We define formally *ExtendedPO* as follows. Given a contract C, a set of contracts *Sub*, and a set of parameters P_{Sub} ($|P_{Sub}| = |Sub|$, i.e., one new parameter for each subcontract in *Sub*), the extended set of proof obligations (PO) with extra parameters is defined as follows:

– The extended proof obligation for satisfying the top level contract

$$\bigwedge_{i=1}^{n} (p_i \rightarrow (\mathsf{A}_i \rightarrow \mathsf{G}_i)) \rightarrow (\mathsf{A} \rightarrow \mathsf{G})$$

– For each $1 \leq j \leq n$, we have n extended proof obligations:

$$\left(\bigwedge_{i=1 \wedge i \neq j}^{n} (p_i \rightarrow (\mathsf{A}_i \rightarrow \mathsf{G}_i)) \wedge \mathsf{A} \right) \rightarrow (p_j \rightarrow \mathsf{A}_j)$$

The idea is that we inject activation variables, one for each subcontract, which are used for activating the corresponding contract C_i. We then construct the proof obligations and at the same time we perform an extension of these proof obligations so that these extensions encode the problem of finding $\underline{Sub'}$.

Note that if p_j is assigned to false, the whole proof obligation concerning the entailment of the assumption A_j is equivalent to trivially true. Such a parameter, however, makes that proof obligation non-monotone. So, in this case, the synthesis engine may be less efficient.

In any case, the result is correct, as proved in the following theorem, i.e. for every parameter evaluation on *ExtendedPO*, the resulting contract refinement is correct.

Algorithm 4 $Strengthen(\phi)$

Require: a formula ϕ
Ensure: $\langle \phi^S, P \rangle$
1: **if** $\phi = a \leq b$ (similar for $a \geq b$) **then**
2:     $\phi^S = \neg p_1 \rightarrow (a < b) \wedge \neg p_2 \rightarrow (a = b) \wedge (p_1 \wedge p_2) \rightarrow (a \leq b)$
3:     **return** $\langle \phi^S, \{p_1, p_2\} \rangle$
4: **else if** $\phi = \phi_1 \vee \phi_2$ **then**
5:     $\langle \phi_1^S, P_1 \rangle = Strengthen(\phi_1), \langle \phi_2^S, P_2 \rangle = Strengthen(\phi_2)$
6:     $\phi^S = \neg p_1 \rightarrow \phi_1^S \wedge \neg p_2 \rightarrow \phi_2^S \wedge (p_1 \wedge p_2) \rightarrow (\phi_1^S \vee \phi_2^S)$
7:     **return** $\langle \phi^S, P_1 \cup P_2 \cup \{p_1, p_2\} \rangle$
8: **else if** $\phi = \phi_1 \wedge \phi_2$ **then**
9:     $\langle \phi_1^S, P_1 \rangle = Strengthen(\phi_1), \langle \phi_2^S, P_2 \rangle = Strengthen(\phi_2)$
10:     $\phi^S = \phi_1^S \wedge \phi_2^S$
11:     **return** $\langle \phi^S, P_1 \cup P_2 \rangle$
12: **else if** $\phi = \phi_1 \, \mathcal{U} \, \phi_2$ **then**
13:     $\langle \phi_1^S, P_1 \rangle = Strengthen(\phi_1), \langle \phi_2^S, P_2 \rangle = Strengthen(\phi_2)$
14:     $\phi^S = \neg p \rightarrow \phi_2^S \wedge p \rightarrow \phi_1^S \, \mathcal{U} \, \phi_2^S$
15:     **return** $\langle \phi^S, P_1 \cup P_2 \cup \{p\} \rangle$
16: **else if** $\phi = \phi_1 \, \mathcal{R} \, \phi_2$ **then**
17:     $\langle \phi_1^S, P_1 \rangle = Strengthen(\phi_1), \langle \phi_2^S, P_2 \rangle = Strengthen(\phi_2)$
18:     $\phi^S = \phi_1^S \, \mathcal{R} \, \phi_2^S$
19:     **return** $\langle \phi^S, P_1 \cup P_2 \rangle$
20: **else if** $\phi = \neg \phi_1$ **then**
21:     $\langle \phi_1^W, P_1 \rangle = Weaken(\phi_1)$
22:     **return** $\langle \neg \phi_1^W, P_1 \rangle$
23: **else**
24:     **return** $\langle \neg p \rightarrow \phi, \{p\} \rangle$
25: **end if**

First, let us define the subset of contracts Sub corresponding to a given parameter evaluation γ as $Sub_\gamma = \{C_S \in Sub \mid \gamma(p_{C_S}) = \top\}$.

Theorem 3 *Let $\langle \langle Sub^P, C^P \rangle, P \rangle$ the result of $TopDownTightening(Sub, C)$ for a given contract C and a set of contracts Sub. Let P_{Sub} be a set of parameters containing one new parameter for each subcontract in Sub^P. Then, for all parameter evaluation γ, $\gamma(ExtendedPO(\langle Sub^P, C^P \rangle, P_{Sub}))$ and $PO(\gamma(Sub_\gamma^P), \gamma(C^P))$ are equivalent.*

Proof We prove that if $\phi \in ExtendedPO(\langle Sub^P, C^P \rangle, P_{Sub})$, then either $\gamma(\phi)$ is valid or there exists an equivalent proof obligation in $PO(\gamma(Sub_\gamma^P), \gamma(C^P))$.

Suppose ϕ is the extended proof obligation for the top level contract, that is, $\bigwedge_{1 \leq i \leq n}(p_i \rightarrow (A_i \rightarrow G_i)) \rightarrow (A \rightarrow G)$. Then, $\gamma(\phi)$ is equivalent to $\bigwedge_{1 \leq i \leq n, \gamma(p_i) = \top} \gamma((A_i \rightarrow G_i) \rightarrow (A \rightarrow G))$, which belongs to $PO(\gamma(Sub_\gamma^P), \gamma(C^P))$.

Suppose ϕ is the extended proof obligation of the C_S assumption, where $C_S \in Sub$. Thus, $\phi = \bigwedge_{1 \leq i \leq n}(p_i \rightarrow (A_i \rightarrow G_i)) \rightarrow (A \rightarrow p_j \rightarrow A_j)$. If $\gamma(p_j) = \bot$, then $\gamma(\phi)$ is equivalent to true. If instead $\gamma(p_j) = \top$, then $\gamma(\phi)$ is equivalent to $\bigwedge_{1 \leq i \leq n, \gamma(p_i) = \top} \gamma((A_i \rightarrow G_i) \rightarrow (A \rightarrow A_j))$, which belongs to $PO(\gamma(Sub_\gamma^P), \gamma(C^P))$.

Similarly, we can prove that for every ψ in $PO(\gamma(Sub_\gamma^P), \gamma(C^P))$, there exists $\phi \in ExtendedPO(\langle Sub^P, C^P \rangle, P_{Sub})$ such that ψ is equivalent to $\gamma(\phi)$. $\qquad\square$

4.4 Multiple validity parameter synthesis problem

The approach to solve the tightening problem proposed in Sect. 4.2 introduces the problem of finding the parameter evaluations γ such that each formula $\phi(P, V) \in PO$ instantiated with γ is valid. In other words, we want to find for which configurations of the parameters the contract refinement holds. Each validity problem can be reduced to a model checking problem but the parameter evaluation is shared by the different verification problems. This is distinct from the standard parameter synthesis problem where only one verification problem is considered. In fact, we have to search for the assignment to P such that all proof obligations are valid. We called this problem a multiple validity parameter synthesis problem (to be not confused with multiple objective parameter synthesis problem). In the following subsections we propose two approaches to deal with this problem, which are alternative solutions for the function $SolveMultipleParamSyntProblem(PO, P)$ called in Algorithm 1.

4.4.1 Compositional approach

The simplest solution is to find the parameter region for each proof obligation and calculate the intersection of the results. The correctness of the approach is formalized by the following theorem.

Theorem 4 *If $W = \{\gamma \in \Sigma(P) \text{ such that } \models \gamma(\phi(V, P)) \text{ for all } \phi \in PO(V, P)\}$ and $W_\phi = \{\gamma \in \Sigma(P) \text{ such that } \models \gamma(\phi(V, P))\}$, then $W = \bigcap_{\phi \in PO} W_\phi$.*

Proof It trivially derives from the definition of sets intersection. □

4.4.2 Encoding all proof obligations into a single one

As an alternative approach, we propose to reduce the multiple validity problems to one validity problem by renaming the variables in V and taking the conjunction of the proof obligations. Namely, if $PO = \{\phi_1, \ldots, \phi_n\}$ we create the formula $\phi_{PO}(P, V_1, \ldots, V_n) = \bigwedge_{1 \leq j \leq n} \phi_j[V_j/V]$, where V_j contains one copy v_j for each variable $v \in V$ and $\phi_j[V_j/V]$ is the formulas obtained by substituting every variable $v \in V$ with v_j (while the parameters P remain unchanged).

Theorem 5 *For all parameter evaluation γ, $\gamma(\phi_{PO})$ is valid iff, for all formulas $\phi \in PO$, $\gamma(\phi)$ is valid.*

Proof $\Rightarrow$) Suppose for some $\phi_j \in PO$, $\gamma(\phi_j)$ is not valid. Let σ be a trace over V satisfying $\neg\gamma(\phi_j)$. Let us define the trace σ_j such that, for every $i \geq 0$, for all $v \in V$, $\sigma_j[i](v_j) = \sigma[j](v)$. Let us extend σ_j to a trace σ'_j over $V_1 \cup \cdots \cup V_n$ assigning variables not in V_j in an arbitrary way. Then σ'_j satisfies $\neg\gamma(\phi_{PO})$.

$\Leftarrow$) Suppose for ϕ_{PO} is not valid. Let σ be a trace over $V_1 \cup \cdots \cup V_n$ satisfying $\neg\gamma(\phi_{PO})$. Then, there exists j, $1 \leq j \leq n$, such that $\sigma \models \neg\gamma(\phi_j[V_j/V])$. Let us define the trace σ_j such that, for every $i \geq 0$, for all $v \in V$, $\sigma_j[i](v) = \sigma[j](v_j)$. Then σ'_j satisfies $\neg\gamma(\phi_j)$. □

4.4.3 Minimal configuration of the parameters

For each parameter evaluation, we should present to the user the corresponding tightened contract refinement. Since the parameter region may be really large, it is not feasible to

present all possible configurations and we need to select some representative cases. In fact, we present only minimal configurations, assuming monotonicity. This is an under-approximation and in case the parameter region is not upward closed (non-monotonicity), there may be supersets (less tightened versions) that are not in the paremeter region. However, all presented configurations are correct tightening (and thus correct contract refinements).

In practice, we use an off-the-shelf algorithm, reported in [7], to compute the parameter region which generates the minimal configurations of the parameters under the monotonicity assumption. In the compositional approach, we compute the parameter region for each proof obligation by calling each time the off-the-shelf algorithm. After that, we intersect such sets of minimal configurations of the parameters. Additionally, we compute the prime implicants of the intersection result to obtain the minimal configurations of the parameters such that all proof obligations are valid. In the second solution, the problem is reduced to one validity problem which means that only one time the off-the-shelf algorithm is called and it is obtained the minimal parameters of the region.

We would like to remark also that we have defined two alternatives solutions considering different encodings of the problem and also taking into account the number of times to be call the parameter synthesis algorithm reported in [7] which is used as back-end for computing the parameter region. Note that we obtain the same parameter region as result for each contract refinement with both approaches. It is clear that the encoding of all proof obligations into a single one is syntactically more complex than the composition approach. Moreover, the off-the-shelf algorithm is called only one time for computing the parameter region, contrary to the compositional approach which is called each time for each proof obligation. Furthermore, we have to integrate each result as was explained above such that all proof obligations are valid. In Sect. 6, we compare the performance of both approaches.

5 Tightening the whole system architecture

5.1 Applying tightening within a system architecture

Let us now explain how we can apply the single tightening technique on the whole system architecture during the development process for the contract specification and refinement. Let us take the architecture example from Fig. 1. We start from the system component `ControlSystem`, where only the contract `speed_control` is defined. This contract is refined by (1) the contract `sense` of the redundant sensor, which guarantees that the value passed to the control unit approximates the physical value with a bounded error; (2) the contract `sensed_speed_is_present` of the redundant sensor, which guarantees that the value provided by the sensor is always available; (3) the contract `speed_control`, which similarly to the system contract guarantees that if the speed is above the threshold, then a brake command is issued. After checking the contract refinement is correct, we are able to apply single top-down tightening on the parent contract `speed_control` of the system component in order to simplify the contract refinement. Suppose in this case we do not obtain any simplification and we proceed in a top-down fashion in the decomposition tree.

We consider now the `RedundantSensor` where two contracts are defined, `sense` and `sensed_speed_is_present` (see Fig. 2), which are refined by the contracts of the subcomponents of `RedundantSensor`. After the contracts specification and refinement is completed at this level and the check of the correctness of the contract refinements for `sense`
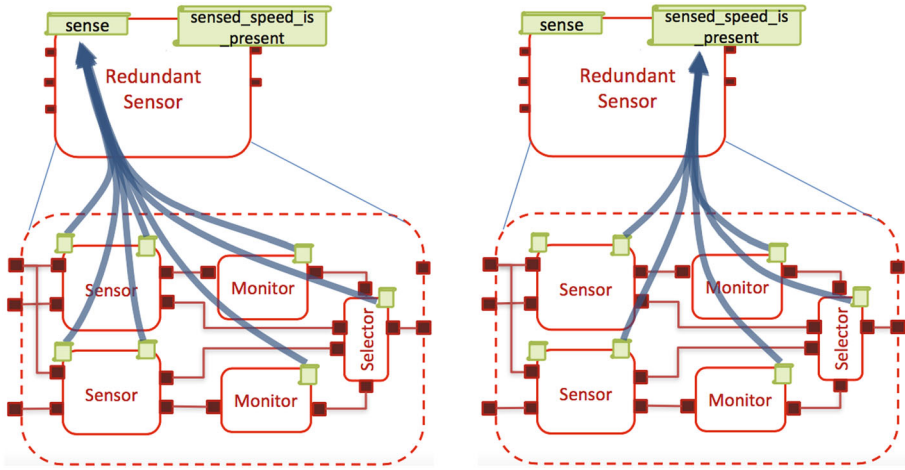
Fig. 2 Sharing of contracts between `sense` and `sensed_speed_is_present` in component `BSCU`

and `sensed_speed_is_present` of component `BSCU` is successful, we proceed with tightening these contracts.

We observe that these contracts share many subcontracts, and in particular they share the subcontract of the `Selector` component, which as explained in Sect. 3.2 can be simplified. In this situation, independently from the order of tightening the contracts `sense` and `sensed_speed_is_present`, the result of each one can in principle break the correctness of the other and vice versa. This is due to the fact that the tightening technique proposed in the previous section only ensures the correctness of an individual contract refinement. We call this *the problem of sharing contracts*.

We propose a solution where we ensure that, when we are tightening a contract refinement, the correctness of the parallel refinements (those that share some subcontracts with the current one) is preserved. We called this solution *parallel tightening* which is presented in Sect. 5.2.

Although parallel tightening produces a result that is correct also in the case of shared subcontracts, ensuring that all refinements are correct may mean that no simplification is possible. As an alternative approach, we can simply duplicate the shared contracts so that the refinements are independent and can be tightened as proposed in Sect. 4. This duplication approach is described in more detail in Sect. 5.3.

We would like to remark here that we are not proposing an automatic application of tightening, simplifying all contract refinements of a system architecture iteratively until reaching a fixpoint. We think that from the designer side, it will be really difficult to understand such tightened results at once. Thus, we propose to analyze one tightening result at a time, as explained at the beginning of the section.

5.2 Parallel tightening of a contract refinement

We define formally the problem of *parallel tightening* of a contract refinement as follows. Given a contract C, a set of contracts Sub such that $Sub \preceq C$, and a set of contract refinements $Sub_k \preceq C R_k$ such that for every k, $1 \leq k \leq n$, $Sub_k \cap Sub \neq \emptyset$, a *parallel tightening* of this contract refinement is given by a contract C', a subset $\underline{Sub}$ of Sub, and a set of contracts $\underline{Sub'} = \{C'_S \mid C_S \in \underline{Sub}\}$ such that:

- $\underline{Sub'} \preceq C'$
- $C' \preceq C$ and, for every $C_S \in \underline{Sub}$, $C_S \preceq C'_S$.
- $Sub'_k \preceq CR_k$, for every k, $1 \leq k \leq n$

where $Sub'_k = \{C_S \mid C_S \in Sub_k \setminus \underline{Sub}\} \cup \{C'_S \mid C_S \in Sub_k \cap \underline{Sub}\}$ (i.e., if the subcontract is not shared, it remains the same; otherwise, it is simplified).

In order to deal with a parallel contract refinement, we have to generate not only the proof obligations of the given contract refinement $Sub \preceq C$ but also the proof obligations of the related contract refinements whose have subcontracts in common. Suppose we want to obtain a parallel top-down tightening of $Sub \preceq C$, we add steps 3 and 4 to the original procedure as follows:

1. We transform C and Sub into parametrized versions C^P and $Sub^P = \{C_S^P \mid C_S \in Sub\}$ with parameters P such that for every evaluation γ of P, if $\gamma(Sub^P) \preceq \gamma(C^P)$, then $\langle \gamma(C^P), \gamma(Sub^P) \rangle$ is a top-down tightening of $\langle C, Sub \rangle$.

2. We extend the construction of the proof obligations of $\gamma(Sub^P) \preceq \gamma(C^P)$ by injecting further parameters $P_{Sub} = \{p_{C_S} \mid C_S \in Sub\}$, one for each subcontract $C_S \in Sub$, such that for every evaluation γ of $P \cup P_{Sub}$, $\gamma(\underline{Sub}^P) \preceq \gamma(C^P)$ where $\underline{Sub}_\gamma^P = \{C_S^P \mid \gamma(p_{C_S}) = \top\}$.

3. We search for those contract refinements $Sub_k \preceq CR_k$ such that $Sub_k \cap Sub \neq \emptyset$.

4. We generate the proof obligations for each of these related contract refinements $Sub_k \preceq CR_k$ considering the parametric version of the subcontracts in common, and we add them to the set PO.

5. We solve the multiple parameter synthesis problem by either the compositional approach or encoding all POs into a single one.

These new steps are sketched in Algorithm 5.

Algorithm 5 Parallel tightening of a contract refinement

Require: a contract C, a set of subcontracts Sub such that $Sub \preceq C$, and a set of contract refinements $Sub_k \preceq CR_k$ such that for every k, $1 \leq k \leq n$, $Sub_k \cap Sub \neq \emptyset$
Ensure: a set of $\langle Sub', C' \rangle$ such that $\underline{Sub} \subseteq Sub$ and $\underline{Sub'} \preceq C'$ and $C' \preceq C$ and, for every $C_S \in \underline{Sub}$, $C_S \preceq C'_S$, and $Sub'_k \preceq CR_k$, for every k, $1 \leq k \leq n$ where $Sub'_k = \{C_S \mid C_S \in Sub_k \setminus \underline{Sub}\} \cup \{C'_S \mid C_S \in Sub_k \cap \underline{Sub}\}$
1: {Calling top-down algorithm on Sub and C}
2: $\langle \langle Sub^P, C^P \rangle, P \rangle = TopDownTightening(Sub, C)$
3: $P_{sub} = \{p_1, \ldots, p_n\}$ with $n = |Sub|$ {one new parameter for each subcontract}
4: $PO = ExtendedPO(Sub^P, C^P, P_{sub})$
5: $P = P \cup P_{sub}$
6: **for all** $Sub_k \preceq CR_k$ **do**
7:    {Replace original shared subcontracts by the parametric version of them}
8:    $Sub_k^P = \{C_S \mid C_S \in Sub_k \setminus Sub\} \cup \{C_S^P \mid C_S \in Sub_k \cap Sub\}$
9:    $PO = PO \cup ConstructPO(Sub_k^P, CR_k)$
10: **end for**
11: {Solve multiple parameter problem by encoding all POs into a single PO or using the compositional approach}
12: $param\_region = SolveMultipleParamSyntProblem(PO, P)$
13: {Generate output}
14: $GenerateTightenedContractRef(PO, param\_region)$

5.3 Removing the sharing of contracts by duplication

In order to deal with the problem stated above regarding some situations when *parallel tightening* does not produce good results, we propose a solution by removing the sharing of contracts by duplication of contracts and running single tightening on the input contract refinement. Let us better explain this method using an example, by taking up again the contract refinements depicted in Fig. 2, where there are two contract refinements.

Suppose now that we want to tighten the contract refinement (1) using *parallel tightening*. As a result we do not get any tightening, obtaining the same contract refinement. Then, we first duplicate the contracts that are shared by the two refinements. We update the contract refinements so that one uses the original subcontracts, while the other uses the duplicated ones. For example, we duplicate the contract `switch` of `Selector` and update the refinements as shown in Fig. 3. Finally, we are able to run the single tightening procedure on each refinement in isolation.

This alternative method can be also applied during the development process in a top-down fashion ensuring the preservation of the correctness of contract refinements for dealing with the problem of shared contracts. The order in which the contract refinements are selected for removing the sharing of contracts by duplication and then run single top-down tightening is arbitrary.

6 Experimental evaluation

6.1 Details of the implementation

We have implemented the algorithms described in the previous sections on top of OCRA [11], a tool for architectural design based on contract-based design. In more details, we implemented a new command in OCRA called `ocra_tighten_contract_refinement` that takes as input an OCRA specification, a contract's name, a component's name, and produces as output a number of OCRA specifications containing the tightened versions of the
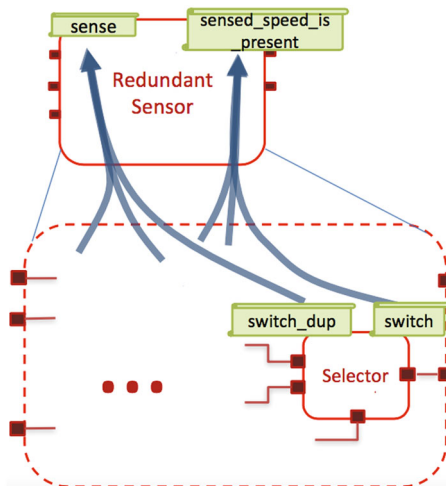

Fig. 3 Duplication of contract `switch` in component `Selector`

given contract and its subcontracts. The command has an option to allow the user to select the approach to the multiple synthesis problem (see Sect. 4.4), either the compositional approach (Sect. 4.4.1) or the encoding approach (Sect. 4.4.2). Other options in the command deal with the problem of sharing contracts: the user can choose to perform a *parallel tightening* or *duplicate* the contracts and run a single top-down tightening. Regarding the parameter synthesis algorithm, we have used as back-end an implementation reported in [7]. Since the synthesis is quite expensive for large number of parameters, we arbitrarily limit the injection to 350 parameters so that only the higher level of the formula structure is considered during weakening/strengthening (i.e., during the recursion of Algorithms 3 and 4, when reaching 350 parameters, the recursion is stopped and subfomulas are considered without modifications). This allows to get a tightening also in cases in which the definitions would produce many more parameters making the synthesis blow up.

We also implemented self checks to validate the results: first, we automatically check that each tightened contract refinement is correct; second, we automatically check for each tightened specification that the original formula entails the weakened formula.

6.2 Description of benchmarks

We have taken several benchmarks from case studies developed using OCRA in past projects. Some examples are: different versions of the Redundant Sensor described in Sect. 3.1, different variants of a Wheel Brake System [6,15], and an Airbag system [1]. Particularly, an interesting case study is taken from [6], where the authors presented a complete formal analysis of the different architectures described in the AIR6110 [30], a document describing the informal design of a Wheel Brake System (WBS), covering all the phases of the process, and modeled the case study by means of a combination of formal methods including contract-based design using OCRA, model checking and safety analysis.

Overall, the benchmarks suite comprises 909 contract refinements, with on average 8.4 subcontracts. 593 out of these 909 share some contracts with other contract refinements.

6.3 Experimental results

6.3.1 Tightening a single contract refinement

We carried out an experimental evaluation on the 909 contract refinements comparing the contract specification before and after simplification with respect to the length of the formulas on the original contracts.[3] The results of applying single top-down tightening is shown in Fig. 4, where blue crosses represent those cases which all subcontracts are relevant after tightening and red circles indicates those cases in which the simplification involves the removal of some irrelevant subcontracts. From the results, we can clearly see a significant simplification. 268 out of the 909 contract refinements did not get any simplification, which seems a reasonable result.

In Fig. 5, it is shown how our approach scales with respect to the number of parameters used for tightening a contract refinement and the time for computing the parameter region for three extended versions of the WBS example. In Fig. 6 we compare the performance for computing the parameter region using the encoding and compositional approaches for single top-down tightening. We can see that the compositional approach is much more efficient than the encoding one. This is probably due to the fact that many assumptions are true (on average

---

[3] We consider the standard definition of the length of a formula (number of symbols), apart from the length of $\top$ and $\bot$, which is set to 0.
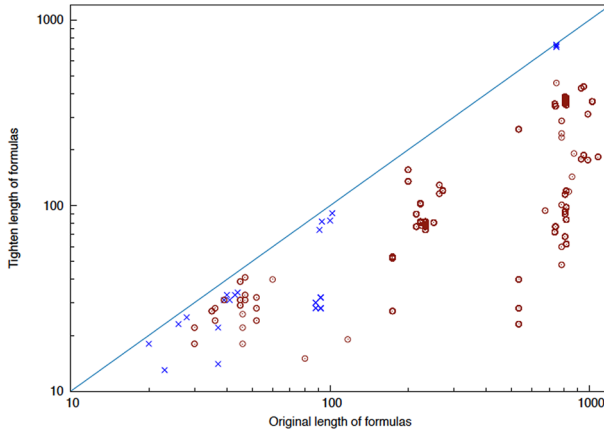
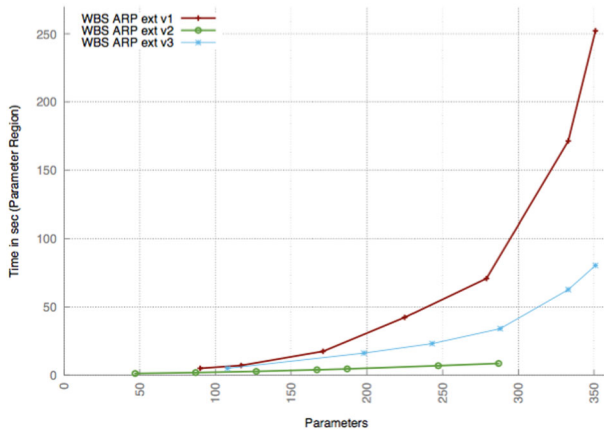**Fig. 4** Analysis of length of formulas for single top-down tightening

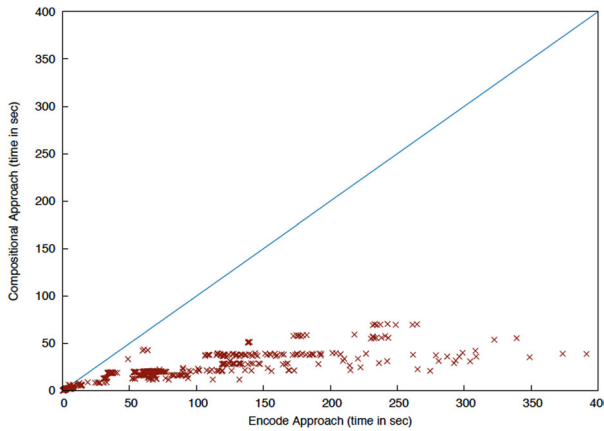**Fig. 5** Parameter scalability for single top-down tightening

Fig. 6 Comparison between compositional and encode approach for single top-down tightening

Table 3 Results of irrelevant contracts for single top-down tightening

Case study	# Contract refinement	Avg. # original subcontracts	Avg. # irrelevant contracts
wbs arch1	106	9.14	4.26
wbs arch2 v1	68	3.47	0.82
wbs arch2 v2	109	11.04	4.93
wbs arch2bis v1	68	3.47	0.77
wbs arch2bis v2	109	11.04	5.14
wbs arch2bis v3	109	11.04	5.21
wbs arch3	122	11.04	4.97
wbs arch4	122	11.04	5.17
Modified CRs wbs arch2 v1	12	28	22
Modified CRs wbs arch3	12	28	22
Modified CRs wbs arch4	12	28	22
Lift system	1	4	0
Simple wbs	1	3	0
wbs arp v1	3	3.66	1.33
wbs arp v2	4	3.25	1.25
wbs arp v3	5	3.2	1.45
wbs arp v4	5	3.2	1.45
wbs arp v4.1	4	3	1.45
wbs arp v2 ext	7	2	1
wbs arp v3 ext	7	1	0
wbs arp v4 ext	7	2	1
Airbag	7	2.42	0
Monitors v1	1	6	2
Monitors v2	1	6	0
gb2	1	3	0
Redundant sensors v1	3	5.66	0
Redundant sensors v2	3	5.66	0

5.04) so that the corresponding proof obligations are trivial and do not restrict the resulting parameter region. In the compositional approach, this means that the computation time for these proof obligation is negligible. In the encoding approach, although the resulting region is the same, the problem is syntactically more complex.

We also report on Table 3 the results of the average of the irrelevant subcontracts removed for each case study on single tightening (fourth column). We can observe that such number is higher for those benchmarks with a complex architecture. The reason is because the designer has to select manually the subcontracts which became a more difficult task with big architectures. Consequently, he/she may include subcontracts that are not relevant.

6.3.2 Taking into account shared contracts

For analyzing the problem of sharing of contracts we have considered the 593 contract refinements that contain shared contracts. We have run parallel tightening on these where
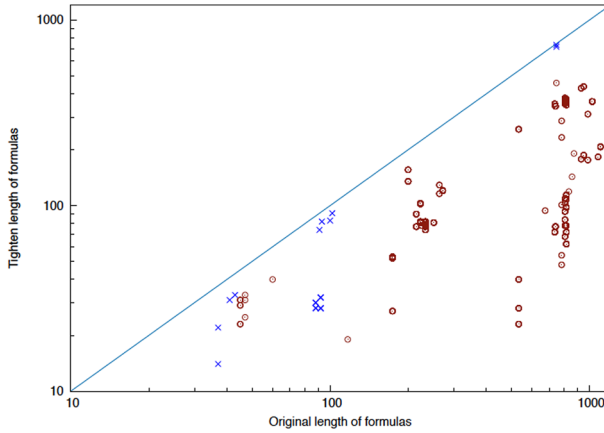
Fig. 7 Analysis of length of formulas applying duplication and single top-down tightening
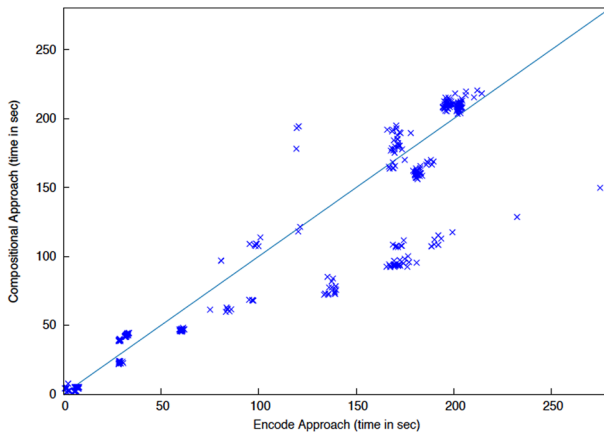

Fig. 8 Comparison between compositional and encode approach for *parallel* top-down tightening

we did not get any simplification in most of them. More precisely, we can simplify only 5 contract refinements. This is a result that can be expected due to we are including the proof obligations of other contract refinements that shared some contracts and it is not possible to get any simplification by keeping all contract refinements correct. Therefore, we have applied duplication of contracts and then run single tightening on those contract refinements that we did not get any simplification with parallel tightening and we have obtained a lot of improvement deployed in Fig. 7, where blue crosses represents those cases which all subcontracts are relevant after tightening and red circles indicates those cases in which the simplification involves the removal of some irrelevant subcontracts. Moreover, we remark that for only 31 contract refinements we did not get any simplification.

Finally, we show in Fig. 8 a comparison of the synthesis approaches for parallel top-down tightening. We can observe that in some cases the compositional approach is faster than the encoding one and vice versa. The performance of the encoding approach is therefore better than the single tightening case. This is probably due to the fact that in the parallel problem, there are more cases in which multiple proof obligations restrict significantly the parameter
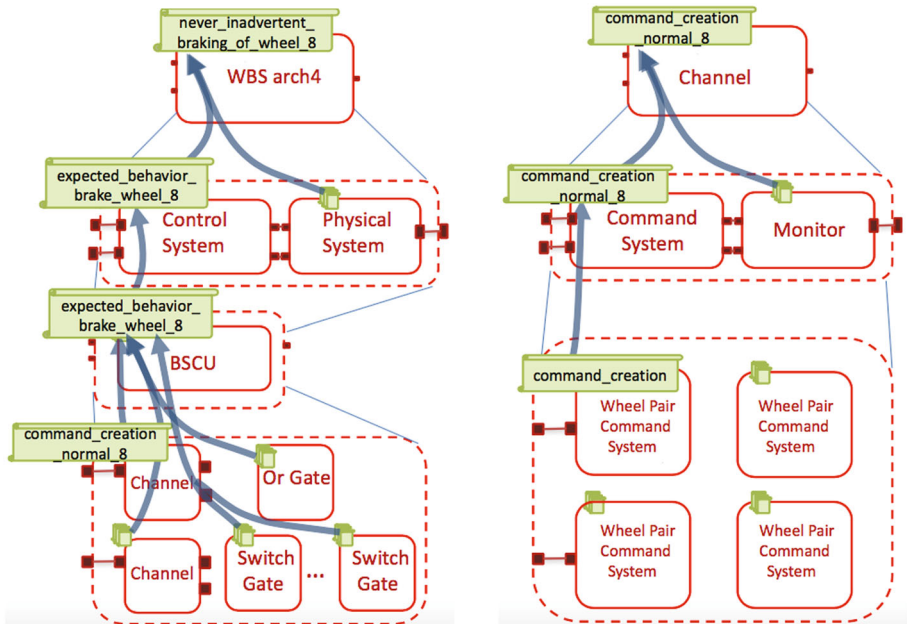
Fig. 9 WBS ARCH4 (simplified version)

region. In fact, as said above, we did not get any simplification in general, which means that the intersection of the parameter regions obtained from the different proof obligation is empty. The encoding approach may take benefit by propagating constraints on the parameters given by one proof obligation to simplify other proof obligations in the monolithic encoding.

All benchmarks have been performed with a time limit of 10 min considering the checking of the contract refinement after tightening, the computation of the parameter region for the encoding and the compositional approach, and the check of the entailments properties. For the 909 contract refinements, 65 could not be completed applying single tightening within the timeout. We have run our experiments on a Linux machine with 8 CPU of 3.40 Ghz Intel Xeon, with a memory of 15 Gb. The benchmarks and executables for reproducing the results are available at https://es.fbk.eu/people/demasi/TightenArch/experiments.html.

6.4 An example of applying tightening on a system architecture

In this subsection we report the evaluation of our tightening techniques on a case study where we have applied these step by step following the process described in Sect. 5 and manually inspecting the results at each step.

We take the most complex architecture, ARCH4, defined in [6] and we apply tightening on the contract refinements from the system component to the leaf components in a top-down fashion. A simplified version of the system architecture of ARCH4[4] is depicted in Fig. 9. We will show the application of our technique only for a subset of contracts just for a reason of space. We start from the system component called `wbs_arch4` which is decomposed into a physical system and a control system. We take one of its

---

[4] A complete documentation for the different architectures defined in [6] can be found in https://es-static.fbk.eu/projects/air6110/.

contracts `never_inadvertent_braking_of_wheel_8` and we apply single tightening. As a result, we obtain 2 tightened variants of this contract refinement. While the new refinements are correct, other contract refinements are broken due to the fact that `never_inadvertent_braking_of_wheel_8` shares some subcontracts with other contracts (e.g., `never_inadvertent_braking_of_wheel_i` for $0 \leq i \leq 7$) defined in the system component like the subcontract `system_validity` from control system component. Then, we run parallel tightening on this contract and we do not get any tightened variant. Therefore, we remove the sharing of contracts by duplication and we obtain two tightened variants as a result. We now take one of these variants and we continue in the decomposition of the contract considering the contract `expected_behavior_brake_as_cmd_8` in the `ControlSystem` component which is just refined by the subcontract `expected_behavior_brake_as_cmd_8` of the `BSCU` component. We run just a single tightening on this contract and we do not obtain any simplification.

We continue going into a top-down fashion and we would like to tighten the contract `expected_behavior_brake_as_cmd_8` in the `BSCU` component which is decomposed into two redundant `Channel`, an `OrGate`, and a 12 `SwitchGate` components. This contract shares several subcontracts with other contracts (e.g., `expected_behavior_brake_as_cmd_j` for $0 \leq j \leq 7$) defined in the `BSCU` component. The shared subcontracts correspond to different subcomponents, e.g., `system_validity`, `command_creation_alternate_k` ($1 \leq k \leq 4$), and `command_creation_normal_l` ($1 \leq l \leq 8$) of the `Channel` component. Also, subcontract `or_behavior` of the `OrGate` component. We first try with parallel tightening but we do not get any simplification. Then, we remove the sharing of contracts by duplication and we obtain as result that several subcontracts are not needed for the contract refinement. In this case that the subcontracts `command_creation_alternate_k` ($1 \leq k \leq 4$) and `command_creation_normal_l` ($1 \leq l \leq 8$) are irrelevant. We have investigated the reason and we observe that the contract `system_validity` is too weak. Therefore, we manually strengthen the contract and we rerun the procedure of duplication and we observe that now two more contracts `channel_1.command_creation_normal_8` and `channel_2.command_creation_normal_8` for the two redundant `Channel` are relevant for this contract refinement.

We continue by tightening the contract `command_creation_normal_8` from the `Channel` component which is refined by the subcontract `command_creation_normal_8` of `CommandSystem` component. This contract from the `Channel` component does not share contracts. By applying tightening, we do not obtain any simplification.

We continue in a top-down fashion where we observe that the `Channel` component is decomposed into component `MonitorSystem` and `CommandSystem`, where the first one is a leaf component which means that we have covered one branch of decomposition and the refinements of contracts from the system component. Subsequently, we tighten the contract `command_creation_normal_8` of `CommandSystem` component. This component is decomposed in four `WheelPairCommandSystem` components. The contract `command_creation_normal_8` is refined by contract `command_creation` of `WheelPairCommandSystem` component. This is shared by contract `command_creation_normal_4` of `CommandSystem`. Thus, we run parallel tightening on the contract and we do not get any tightened variant. Then, we try by removing the sharing by duplication and we get a simplified version of the contract refinement `command_creation_normal_8`. Finally, as `CommandSystem` is a leaf component

we have covered another branch of decomposition and the contracts refinement from the system component.

We conclude that the tightening techniques can support the designer during the process for the contracts specification and refinement and also when the specification of the architecture is already complete like in this case.

6.5 Discussion of the results

As shown by the results in Fig. 4, contract tightening is very effective and can simplify some formulas aggressively. This may be found surprising, especially for models coming from a solid benchmark, like the one presented in [6]. Investigating further the reason why such simplifications are possible, we discovered that in some cases there are indeed shared contracts. We evaluated these cases using the *parallel tightening* technique to see when the simplifications could be maintained keeping correct all refinements in the specification and the result showed that in the case of [6] none of such simplifications was possible. This means that in order to apply the tightening we have to first separate the refinements by either duplicating or removing the shared subcontract. We have applied on these cases the procedure for removing the sharing of contracts by duplication and we have obtained significant simplifications. This strengthens the usability of tightening: the designer typically focuses on having the contract refinement correct and reuse as much as possible existing contracts without caring about the burden of the engine due to the additional redundant contracts. In a second phase, tightening can be run to eliminate such redundancy in a fully automatic way. On the other hand, we have observed in some situations that redundancy is introduced by the designer on the functionality of the components in order to deal with failures of the system. In general, the tightening techniques cannot detect this kind of redundancy and consequently it is removed, so the designer after inspecting the result of tightening has to decide to keep the redundancy or not in this case.

We remark that the tightening technique assists/supports designers on the specification of the contracts and refinement during the development process by simplifying the contract refinements at the level of formulas and contracts. The designers are responsible of analyzing the tightening results and decide for example which one of them is taken for continue with the application of the approach in a top-down fashion. Moreover, duplication of contracts could produce complicated specifications if all duplicates are considered and not significant simplification is obtained. Then, after applying single tightening the designer should analyze the results and taking into account the sharing of contracts, he/she has to decide to keep the original refinement or the result obtained on the duplication of the contracts.

7 Related work

This article is a revised and expanded version of a conference paper presented at *SEFM* 2016 [10] and it extends it as follows: (1) it describes how the tightening techniques can support in general the development process in a top-down fashion for the contracts specification and refinement of a system architecture; (2) it extends the formal definition of tightening to provide the designer with information regarding which contracts are sufficient for (or prevent) correct refinement; (3) it introduces the problem of sharing contracts and provides a solution for this problem called *parallel tightening*, moreover, proposes an automatic approach for removing the sharing of contracts by duplication; (4) it presents a new approach for the multiple validity parameter synthesis problem called *compositional approach*; and (5) it pro-

poses a more thorough experimental analysis, including more insights on the results of single tightening, an evaluation taking into account shared contracts, and a comparison of the two synthesis approaches.

We are not aware of similar works in the context of contract-based design. The problem of contract tightening is related to vacuity checking [24], unsatisfiability core extraction [12]. Unsatisfiability cores are used also to find which portions of a system model are relevant in an inductive proof [21]. Unsatisfiability cores are typically restricted to conjunctions of formulas. The probably most related work is the extension of the notion of unsatisfiability core to temporal formulas expressed in LTL proposed in [31]. However, the design problem, the formal problem, and the technical solution are very different. First, differently from the above-mentioned problems, we are not weakening/strengthening the occurrence of a subformula, but we need to weaken/strengthen all occurrences of an assumption/guarantee inside the proof obligations in the same way. Second, we do not have just one property to simplify, but every assumption/guarantee that is simplified occurs in different proof obligations; this corresponds to different unsatisfiability or model checking problems to consider at the same time. Third, we reduce the problem to a parameter synthesis problem and we ensure the monotonicity of parameters to ensure scalable results.

Also the work described in [23] addresses the problem of simplifying a contract refinement, but with a different purpose and solution: the approach relies on a library of contracts and refinement relations considered as additional inputs to the refinement check problem, and simplifies the contract refinement based on such library. The main objective of the authors is to improve the performance of the refinement check based on the library, while we search for a tighter version of the contracts that still ensure the correctness of the refinement.

8 Conclusions and future work

We have presented in this article a technique to support the development process for the contracts specification and refinement of a system architecture by simplifying the contract refinements along this process. Motivated by the need of validating contract-based designs, we have defined the problem of tightening a contract refinement. This consists of automatically synthesizing simplified versions of the involved assumptions and guarantees, including removing those that are irrelevant in the contract refinement. We consider also the tightening problem within a system architecture that contains multiple contract refinements that share some subcontracts. Our tightening techniques are based on the synthesis of parameters of temporal satisfiability problems. We have evaluated the approach on a number of benchmarks and showed that the solution is effective and scalable. Moreover, we have also manually inspected the results on a significant case study obtaining interesting feedback.

In future research, we plan to extend the approach to consider also the tightening of metric operators and the preservation of realizability. We also plan to detect and avoid the removal of redundancy introduced by the designer for specific goals like incorporating redundancy by adding additional components in order to deal with failures of the system.

References

1. Arts T, Dorigatti M, Tonetta S (2014) Making implicit safety requirements explicit—an AUTOSAR safety case. In: SAFECOMP, pp 81–92
2. Baracchi L, Cimatti A, Garcia G, Mazzini S, Puri S, Tonetta S (2014) Requirements refinement and component reuse: the FoReVer contract-based approach. In: Handbook of research on embedded systems design. IGI Global, pp 209–241
3. Bauer SS, David A, Hennicker R, Larsen KG, Legay A, Nyman U, Wasowski A (2012) Moving from specifications to contracts in component-based design. In: FASE, pp 43–58
4. Benveniste A, Caillaud B, Ferrari A, Mangeruca L, Passerone R, Sofronis C (2007) Multiple viewpoint contract-based specification and design. In: FMCO, pp 200–225
5. Benveniste A, Caillaud B, Nickovic D, Passerone R, Raclet J-B, Reinkemeier P, Sangiovanni-Vincentelli A, Damm W, Henzinger T, Larsen KG (2012) Contracts for system design. Technical report RR-8147, INRIA
6. Bozzano M, Cimatti A, Pires AF, Jones D, Kimberly G, Petri T, Robinson R, Tonetta S (2015) Formal design and safety analysis of AIR6110 wheel brake system. In: CAV, pp 518–535
7. Bozzano M, Cimatti A, Griggio A, Mattarei C (2015) Efficient anytime techniques for model-based safety analysis. In: CAV, pp 603–621
8. Broy M, Huber F, Schätz B (1999) AutoFocus—Ein Werkzeugprototyp zur Entwicklung eingebetteter Systeme. Inform Forsch Entwickl 14(3):121–134
9. Cavada R, Cimatti A, Dorigatti M, Griggio A, Mariotti A, Micheli A, Mover S, Roveri M, Tonetta S (2014) The nuXmv symbolic model checker. In: CAV, pp 334–342
10. Cimatti A, Demasi R, Tonetta S (2016) Tightening a contract refinement. In: Software engineering and formal methods—14th international conference, SEFM 2016, held as part of STAF 2016, Vienna, Austria, July 4–8, 2016, Proceedings, pp 386–402
11. Cimatti A, Dorigatti M, Tonetta S (2013) OCRA: a tool for checking the refinement of temporal contracts. In: ASE, pp 702–705
12. Cimatti A, Roveri M, Schuppan V, Tonetta S (2007) Boolean abstraction for temporal logic satisfiability. In: CAV, pp 532–546
13. Cimatti A, Roveri M, Tonetta S (2009) Requirements validation for hybrid systems. In: CAV, pp 188–203
14. Cimatti A, Roveri M, Tonetta S (2015) HRELTL: a temporal logic for hybrid systems. Inf Comput 245:54–71
15. Cimatti A, Tonetta S (2012) A property-based proof system for contract-based design. In: SEAA
16. Cimatti A, Tonetta S (2015) Contracts-refinement proof system for component-based embedded systems. Sci Comput Program 97:333–348
17. Cofer DD, Gacek A, Miller SP, Whalen MW, LaValley B, Sha L (2012) Compositional verification of architectural models. In: NFM, pp 126–140
18. Damm W, Hungar H, Josko B, Peikenkamp T, Stierand I (2011) Using contract-based component specifications for virtual integration testing and architecture design. In: DATE, pp 1023–1028
19. Ebert C, Jones C (2009) Embedded software: facts, figures, and future. IEEE Comput 42(4):42–52
20. FoReVer project. https://es.fbk.eu/projects/forever/
21. Ghassabani E, Gacek A, Whalen MW (2016) Efficient generation of inductive validity cores for safety properties. In: Proceedings of the 24th ACM SIGSOFT international symposium on foundations of software engineering, FSE 2016, Seattle, WA, USA, Nov 13–18, 2016, pp 314–325
22. Graf S, Passerone R, Quinton S (2011) Contract-based reasoning for component systems with complex interactions. In: TIMOBD'11
23. Iannopollo A, Nuzzo P, Tripakis S, Sangiovanni-Vincentelli AL (2014) Library-based scalable refinement checking for contract-based design. In: DATE, pp 1–6
24. Kupferman O, Vardi MY (2003) Vacuity detection in temporal model checking. STTT 4(2):224–233
25. Manna Z, Pnueli A (1992) The temporal logic of reactive and concurrent systems. Springer, New York
26. Meyer B (1992) Applying "design by contract". Computer 25(10):40–51
27. OCRA website. http://ocra.fbk.eu
28. Pnueli A (1977) The temporal logic of programs. In: FOCS, pp 46–57
29. Quinton S, Graf S (2008) Contract-based verification of hierarchical systems of components. In: SEFM, pp 377–381

30. SAE. AIR 6110 (2011) Contiguous aircraft/system development process example
31. Schuppan V (2012) Towards a notion of unsatisfiable and unrealizable cores for LTL. Sci Comput Program 77(7–8):908–939