Library-Based Scalable Refinement Checking for Contract-Based Design

Antonio Iannopollo*, Pierluigi Nuzzo*, Stavros Tripakis*†, Alberto Sangiovanni-Vincentelli*
* EECS Department, University of California at Berkeley, Berkeley, CA 94720
† Department of Information and Computer Science, Aalto University, Finland
Email: {antonio, nuzzo, stavros, alberto}@eecs.berkeley.edu

***Abstract*—Given a global specification contract and a system described by a composition of contracts, system verification reduces to checking that the composite contract refines the specification contract, i.e. that any implementation of the composite contract implements the specification contract and is able to operate in any environment admitted by it. Contracts are captured using high-level declarative languages, for example, linear temporal logic (LTL). In this case, refinement checking reduces to an LTL satisfiability checking problem, which can be very expensive to solve for large composite contracts. This paper proposes a scalable refinement checking approach that relies on a library of contracts and local refinement assertions. We propose an algorithm that, given such a library, breaks down the refinement checking problem into multiple successive refinement checks, each of smaller scale. We illustrate the benefits of the approach on an industrial case study of an aircraft electric power system, with up to two orders of magnitude improvement in terms of execution time.**

I. Introduction

Contract-Based Design (CBD) has recently emerged as a paradigm for the design of complex systems, emphasizing the concept of interface and requirement formalization to facilitate system integration and provide formal support to the whole design flow, including stepwise refinement of specifications and reuse of pre-designed components [1].

The notion of contracts originates in the context of compositional assume-guarantee reasoning, which has been known for a long time as a verification method for the design of hardware and software, but has been advocated only recently in the context of embedded system design. In CBD, components are specified by contracts and systems by compositions of contracts. In general terms, a *contract* denotes both a set of components that implement a specification and a set of environments in which it can operate. The ultimate objective is to be able to infer "global" properties of complex system by appropriately combining "local" properties of components and their environments. Several rigorous contract theories have then been developed over the years, for instance, assume-guarantee (A/G) contracts [2] and interface theories [3], each using a different formalism to represent the sets of contract implementations and environments. However, the development of efficient algorithms and tools that can support the algebra of contracts and its concrete application to system design is still at its infancy.

This work was partially supported by the NSF Expeditions in Computing project ExCAPE: Expeditions in Computer Augmented Program Engineering, by IBM and United Technologies Corporation (UTC) via the iCyPhy consortium, by TerraSwarm, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, by the Academy of Finland, and by the NSF via project COSMOI: Compositional System Modeling with Interfaces.

978-3-9815370-2-4/DATE14/©2014 EDAA

An important task for the successful deployment of a contract-based methodology is *refinement checking*. In all contract frameworks, given a global specification contract and a system, also described by a composition of contracts, system verification reduces to checking that the composite contract refines the specification contract, i.e. that any implementation of the composite contract implements the specification contract and is able to operate in any environment admitted by it. Even if refinement checking can be carried out compositionally, it can still be very expensive to solve for large composite contracts. For instance, in several applications, contracts can be captured using high-level declarative languages, such as linear temporal logic (LTL). In this case, refinement checking reduces to an LTL satisfiability checking problem, which is PSPACE-complete [4]. Even if contracts are not captured in LTL but instead are expressed directly in an automata-based formalism such as interface automata, for which refinement checking is polynomial [3], the method still suffers from scalability issues due to state explosion. Indeed, the size of the system automaton is often prohibitive, as the system is formed by composing several sub-systems.

In this paper, we took inspiration by the strong growth of library-based design approaches in VLSI where, according to a recent market survey, more than 50% of components at the macro-level come from pre-designed Intellectual Property blocks (IP) that are fully characterized, pre-verified, and fully documented. The cost of building a library of IPs is non trivial but is highly compensated by the saving in design time and cost (International Business Strategies (IBS) estimates that design costs for a chip implemented in the latest technology will exceed 200Million US$ if IP libraries are not used to the fullest extent). Indeed the market for IP blocks is now above 2Billion US$. Motivated by this trend in semiconductor companies, system companies share a growing interest in design reuse both for hardware and software. To make it possible to utilize pre-designed blocks with confidence, providing strong collateral documentation and models is necessary. Along these lines, we show how refinement checking can be made more efficient when a system is described by contracts out of a pre-characterized *library* of components that carry as collateral a characterization in terms of a set of refinement assertions, which we call a *contract library*. We propose an algorithm which, given such a library, breaks down the refinement checking problem into multiple successive refinement checks, each of smaller scale. While our algorithm is not bound to any specific contract framework, to be concrete, we instantiate and demonstrate it by using A/G contracts to describe both the system and the property to be satisfied. A/G contracts specify the behavior of a component by defining what the component guarantees, provided that its environment obeys some given assumptions. We formulate both the component guarantees and the environment assumptions using LTL formulas, a widespread formalism to reason about reactive systems and perform analysis and synthesis of embedded control

software [5], [6], [7]. We illustrate the benefits of the approach on a case study of industrial relevance, i.e. the verification of an embedded controller for an aircraft electric power system, showing up to two orders of magnitude improvement in terms of execution time.

As in traditional assume-guarantee proof strategies, we decompose the main verification task into smaller sub-tasks, where an aggregation of components is replaced by a more abstract representation [8]. However, in most cases, finding the appropriate abstraction is an issue, since no general guidelines are available to the verification engineer. A few approaches have been proposed, which use learning algorithms to automatically build such abstractions, e.g., see [9], [10]. In this paper, the abstraction process is instead guided by the contract library, which systematically encodes the available information on both the structural decomposition of the system architecture and the relevant system domain knowledge. Based on the library, we provide a mechanism to automatically build abstractions on the fly, as we solve the problem by successive refinements. In this respect, our solution is inspired by the platform-based design paradigm [11], where a design at each abstraction layer is also regarded as a platform instance, i.e. a legal interconnection of component out of a pre-characterized library, which also includes composition rules. However, in this paper, the concept of library is further extended to also include refinement rules. As in [12], we exploit the relation between decomposition of component contracts and system architecture and provide a concrete framework to verify a system architecture relying on temporal logic formulas. However, in addition to automatically generate proof obligations, the contribution of our work is twofold: (i) we propose an algorithm to improve the performance of refinement checking, the core verification task underlying any proof obligation in CBD; (ii) we illustrate the benefits of a library-based approach for contract-based verification on a case study of industrial relevance.

II. Background on A/G Contracts

In this Section, we provide an overview of the assume-guarantee contract framework, based on the theory in [2], [1], which we use to instantiate the algorithms in this paper. An *assume-guarantee (A/G) contract* is a pair $C = (A, G)$ where A and G are sets of behaviors defined on a set of variables V. A represents the assumptions that a component makes on its environment, and G represents the guarantees provided by the component. In several applications, such as the specification of reactive controllers, it is also useful to distinguish between input and output variables of a component. We denote a variable as *output variable* if it is controlled by the component. Otherwise it will be an *input variable*, under the responsibility of the environment [1]. We concretely express the sets A and G as formulas in *linear temporal logic* (LTL) [13], each denoting the set of all traces (behaviors) that satisfy it. Each LTL A/G contract can then be represented as a pair of LTL formulas (φ_e, φ_s), which implicitly refer to a set of propositional variables, used in both the assumption and the guarantee formulas.

In what follows, we also assume that contracts are in *saturated form*, meaning that they satisfy $\overline{A} \subseteq G$, where $\overline{A}$ is the complement of A. For an LTL contract, this translates into $(\neg\varphi_e) \rightarrow \varphi_s$, which is not a restrictive assumption, since it can always be satisfied by setting $\varphi_s := \varphi_e \rightarrow \varphi_s$. We then recall two fundamental operations widely used in our verification algorithm, i.e. parallel composition and refinement. The parallel *composition* of two contracts $C_1 = (\varphi_{e1}, \varphi_{s1})$ and $C_2 = (\varphi_{e2}, \varphi_{s2})$ can be directly defined in terms of LTL formulas as

$$C_1 \otimes C_2 = ((\varphi_{e1} \wedge \varphi_{e2}) \vee \neg(\varphi_{s1} \wedge \varphi_{s2}), \varphi_{s1} \wedge \varphi_{s2}).$$

Contract composition preserves the saturated form, that is, if C_1 and C_2 are in saturated form, then so is $C_1 \otimes C_2$. Moreover, $\otimes$ is associative and commutative and generalizes to an arbitrary number of contracts. We therefore can write $C_1 \otimes C_2 \otimes \cdots \otimes C_n$. *Refinement* is instead a preorder on contracts, which formalizes a notion of substitutability. We say that contract $C_1 = (\varphi_{e1}, \varphi_{s1})$ refines contract $C_2 = (\varphi_{e2}, \varphi_{s2})$, written $C_1 \preceq C_2$, if formulas $\varphi_{e2} \rightarrow \varphi_{e1}$ and $\varphi_{s1} \rightarrow \varphi_{s2}$ are both valid, or equivalently, if $\neg(\varphi_{e2} \rightarrow \varphi_{e1})$ and $\neg(\varphi_{s1} \rightarrow \varphi_{s2})$ are both unsatisfiable.

In our framework, we aim to specify systems that are built as aggregation and interconnection of components. To do so, we also define a set of operations that manipulate contract variables. A first operation on contract variables is *instantiation*. Given a set of contracts $\mathcal{C}$, defined on a set of variables $V_\mathcal{C}$ an instance of a contract $C = (\varphi_e, \varphi_s) \in \mathcal{C}$ is a contract $C' = (\varphi'_e, \varphi'_s)$ obtained by C by renaming its variables so that all the variable names $v_1, ..., v_n$ in $\varphi'_e \vee \varphi'_s$ are unique in $V_\mathcal{C}$, i.e. they are not used by any other contract in $\mathcal{C}$. We will indicate the instantiation of a contract C as $inst(C)$. Given three contracts C, C_1 and C_2, where $C_1 = inst(C)$ and $C_2 = inst(C)$, C, C_1 and C_2 will not share any variable. We then define a *renaming* operator. Given a contract $C = (\varphi_e, \varphi_s)$, where φ_e, φ_s are defined on variables $v_1, ..., v_n$, then a renaming for C is a set M of pairs of the form (v_i, u_j), where u_j is a new variable or an existing variable v_k. The renaming operator $ren_M(C)$ returns a new contract $C' = (\varphi'_e, \varphi'_s)$ where variables in φ_e, φ_s are renamed according to M. We will say that two contracts C_1 and C_2 are *isomorphic* if there exists a renaming M such that $ren_M(C_1) = C_2$. With the exception of their variable names, isomorphic contracts represent the same contract. Finally, given a contract C, we define the operations $input(C)$ and $output(C)$, which return, respectively, the list of input and output variables of C.

III. Problem formulation

A. The Refinement Check Problem (RCP)

Given a set of contracts $\mathcal{C}$, a composition of contracts specifying a system $C_s = ren_M(inst(C_1) \otimes inst(C_2) \otimes \cdots \otimes inst(C_n))$, where $C_1, \ldots, C_n \in \mathcal{C}$ and M is a renaming, and a property expressed as a contract C_p, to ensure that any implementation of C_s satisfies C_p and can operate in all environments admitted by C_p, we need to verify that $C_s \preceq C_p$. We denote this verification task as the *refinement check problem* (RCP).

For LTL A/G contracts, RCP can be solved using LTL satisfiability solving techniques, which suffers from the well-known state-explosion problem. In subsequent sections, we will refer to RCP indicating a routine that solves the refinement problem using such techniques. To perform such task more efficiently, we recur to a different problem formulation, which relies on a *library of contracts* as an additional input.

B. Library of contracts and library verification problem (LVP)

Formally, a library of contracts L is a pair $(\mathcal{C}, \mathcal{R})$ where:

- $\mathcal{C} = \{C_1, ..., C_n\}$ is a finite set of contracts.
- $\mathcal{R}$ is a finite set of *refinement relations* between contracts in $\mathcal{C}$. Every refinement relation has the form $R_i = (C_{Ri}, C_{Ai}, M_i)$, where $C_{Ri} = ren_{M_i}(inst(C_{i_1}) \otimes \cdots \otimes inst(C_{i_k}))$ and $C_{Ri} \preceq C_{Ai}$ for $k > 1$, $C_{i_1}, ..., C_{i_k}, C_{Ai} \in \mathcal{C}$, and M_i a renaming for contract $inst(C_{i_1}) \otimes \cdots \otimes inst(C_{i_k})$. If $k = 1$, we require $C_{Ri} \prec C_{Ai}$, that is, C_{Ri} strictly refines C_{Ai}, meaning that the two contracts cannot have equivalent

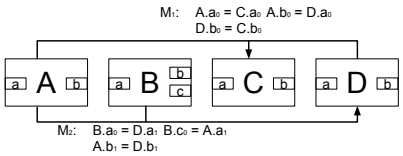

Figure 1. Example contract library with refinement assertions.

formulas. This constraint is introduced to avoid, in the library, the presence of circular dependencies, and therefore ensure termination of the algorithms presented below. We will call the contract C_{i_1} the *root* of R_i.

Refinement relations are *assertions* made by library designers based on their knowledge of the system architecture at hand. We say that the library L is *valid* if all its refinement relations are true. The *library verification problem* (LVP) is the problem of checking whether a given library is valid.

Figure 1 shows an example of a contract library and its refinement relations. In this case, $L_{ex} = (\mathcal{C}_{ex}, \mathcal{R}_{ex})$ where $\mathcal{C}_{ex} = \{A, B, C, D\}$, and $\mathcal{R}_{ex} = \{R_1, R_2\}$ with $R_1 = (ren_{M_1}(inst(A) \otimes inst(D)), C, M_1)$ and $R_2 = (ren_{M_2}(inst(A) \otimes inst(B)), D, M_2)$.

C. The Refinement Check Problem with Library (RCPL)

When a library of contracts defined as in Section III-B is available as an additional input, a *refinement check problem with library* (RCPL) can be formulated as follows. Given a property contract C_p, a contract library $L = (\mathcal{C}, \mathcal{R})$ and a system contract $C_s = ren_M(inst(C_1) \otimes inst(C_2) \otimes \cdots \otimes inst(C_n))$, for a renaming M, and such that $C_1, C_2, \ldots, C_n \in \mathcal{C}$, check whether $C_s \preceq C_p$.

IV. Scalable Contract Refinement Checking

A. Library Verification

Given a library defined as in Section III-B, the library verification process ensures that all its refinement assertions are correct. If any of such refinement relations is not verified, the returned value of the algorithm will be `false`. A description of the library verification process is given in Algorithm 1.

Algorithm 1: *Library Verification Problem.*

Input: A library of contracts, $L = (\mathcal{C}, \mathcal{R})$.
Output: `true`, if all refinement relations in the library are true, `false` otherwise.

1) For each tuple $(C_{Ri}, C_{Ai}, M_i) \in \mathcal{R}$, $C_{Ri} = ren_{M_i}(inst(C_{i_1}) \otimes \cdots \otimes inst(C_{i_k}))$, $k \geq 1$, $C_{i_1}, ..., C_{i_k}, C_{Ai} \in \mathcal{C}$, and M_i a renaming
   a) if $k > 1$ and $C_{Ri} \not\preceq C_{Ai}$ then return an error.
   b) if $k = 1$ and $C_{Ri} \not\prec C_{Ai}$ then return an error.
2) If no errors are found, then return `true`.

Each refinement check in Algorithm 1 is performed by solving an RCP instance as described in Section III-A, which is reasonable in terms of computation time, since aggregations of library contracts are expected to have a small size. Moreover, the overall efficiency of the LVP is deemed to be less critical since it is performed only once, outside of the main verification flow.

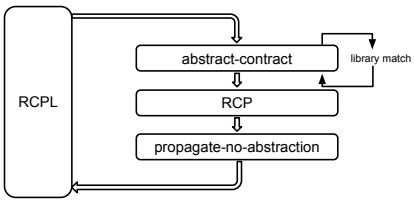

Figure 2. Pictorial representation of the RCPL algorithm.

B. Refinement check with library

Our refinement checking procedure is described in Algorithms 2, 3 and 4. We start with a valid library $L = (\mathcal{C}, \mathcal{R})$, defined as in Section III-B, a property contract C_p (where possibly $C_p \notin \mathcal{C}$), and a system contract C_s, obtained as the composition of a set of contracts $\mathcal{S} = \{C_1, \ldots, C_n\}$, $C_1, \ldots, C_n \in \mathcal{C}$, after instantiation and renaming, as defined in Section III. The system contract C_s represents the specification of a complex system, while the property contract C_p captures a requirement that must be satisfied by the system. We further assume that, given a variable v such that $v \in output(C_i),\ C_i \in \mathcal{S}$, then $v \notin output(C_j),\ C_j \in \mathcal{S}$ and $j \neq i$, meaning that each variable is controlled only by one contract in $\mathcal{S}$ or by a legal environment of C_s.

We solve the RCPL using the algorithm represented in Figure 2 and consisting of two nested loops. In the inner loop, the procedure AbstractContract tries to create a maximal abstraction for C_s given the refinement assertions in L and an indication about which contracts can be abstracted. As a result, some of the contracts in $\mathcal{S}$ will be replaced by an equal or smaller number of more abstract contracts, resulting in a composition that we will denote as C_{abstr}. Since, in general, a more abstract contract is expressed by smaller formulas, C_{abstr} will be simpler and more compact than C_s. The indication on which contracts can be abstracted is provided via the outer loop by the routine PropagateNoAbstraction.

In the outer loop, refinement between C_{abstr} and C_p is checked by the RCP routine. If $C_{abstr} \preceq C_p$ holds, then $C_s \preceq C_p$ will also hold since, by construction, we have $C_s \preceq C_{abstr}$ and the RCPL routine terminates. If the property is not verified at the current level of abstraction, subsequent iterations will use a less and less abstracted representation of C_s. In the worst case, no abstraction is performed and RCPL reduces to an instance of RCP with the not abstract contract. The outer loop of the RCPL procedure is illustrated in Algorithm 2. To control the level of abstraction, each contract (including C_p) has an associated Boolean flag that corresponds to a *no-abstraction* constraint. If the flag is `true`, the contract will not be substituted by a more abstract one, even this is available in the library. As shown in line 6 in Algorithm 2, the main loop terminates when $C_{abstr} \preceq C_p$ or when the function AbstractContract cannot return a more abstract contract.

Algorithm 2: *RCPL*

Input: A contract C_p, a library of contracts $L = (\mathcal{C}, \mathcal{R})$, a contract C_s obtained by composition of $C_1, \ldots, C_n \in \mathcal{C}$.
Output: `true`, if $C_s \preceq C_p$, `false` otherwise.

1) Let $S = [C_1, \ldots, C_n, C_p]$.
2) Build hashtable A such that $A[C_p] =$ `true` and $\forall i$, $A[C_i] =$ `false`.
3) S' := AbstractContract(A, L, C_s)
4) $C_{abstr} := \otimes\{C_i \mid C_i \in S'\}$
5) If $C_{abstr} \preceq C_p$ return `true`.
6) Create a copy $C_{s'} = C_s$.

7) While $C_{abstr} \not\preceq C_p$ and $C_{abstr} \neq C_{s'}$
   a) $C_{s'} := C_{abstr}$,
   b) A := PROPAGATENOABSTRACTION(A, S),
   c) S':=ABSTRACTCONTRACT(A, L, C_s)
   d) $C_{abstr} := \otimes\{C_i \mid C_i \in S'\}$
8) If $C_{abstr} \preceq C_p$ return `true`, otherwise `false`

Algorithm 3: ABSTRACTCONTRACT

Input: A library of contracts $L = (\mathcal{C}, \mathcal{R})$, a composite contract C_s (obtained by composition of $C_1, \ldots, C_n \in \mathcal{C}$), a hashtable A as in Algorithm 2.
Output: A list of contracts $S = [C_{a_1}, \ldots, C_{a_m}]$, such that $C_s \preceq C_{a_1} \otimes \cdots \otimes C_{a_m}$.

1) Create $S := [C_1, \ldots, C_n]$.
2) Create $C_{abstr} :=$ `null`.
3) Create a copy $C_{s'} := C_s$.
4) While $C_{abstr} \neq C_{s'}$
   a) assign $C_{s'} = C_{abstr}$,
   b) create copy $S' := S$;
   c) for each contract $C_k \in S \cap S'$ that satisfies *abstraction-acceptance-condition* then
      i) $S' := (S' - [C_k, C_{k_1}, \ldots, \mathcal{C}_{k_m}]) \cdot [ren_N(C_{Ai})]$;
      ii) $C_{abstr} := \otimes\{C_i \mid C_i \in S'\}$.
   d) $S := S'$;
5) return S

The procedure ABSTRACTCONTRACT in Algorithm 3 implements the inner loop of RCPL. It accepts as inputs a library $L = (\mathcal{C}, \mathcal{R})$, a contract C_s composed of contracts in $\mathcal{C}$, and a list of flags A built as described in Algorithm 2. The algorithm tries to abstract C_s by using the information in L until no progress is made. Line 4 describes how the abstraction is performed in terms of the operations defined in Section II. At each iteration, a copy of the current list of contracts S is maintained in S' and the *abstraction-acceptance-condition* is checked on each contract $C_k \in S \cap S'$. If it evaluates to true, a subset of contracts is matched to an aggregation of contracts C_{Ri} in L and then replaced by its abstraction C_{Ai}. The *abstraction-acceptance-condition* requires the following sub-conditions to hold:

- C_k is not flagged by a *no-abstraction* constraint, that is $A[C_k] =$ `false`;
- $\exists R_i = (C_{Ri}, C_{Ai}, M_i) \in \mathcal{R}$ such that C_k and the root of R_i are isomorphic;
- $\exists C_{k_1} \ldots C_{k_m} \in S \cap S'$, such that $A[C_{k_1}] = \cdots = A[C_{k_m}] =$ `false`, and a renaming N such that $ren_N(C_{Ri}) = C_k \otimes C_{k_1} \otimes \cdots \otimes \mathcal{C}_{k_m}$, i.e. there exists a subset of contract that can be abstracted and such that, when composed with C_k, generate a contract that is isomorphic with C_{Ri};
- $(output(ren_N(C_{Ri})) \setminus output(ren_N(C_{Ai}))) \cap input(C_r) = \emptyset$, $C_r \in S' \setminus \{C_k, C_{k_1}, \ldots, \mathcal{C}_{k_m}\}$, i.e. no substitution is made if there is some other contract in C_s, which is not in $\{C_k, C_{k_1}, \ldots, C_{k_m}\}$, and such that at least one of its input variables is missing in the abstract contract.

The replacement of the contracts in the original list as well as the selection of candidate abstractions from the library are currently performed in a random order. More sophisticated heuristics will be considered in future implementations.

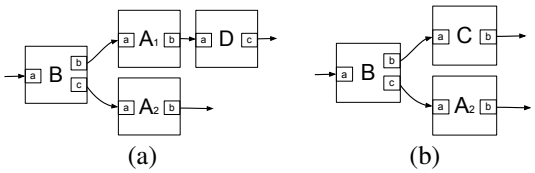

Figure 3. Representation of a composite contract obtained from the library in Figure 1 (a) and its abstraction (b).

Termination of ABSTRACTCONTRACT is guaranteed since contract C_{abstr} will not change after a certain number of iterations. In fact, the number of matchings of contracts in $\mathcal{R}$ performed in line 4.c) is finite. Therefore, since the library is finite, we just need to prove the absence of circular dependencies between contract relations in $\mathcal{R}$. To show this, we observe that for each matching relation $R_i = (C_{Ri}, C_{Ai}, M_i)$, with $C_{Ri} = ren_{M_i}(inst(C_{i_1}) \otimes \cdots \otimes inst(C_{i_k}))$, there are two possible cases. If $k > 1$, after replacing C_{Ri} with C_{Ai}, the number of contracts in S decreases. Obviously, this operation can only be performed a finite number of times. On the other hand, if $k = 1$, since we requires $C_{Ri} \prec C_{Ai}$, we will always have $C_{Ai} \not\prec C_{Ri}$. Therefore, it is impossible to find in the library a relation $R'_i = (C_{Ai}, C_{Ri}, M_i)$, which would represent a circular dependency between R_i and R'_i.

The runtime of ABSTRACTCONTRACT is mostly determined by the time it takes to find a matching between a set of library contracts and a subset of the contracts composing C_s. Such a matching problem can be reduced to a graph isomorphism problem, which can be efficiently solved [14], [15]. In our case, graphs can be built to represent contract compositions, while incorporating information on the names of the variables of the component contracts and their isomorphism.

Algorithm 4: PROPAGATENOABSTRACTION

Input: A list of contracts $S = [C_1, \ldots, C_n]$, a hashtable A as in Algorithm 2.
Output: A new hashtable A.

1) Create list $M := \emptyset$
2) For each contract $C_k \in S$ such that $A[C_k] =$ `true`
   a) for each contract $C_h \in S$ such that $(input(C_k) \cup output(C_k)) \cap output(C_h) \neq \emptyset$
      i) add C_h to M,
3) For each contract $C_i \in M$, assign $A[C_i] =$ `true`
4) return A.

The heuristic used in the propagation of the *no-abstraction* constraint is finally detailed in Algorithm 4. We propose an incremental propagation of the constraint according to the syntactical dependence between contracts. The algorithm receives as a parameter the list of contracts that compose C_s, extended with the addition of the property contract C_p (the first to receive the *no-abstraction* mark). Each time PROPAGATENOABSTRACTION is called, the *no-abstraction* mark will be propagated to all contracts that share at least one of their output variables with a marked contract. This approach is similar to the concept of "cone of influence" used in Counterexample-Guided Abstraction Refinement [16].

We provide an example of execution of our algorithm in Figure 3. The contract in Figure 3 (a) is obtained by composition of contracts from the library in Figure 1. The arrows denote a renaming operation. We assume that the property contract C_p involves only variables $B.a$ and $D.c$. We then

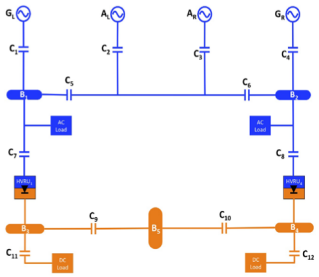

Figure 4. Aircraft electric power system plant architecture used in the case study.

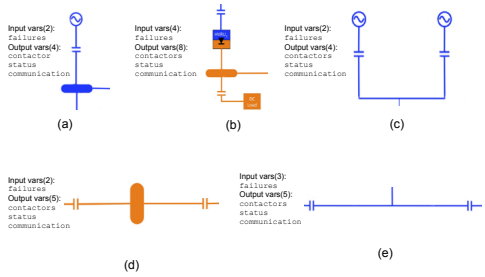

Figure 5. Subsets of components of the EPS plant and number of variables associated with the related contracts, including communications variables and variables related to the health status of plant components (e.g. buses, contactors).

call the RCPL algorithm using C_p, C_s in Figure 3 (a), and L_{ex} from Section III-B. At the first execution of AbstractContract, all contracts can be potentially abstracted. However, there are only two possible matchings between portions of the architecture in Figure 1 and the refinement relations in $\mathcal{R}_{ex}$. In particular, the composite contract $B \otimes A_2$ can be abstracted as D_1, an instance of D, while $A_1 \otimes D$ can be abstracted as C. However, $B \otimes A_2$ does not satisfy the last condition for the *abstraction-acceptance-condition* to hold in line 4.c) of Algorithm 3. In fact, replacing $B \otimes A_2$ with D_1 would cause the loss of a variable ($B.b$) that should be shared with A, hence an incorrect abstraction. Conversely, the substitution of $A_1 \otimes D$ with C is legal and the resulting contract composition, C_{abstr}, is shown in Figure 3 (b). If $C_{abstr} \preceq C_p$, the algorithm would terminate by executing an instance of the RCP on a more compact representation of the system contract. Otherwise, if $C_{abstr} \not\preceq C_p$, PropagateNoAbstraction would mark D with a *no-abstraction* annotation. At this point, no contract aggregation can be further abstracted, and the algorithm terminates by solving an instance of the RCP on the original composition.

V. Application Example

The proposed algorithm was implemented in Python and applied on the verification of a controller for an aircraft electric power system (EPS). To solve the LTL satisfiability problems, we used NuSMV [17]. All tests were performed on a 2.3-GHz Intel Core i7 machine with 8 GB of RAM.

Figure 4 shows the architecture of the EPS plant. The set of components includes primary generators (G_L, G_R), auxiliary generators (A_L, A_R) on both the left and right side of the diagram, contactors ($c_1, ..., c_{12}$), buses ($B_1, ..., B_5$), high-voltage rectifier units (HVRU$_i$) and loads. The EPS controller must appropriately open or close the contactors (electromechanical switches) to ensure that loads are always adequately powered by accommodating the highest possible number of failures in the components. In our example, we assume that failures can only affect generators and rectifiers.

Each contract in our library specifies a "local" controller for a portion of the EPS plant, i.e. a subset of its components. In addition to sensing (input) and actuation (output) variables, contracts can include a set of communication variables to propagate information on error conditions and component health status. Figure 5 shows some of the EPS subsystems supported by our library. A contract for the subsystem in Figure 5.a) specifies that the contactor should be opened and the failure variable asserted if the generator fails; otherwise the contactor must be closed. For the subsystem in Figure 5.c), the same requirement as for Figure 5.a) will hold, with the addition that both generators should never be connected at the same time to avoid paralleling AC sources. For the subsystem in Figure 5.e), we require that the contactor on one side should be closed upon reception of a failure signal from a component connected to the opposite side. The contract for the subset in Figure 5.b) specifies that the load should be isolated in case of failures in one of the interconnected portions of the plant, or in the HVRU. Finally, the subsystem in Figure 5.d) is associated to a contract similar to the one in Figure 5.e), while handling one additional bus and only two interconnection branches. For each portion of the plant, the library can provide multiple contracts to specify different sets of behaviors. Moreover, we provide contracts that specify abstractions of controllers for specific portions of the plant. For example, a contract may represent the behavior of the controller associated to an idealized generator, which abstracts both the sub-systems in Figure 5.a) and 5.c). Overall, the library includes 17 contracts and 9 refinement assertions. The verification of the refinement assertion using NuSMV required 1.55 s.

A controller for the EPS has been assembled out of 5 different contracts from the library, associated to the subsystems shown in Figure 5. The composite contract has a total of 46 variables. On the other hand, the most compact abstraction of the design based on the available library had only 12 variables. On this design, we checked the following properties expressed as LTL A/G contracts:

- $C_{p1}, ..., C_{p4}$: If generator $G \in \{G_L, A_L, A_R, G_R\}$ fails, the closest contactor $c \in \{c_1, ..., c_4\}$ must be opened;
- C_{p5}: If generators G_L and G_R are healthy, contactors c_5 and c_6 must be opened;
- C_{p6}: Contactors c_2 and c_3 cannot be both closed at the same time;
- $C_{p7}, ..., C_{p10}$: If at least one generator is healthy, bus $B \in \{B_1, ..., B_4\}$ cannot stay unpowered for more than three clock cycles;
- C_{p11}: If all generators are healthy, bus B_5 must not be powered;
- C_{p12}, C_{p13}: If at least one generator is healthy, c_{11} and c_{12} cannot stay opened for more than three clock cycles.

This set of property contracts has been verified using both the RCPL and the RCP algorithms. The total execution time was 123.1 s for RCPL, and 638.82 s for RCP. Figure 6 shows the execution times required by each verification task. For more than half of the properties ($C_{p1}, C_{p2}, C_{p3}, C_{p4}, C_{p5}, C_{p6}, C_{p11}$), RCPL allows to obtain a performance improvement of two orders of magnitude, by using an abstraction of the controller with only 12 variables.

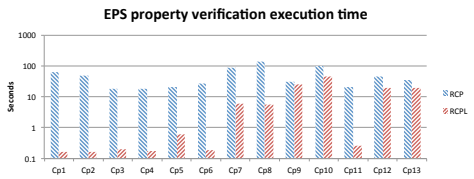

Figure 6. Execution time of RCPL and RCP algorithms for the EPS case study for the verification of the set of 13 property contracts

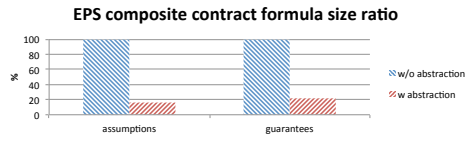

Figure 7. LTL formula size ratio of the abstract EPS contract w.r.t. the non-abstract version

For C_{p7} and C_{p8} RCPL shows a performance improvement of one order of magnitude, while for the other properties, the execution times are comparable to the one obtained with plain RCP. C_{p10} produced the worst execution time, using an abstraction with 37 variables. Figure 7 shows the difference in terms of formula sizes, computed as the ratio between the non-abstract EPS contract size and the one of its maximal abstraction obtained at the first iteration of the ABSTRACTCONTRACT algorithm. Formulas in abstract contracts are indeed smaller than the original ones, which provides an explanation of the performance improvement obtained using RCPL.

To test the scalability of the algorithm, the same properties have been checked on an extended plant architecture, including one more generator, 7 contactors, 2 rectifier units, 2 AC loads, 2 DC loads and one bus. The contract specifying a controller for the new plant includes 66 variables. Verification of the whole property set was performed in 1724.43 s with RCPL and 8371.01 s with RCP. Also in this example, an execution time two orders of magnitude smaller for RCPL has been observed. In the best case, the generated abstract contract included only 16 variables.

VI. CONCLUSION

We addressed the problem of performing scalable refinement checks for contract-based design. We presented an algorithm that leverages a pre-characterized library of contracts enriched with refinement assertions to break the main verification task into a set of smaller tasks. We applied the proposed algorithm to verify controllers for aircraft electrical power systems, with up to two orders of magnitude improvement with respect to a standard implementation based only on LTL satisfiability solving. A full-fledged theoretical study of its complexity is challenging, since its runtime is highly dependent on the characteristics of the library, in addition to the structure of the system and the property under consideration. A characterization of the role of the library via domain-related benchmarks will be object of future work. We here anticipate that the benefits of having a richer library in terms of refinement assertions will largely repay the overhead of building it. In fact, we recall that the library verification process must be performed only once, outside of the main verification flow. Moreover, the proposed algorithm already offers a way of automatically proving new refinement relations that can be effectively used to further populate the original library so as to enrich it for future verification tasks.

Possible extensions of this paper include investigating algorithms for automatic mapping of library contracts to plant architectures, the adoption of learning algorithms for library optimization, and the definition of benchmarks and quality metrics to estimate the effectiveness of a library.

REFERENCES

[1] A. Sangiovanni-Vincentelli, W. Damm, and R. Passerone, "Taming Dr. Frankenstein: Contract-Based Design for Cyber-Physical Systems," *European Journal of Control*, vol. 18, no. 3, pp. 217–238, Jun. 2012.

[2] A. Benveniste, B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, and C. Sofronis, "Multiple Viewpoint Contract-Based Specification and Design," in *Formal Methods for Components and Objects*, 2008.

[3] L. de Alfaro and T. A. Henzinger, "Interface automata," in *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2001, pp. 109–120.

[4] A. P. Sistla and E. M. Clarke, "The complexity of propositional linear temporal logics," *J. ACM*, vol. 32, no. 3, pp. 733–749, 1985.

[5] N. Piterman, A. Pnueli, and Y. Sa'ar, "Synthesis of reactive(1) designs," in *Verification, Model Checking, and Abstract Interpretation*. Springer Berlin Heidelberg, 2006.

[6] N. Ozay, U. Topcu, and R. Murray, "Distributed power allocation for vehicle management systems," in *Decision and Control and European Control Conference (CDC-ECC), 2011 50th IEEE Conference on*, 2011.

[7] T. Wongpiromsarn, U. Topcu, and R. Murray, "Automatic synthesis of robust embedded control software," *AAAI Spring Symposium on Embedded Reasoning: Intelligence in Embedded Systems*, 2010.

[8] O. Grumberg and D. E. Long, "Model checking and modular verification," *ACM Transactions on Programming Languages and Systems*, vol. 16, 1991.

[9] J. Cobleigh, D. Giannakopoulou, and C. Pasareanu, "Learning assumptions for compositional verification," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2003, pp. 331–346.

[10] A. Gupta, K. McMillan, and Z. Fu, "Automated assumption generation for compositional verification," *Formal Methods in System Design*, pp. 285–301, 2008.

[11] A. Sangiovanni-Vincentelli, "Quo Vadis, SLD? Reasoning About the Trends and Challenges of System Level Design," *Proceedings of the IEEE*, vol. 95, no. 3, pp. 467–506, 2007.

[12] A. Cimatti and S. Tonetta, "A property-based proof system for contract-based design," in *EUROMICRO Conference on Software Engineering and Advanced Applications*, 2012, pp. 21–28.

[13] A. Pnueli, "The temporal logic of programs," in *Annual Symposium on Foundations of Computer Science*, 1977, pp. 46–57.

[14] J. Torán, "On the hardness of graph isomorphism," *SIAM Journal on Computing*, vol. 33, no. 5, pp. 1093–1108, 2004.

[15] L. Babai and E. M. Luks, "Canonical labeling of graphs," in *Proc. of ACM Symp. on Theory of Computing*, ser. STOC '83, 1983, pp. 171–183.

[16] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, 2000, pp. 154–169.

[17] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "Nusmv 2: An opensource tool for symbolic model checking," in *Proceedings of the 14th International Conference on Computer Aided Verification*, 2002.