From Partial to Global Assume-Guarantee Contracts: Compositional Realizability Analysis in FRET

Anastasia Mavridou[1], Andreas Katis[1], Dimitra Giannakopoulou[2], David Kooi[3], Thomas Pressburger[2], and Michael W. Whalen[4]

[1] KBR, NASA Ames Research Center, CA, USA
[2] NASA Ames Research Center, CA, USA
{anastasia.mavridou, andreas.katis, dimitra.giannakopoulou, tom.pressburger}@nasa.gov
[3] University of California, Santa Cruz, CA, USA dkooi@ucsc.edu
[4] University of Minnesota, MN, USA whalen@cs.umn.edu

Abstract. Realizability checking refers to the formal procedure that aims to determine whether an implementation exists, always complying to a set of requirements, regardless of the stimuli provided by the system's environment. Such a check is essential to ensure that the specification does not allow behavior that can force the system to violate safety constraints. In this paper, we present an approach that decomposes realizability checking into smaller, more tractable problems. More specifically, our approach automatically partitions specifications into sets of non-interfering requirements. We prove that checking whether a specification is realizable reduces to checking that each partition is realizable. We have integrated realizability checking and implemented our decomposition approach within the open-source Formal Requirements Elicitation Tool (FRET). A FRET user may check the realizability of a specification monolithically or compositionally. We evaluate our approach by comparing monolithic and compositional checking and showcase the strengths of our decomposition approach on a variety of industrial-level case studies.

1 Introduction

Defining requirements for a complex system is a challenging, error-prone task. The focus of this paper is on ensuring consistency of system component requirements, thus building a solid foundation for subsequent system-level analysis [12,13,55,4]. For *reactive* systems, which interact with an uncontrollable environment, consistency must be established for all reasonable inputs from the environment, leading to the notion of realizability [50]. Realizability checking, however, comes with challenges. While optimal algorithms exist for finite state problems over subsets of Linear Temporal Logic specifications (LTL) [49,18], scalability issues can make the analysis impractical for realistic systems, and the use of infinite data types can render the entire problem undecidable [20,29].

This work makes the following contributions for checking realizability:

1. a novel compositional theoretical framework to check realizability of a global contract through smaller, more tractable parts, named *partial contracts*;
2. an algorithm that identifies, for a global system contract, equivalent partial contracts that can be checked for realizability instead of the global one;
3. implementation of our framework in the open-source Formal Requirements Elicitation Tool FRET [23], and its evaluation on industrial-level projects.

Partial contracts describe requirements that observe only a subset of a global system state. Partial-contract realizability then introduces the notion of realizability of a system with respect to a set of partial contracts. We show that partial-contract realizability is equivalent to checking that every partial contract is realizable, when partial contracts are non-interfering, meaning that they observe disjoint sets of system state variables. Finally, we provide conditions under which checking realizability of a global contract is equivalent to partial-contract realizability. This equivalence enables us to: 1) ensure realizability of a global contract by checking that each one of its partial contracts is realizable, and 2) when a partial contract is unrealizable, conclude that the global contract is also unrealizable.

Our decomposition algorithm automatically computes partial contracts that fit the conditions of our theory, based on the notion of connected components for undirected graphs. The evaluation of our compositional approach showcases several benefits as compared to monolithic realizability analysis. Decomposition is key for both scalability and performance of realizability analysis. Moreover, when both monolithic and compositional analyses fail to complete, compositional analysis may still be able to return results for some of the partial contracts. Finally, for unrealizable contracts, our approach is able to attribute the causes of unrealizability to partial contracts, which are generally easier to debug.

Specification decomposition has also been studied, independently, in a recent work by Finkbeiner et al. [17], in the context of reactive synthesis. Even though the theoretical formulation of the two works is different due to the respective settings in which they have been developed, they explore similar avenues. We dedicate Subsection 6.2 of our evaluation section to providing a detailed comparison of the two approaches.

2 Liquid Mixer System Example

We use as running example the controller of a liquid mixing system [35] (see Figure 1), whose behavior must satisfy the 12 requirements shown in Table 1. To relate our approach with the EARS-CTRL approach presented in [35], we took the 12 requirements expressed in EARS-CTRL and translated them into FRETish, the requirements language of FRET.

A FRETish requirement consists of up to six fields: `scope`, `condition`, `component*`, `shall*`, `timing`, and `response*`. Mandatory fields are indicated by an asterisk. `component` specifies the component that the requirement refers to. `shall` is used to express that the component's behavior must conform to the requirement. `response` is a Boolean condition that the component's behavior

Table 1: Liquid mixer system requirements in English and FRETish.

Req ID	Original Requirement Text	Requirement in FRETish
[LM-001]	While not liquid level 1 is reached, when start button is pressed the liquid mixer controller shall open valve 0	`when start_button` the liquid_mixer `shall` immediately satisfy if ! liquid_level_1 then valve_0
[LM-002]	When liquid level 1 is reached occurs, the liquid mixer controller shall close valve 0	`when liquid_level_1` the liquid_mixer `shall` immediately satisfy ! valve_0
[LM-003]	While not liquid level 2 is reached, when liquid level 1 is reached the liquid mixer controller shall open valve 1 until emergency button is pressed.	`when liquid_level_1` the liquid_mixer `shall` until emergency_button satisfy if ! liquid_level_2 then valve_1
[LM-004]	When liquid level 2 is reached occurs, the liquid mixer controller shall close valve 1.	`when liquid_level_2` the liquid_mixer `shall` immediately satisfy ! valve_1
[LM-005]	When liquid level 2 is reached occurs, the 60 sec timer shall start.	`when liquid_level_2` the liquid_mixer `shall` immediately satisfy timer_60sec_start
[LM-006]	When liquid level 2 is reached happens, liquid mixer controller shall start stirring motor until 60 second timer expires or emergency button is pressed.	`when liquid_level_2` the liquid_mixer `shall` until (timer_60sec_expire | emergency_button) satisfy stirring_motor
[LM-007]	When 60 second timer expires occurs, the 120 sec timer shall start.	`when timer_60sec_expire` the liquid_mixer `shall` immediately satisfy timer_120sec_start
[LM-008]	When 60 second timer expires happens, the liquid mixer controller shall open valve 2 until 120 sec timer expires or emergency button is pressed.	`when timer_60sec_expire` the liquid_mixer `shall` until (timer_120sec_expire | emergency_button) satisfy valve_2
[LM-009]	When emergency button is pressed occurs, the liquid mixer controller shall close valve 0.	`when emergency_button` the liquid_mixer `shall` immediately satisfy ! valve_0
[LM-010]	When emergency button is pressed occurs, the liquid mixer controller shall close valve 1.	`when emergency_button` the liquid_mixer `shall` immediately satisfy ! valve_1
[LM-011]	When emergency button is pressed occurs, the liquid mixer controller shall close valve 2.	`when emergency_button` the liquid_mixer `shall` immediately satisfy ! valve_2
[LM-012]	When emergency button is pressed occurs, the liquid mixer controller shall stop stirring motor.	`when emergency_button` the liquid_mixer `shall` immediately satisfy ! stirring_motor

must satisfy. `scope` specifies the period when the requirement holds. The optional `condition` field is a Boolean expression that further constrains when the `response` shall occur. The `timing` field, e.g., *always*, *after N time units*, specifies when the response shall happen, subject to `condition` and `scope`.

The original text of the Liquid Mixer requirements and their FRETish versions are shown in Table 1. We used the following variables to write requirement **[LM-001]** in FRETish: 1) *liquid_level_1* that evaluates to true when liquid level 1 is reached; 2) *start_button* that becomes true when the start button is pressed; 3) *valve_0* that evaluates to true while valve 0 is open. This requirement refers to the `liquid_mixer` component. We omit the `scope`, which means that the requirement holds during the entire execution. FRET conditions trigger a requirement when their corresponding boolean expression becomes true from false. In this case, every time *start_button* becomes true (from false) the response if ! *liquid_level_1* then *valve_0* must hold.

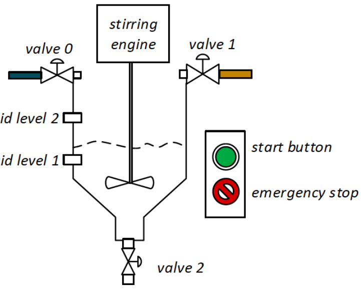

Fig. 1: Liquid Mixing System (figure taken from Lúcio et al. [35]).

3 Background on Realizability

This section provides background on modeling requirements as Assume-Guarantee contracts and on the notion of realizability.

3.1 Assume-Guarantee (AG) Contracts

We rely on the notion of AG contracts as defined by previous work on JSyn and JSyn-vg [20,29]. We use two types *state* and *inputs* for a transition system (I, T) where predicate $I(s) : \mathit{state} \rightarrow \mathit{bool}$ denotes the set of initial states, and predicate $T(s, a, s') : \mathit{state} \times \mathit{inputs} \times \mathit{state} \rightarrow \mathit{bool}$ is the system's transition relation from states s to primed states s', given inputs a. State variables represent both internal and output variables of the system.

A contract (A, G) for system (I, T) consists of an assumption predicate $A(s, a) : \mathit{state} \times \mathit{inputs} \rightarrow \mathit{bool}$ and a guarantee G, made up of two predicates: $G_I(s) : \mathit{state} \rightarrow \mathit{bool}$ and $G_T(s, a, s') : \mathit{state} \times \mathit{inputs} \times \mathit{state} \rightarrow \mathit{bool}$, capturing initial-state and transitional guarantees, respectively. In practice, as described in Section 5, A and G may be expressed as sets of predicates, with A and G corresponding to their conjunctions. Note that, any behavior following an environmental input that violates the contract's assumptions is unrestricted by the contract.

Consider the FRETish liquid mixer system requirements of Table 1. The state variables are: {*stirring_motor, valve_0, valve_1, valve_2, timer_60sec_start, timer_120sec_start*}. The input variables are: {*emergency_button, start_button, liquid_level_1, liquid_level_2, timer_60sec_expire, timer_120sec_expire*}. All variables involved in this system are of type boolean. Let us take input variable *liquid_level_1*, for example. We use *liquid_level_1* and !*liquid_level_1* to represent a true or false valuation for it.

The liquid mixer system does not involve any assumptions or initial guarantees, so for all s, a, $A(s, a) = \mathit{true}$ and $G_I(s) = \mathit{true}$. Moreover, $G_T(s, a, s') = \mathit{true}$ if and only if the transition satisfies all requirements of Table 1, i.e., their conjunction. The FRET tool automatically produces requirement formalizations in a variety of languages, as well as generates and exports analysis code, e.g., CoCoSpec[5] code [40]. For the purposes of this work, we have extended the code generation functionality of FRET to support Lustre code that is digested by the JSyn and JSyn-vg procedures of the JKind model checker. The generated Lustre code captures the transition relation of an AG contract.

3.2 Realizability

An AG contract is *realizable* if there exists a system implementation that satisfies the contract guarantees for all assumption-complying stimuli provided by the environment. As mentioned above, any behavior following an environmental input that violates the contract's assumptions is unrestricted by the contract.

[5] CoCoSpec [9] is a contract-based extension of the Lustre synchronous language.

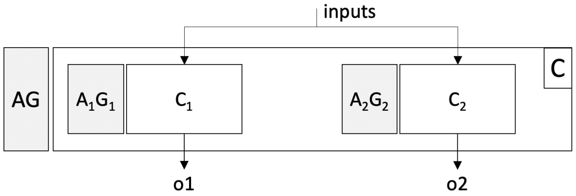

Fig. 2: Partial Assume-Guarantee Contracts.

Definition 1 (Viability of an AG contract). *A set of viable [20] system responses is defined coinductively, as the greatest fixed point of the following equation:*

$$\mathsf{Viable}_{AG}(s) = \forall a.\ (A(s,a) \Rightarrow \exists s'.\ G_T(s,a,s') \wedge \mathsf{Viable}_{AG}(s'))$$

Realizability of a contract (A, G) is then defined as follows:

Definition 2 (Realizability of an AG contract).

$$\mathsf{Realizable}_{AG} \overset{\text{def}}{=} \exists s.\, G_I(s) \wedge \mathsf{Viable}_{AG}(s)$$

For realizability checking, we use a combination of two off-the-shelf algorithms, namely JSyn [20,30] and JSyn-vg [29]. These algorithms are automated; the engineer does not need to be actively involved during analysis. Moreover, both algorithms are agnostic with respect to the theories that may be exercised within the specification, allowing for a wide range of supported expressions. As of this paper, JSyn and JSyn-vg employ techniques that perform over the theories of Linear Integer and Real Arithmetic (LIRA). Since the input specification can admit infinite theories, the overall problem of realizability checking is undecidable. Problem decomposition is therefore an attractive means of dividing the original challenge into subproblems of smaller size. Nevertheless, decomposition over quantified formulas is not straightforward.

4 Decomposing Realizability

Our theory of compositional realizability checking is based on the notion of partial contracts, i.e., contracts that observe only part of the state of a target system. We use the example of Figure 2 to provide intuition for the concepts that we present. In the example, a component C, is made up of components C_1 and C_2, each with their individual contracts (A_1, G_1) and (A_2, G_2). Note that we do not consider the case where C_1 and C_2 communicate with each other; in other words, outputs[6] of C_1 do not intersect with input variables of C_2, and vice versa. Note also that we study the simple case where the components share inputs. We can generalize this case later. In the context of component C, contracts (A_1, G_1) and (A_2, G_2) are partial contracts, as defined below.

[6] Note that output and internal variables are considered state variables.

Let us assume a set of types $T = \{T_1, \ldots, T_k\}$ and a set of typed state variables $SV = \{s_1 : T_1, \ldots, s_k : T_k\}$. Let $\mathit{STATES} \overset{\text{def}}{=} T_1 \times \ldots \times T_k$. We use k-dimensional vectors $(v(s_1), \ldots, v(s_k))$ to represent states that range over *STATES*, where $v(s_i) \in T_i$ is the valuation of variable s_i.

For a state $s = (v(s_1), ..., v(s_k)) \in \mathit{STATES}$, and a subset $SV_i \subseteq SV$ over some of its state variables:

$$s@SV_i \overset{\text{def}}{=} (v(s_j) \mid s_j \in SV_i)$$

In other words, operator @ maps a state vector s to a sub-vector based on the subset of variables in SV_i. We can extend this operation to sets of states:

$$\mathit{STATES}@SV_i \overset{\text{def}}{=} \{s_i \mid \exists s \in \mathit{STATES}.\ s_i = s@SV_i\}.$$

Let (I, T) be a transition system over SV, with $I(s) : \mathit{STATES} \rightarrow \mathit{bool}$ and $T(s, a, s') : \mathit{STATES} \times \mathit{inputs} \times \mathit{STATES} \rightarrow \mathit{bool}$. In Section 3, we defined AG contracts (A, G) over the states and inputs of a system. In this section, we consider partial contracts (A_i, G_i) that only refer to some state variables $SV_i \subseteq SV$ of the system, i.e.:

$$\begin{aligned}
&A_i : \mathit{STATES}@SV_i \times \mathit{inputs} \rightarrow \mathit{bool}\\
&G_{Ii} : \mathit{STATES}@SV_i \rightarrow \mathit{bool}\\
&G_{Ti} : \mathit{STATES}@SV_i \times \mathit{inputs} \times \mathit{STATES}@SV_i \rightarrow \mathit{bool}
\end{aligned}$$

In our example of Figure 2, contracts (A_1, G_1) and (A_2, G_2) are partial contracts for component C, because they relate to its sub-components, and as such, they each observe a subset of C's state variables, namely o_1, and o_2, respectively. As our goal is contract decomposition for realizability, we are particularly interested to discover conditions under which contracts (A_1, G_1) and (A_2, G_2) can be equivalently represented by a global contract (A, G) where $A = A_1 \wedge A_2$ and $G = G_1 \wedge G_2$, meaning that $G_I = G_{I1} \wedge G_{I2}$ and $G_T = G_{T1} \wedge G_{T2}$. We have identified two challenges in addressing this goal: guarantee and assumption interference.

Subcontracts sharing state variables may cause guarantee interference. In our example, imagine that o_1 and o_2 are the same variable. Then finding an implementation for (A_1, G_1) and an implementation for (A_2, G_2) does not mean that there exists one for (A, G), since these implementations may be based on conflicting valuations for the common output. Because of guarantee interference, checking realizability of a global contract by checking realizability of its partial contracts may be too optimistic, in the sense that it may return false positives.

In previous work, we proposed a decomposition approach based on connected components to avoid common state variables [33]. This approach was applied to requirements expressed as sets of guarantees, i.e., they were not taking AG contracts into account. In terms of assumptions, we observe that common input and state variables may create assumption interference. In our example, let i be an input variable, and let $A_1 = (i > 0)$ and $A_2 = (i < 3)$. When $i = 5$, A_1 holds but A_2 does not. Realizability of (A_1, G_1) will still require an implementation that conforms to G_1 for $i = 5$, but realizability of (A, G) will not, because A_2 is violated. Because of assumption interference, checking realizability of a global contract by checking realizability of its partial contracts may be too pessimistic, in the sense that it may return false negatives.

This work builds upon our previous work [33] by examining how to decompose global contracts in the presence of assumptions. Let $AG_1^n \overset{\text{def}}{=} \{(A_i, G_i) : i = 1, \ldots, n\}$ represent a set of n partial AG contracts for a system over $STATES$ and *inputs*. We start by introducing a notion of realizability $\mathsf{PRealizable}_{AG_1^n}$ for AG_1^n. Theorem 1 then shows that in the context of contracts that do not share state, $\mathsf{PRealizable}_{AG_1^n}$ is equivalent to ensuring that every subcontract (A_i, G_i) is realizable. Finally, Theorem 2 uses these results to decompose the realizability of a contract (A, G) into realizability of subcontracts. Due to space limitations, the proofs of Theorems 1 and 2 are provided in [22].

We first extend the notions of viability and realizability presented in Section 3 for a set of partial contracts AG_1^n as follows.

Definition 3 (Partial-contract viability). *A set of viable system responses with respect to a set of partial contracts is defined coinductively, as the greatest fixed point of the following equation:*

$$\begin{aligned}
&\mathsf{PViable}_{AG_1^n}(s) = \forall a : inputs. \\
&(\vee_{i=1}^n A_i(s@SV_i, a)) \Rightarrow \\
&\exists s'.[(\wedge_{i=1}^n (A_i(s@SV_i, a) \Rightarrow G_{Ti}(s@SV_i, a, s'@SV_i))) \wedge \mathsf{PViable}_{AG_1^n}(s')]
\end{aligned}$$

Intuitively, each partial contract (A_i, G_i) imposes constraints on how a subset of the state variables must evolve. When at least one assumption $A_i(s@SV_i, a)$ holds, then constraints are imposed on the state transition. Note that when a system consists of a single contract (A_1, G_1) where $SV_1 = SV$, Definition 3 becomes equivalent to Definition 1.

We define realizability of a set of partial contracts AG_1^n as:

Definition 4 (Partial-contract Realizability).

$$\mathsf{PRealizable}_{AG_1^n} \overset{\text{def}}{=} \exists s.\ \wedge_{i=1}^n G_{Ii}(s@SV_i) \wedge \mathsf{PViable}_{AG_1^n}(s)$$

Following our observations of [33], we call a *non-interfering contract set over* SV, a set of partial contracts (A_i, G_i) over SV_i iff the sets SV_i partition SV. In other words, the partial contracts have no common state variables and together they cover SV. For non-interfering contract sets, realizability can be decomposed, following Theorem 1 below.

Theorem 1. *Let AG_1^n be a non-interfering contract set. Then:*

$$(\wedge_{i=1}^n \mathsf{Realizable}_{A_i G_i}) \Leftrightarrow \mathsf{PRealizable}_{AG_1^n}$$

In other words, for non-interfering contract sets, partial-contract realizability is equivalently decomposed into realizability of the individual partial contracts. It remains to discover conditions under which partial-contract realizability coincides with global contract realizability. By comparing the definitions of Viable_{AG} and $\mathsf{PViable}_{AG_1^n}$, a main difference that stands out is $\wedge_{i=1}^n A_i$ vs $\vee_{i=1}^n A_i$. So we examine the case where $\wedge_{i=1}^n A_i \equiv \vee_{i=1}^n A_i$, which is equivalent to

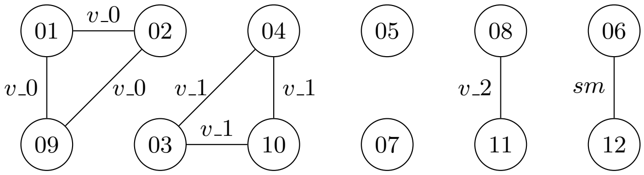

Fig. 3: `liquid_mixer` connected components. We use the last two digits of req. names, and abbreviate `valve_x` to `v_x`, `stirring_motor` to `sm`.

$A_1 \equiv A_2 \equiv \ldots \equiv A_n$ (from Boolean algebra). Additionally, since $A = \wedge_{i=1}^{n} A_i$, it follows that $A \equiv A_1 \equiv A_2 \equiv \ldots \equiv A_n$. Since the partial contracts (A_i, G_i) are non-interfering, they are defined over state variable sets SV_i that partition SV. For all the assumptions A_i to be equivalent under all circumstances, these assumptions, including assumption A, must be independent of state. An assumption $A(s,a) : \mathit{state} \times \mathit{inputs} \rightarrow \mathit{bool}$ is considered *independent of state*, iff $\forall s_1, s_2 \in \mathit{state}, \forall a \in \mathit{inputs}.\, A(s_1,a) = A(s_2,a)$. We abbreviate $A(*,a)$ by $A(a)$. The following theorem captures these observations.

Theorem 2. *Let (A,G) be an AG contract over state variable set SV, with A independent of state, and let (A,G_i) with $i = 1 \ldots n$, be a non-interfering contract set over state variable sets SV_i, where $G = \wedge_{i=1}^{n} G_i$ (i.e., $G_I = \wedge_{i=1}^{n} G_{Ii}$ and $G_T = \wedge_{i=1}^{n} G_{Ti}$). Then* $\mathsf{Realizable}_{AG} \equiv (\wedge_{i=1}^{n} \mathsf{Realizable}_{AG_i})$.

5 Connected Components

In this section, we present one approach to automatically decomposing an assume-guarantee contract (A,G) into an equivalent set of partial contracts (A,G_i) per the conditions of Theorem 2. As discussed, Theorem 2 requires the assumption A to be independent of state. Consequently, to obtain non-interfering contracts, we only need to consider guarantees.

More specifically, we decompose an assume-guarantee contract that fits the conditions of Theorem 2 by splitting the guarantees G based on the notion of connected components [25,54] for undirected graphs. As seen in Figure 3, the connected components of a graph essentially represent separated pieces of the graph. Two vertices belong to the same connected component if and only if there exists some path between them.

As discussed in Section 3, AG contracts (A,G) are typically expressed as sets of assumption and guarantee predicates, with A and G corresponding to their respective conjunctions. More formally, let R_{GI} and R_{GT} be sets of predicates that define G, meaning that $G_I = \bigwedge_{R_i \in R_{GI}} R_i$, and $G_T = \bigwedge_{R_i \in R_{GT}} R_i$. We use the set $R = R_{GI} \cup R_{GT}$ to represent all guarantee predicates involved in G, without differentiating between initial state and transitional guarantees. We refer to elements of set R as requirements.

A *requirements graph* for R is an undirected graph (V,E), which is built as follows. Each vertex in V corresponds to a requirement in R. If the state variables referenced by two requirements overlap, their corresponding vertices in the graph are connected by an edge in E. By computing connected components in R, we are able to decompose the original specification into partial contracts.

Figure 3 illustrates the connected components for the liquid mixer system. The components partition the requirement state variables, namely *valve_0*, *valve_1*, *valve_2*, *timer_60sec_start* (referenced only by [**LM-005**]), *timer_120sec_start* (referenced only by [**LM-007**]), and *stirring_motor*. Let us now formally present our connected component approach.

Let R be the set of requirements in an AG contract (A, G) over state variables in SV, as described previously. For $R_i \in R$, we use SV_{Ri} to denote the state variables that are referenced by requirement R_i. For initial state guarantees, this means that $\forall s_1, s_2 \in STATES.\, (s_1@SV_{Ri} = s_2@SV_{Ri}) \Rightarrow R_i(s_1) = R_i(s_2)$. For each R_i we can therefore define the predicate $R_i@SV_{Ri}$ that behaves as R_i, but has lower dimensionality when $SV_{Ri} \subset SV$:

$$\begin{aligned} &\forall s_i \in STATES@SV_{Ri}. \\ &R_i@SV_{Ri}(s_i) \overset{\text{def}}{=} \exists s \in STATES.\, ((s@SV_{Ri} = s_i) \wedge R_i(s)). \end{aligned}$$

The above notations and definitions naturally extend to transitional guarantees.

Definition 5 (Requirements Graph). *A requirements graph RG is an undirected graph whose vertices are requirements $R_i \in R$ and with an edge (R_i, R_j) between every pair of requirements R_i and R_j that share at least one state variable; that is, $SV_{Ri} \cap SV_{Rj} \neq \varnothing$. Notice that we do not consider input variables for the construction of this graph.*

Definition 6 (State-Connected Component (SCC)). *Let R be a set of requirements, and RG be its corresponding requirements graph. A State-Connected Component is a tuple $C = (R_c, SV_c)$ where*

- *$R_c \subseteq R$ is the set of requirements in a connected component of RG; that is, there is a connected path of edges between each pair of requirements in R_c and no superset of R_c also has this property.*
- *SV_c is the set of state variables that are mentioned in any requirement in R_c; that is, $SV_c = \bigcup\limits_{R_i \in R_c} SV_{Ri}$.*

State-connected components can be computed with a connected component algorithm. For example, our framework implements Tarjan's classic connected components algorithm [25] with an $O(|V| + |E|)$ complexity. State-connected components can then be used to create AG contracts as follows.

Definition 7 (Connected AG Contract). *Let (A, G) be an AG contract over state variables SV, where A is independent of state. Let $R = R_{GI} \cup R_{GT}$ be the contract's set of requirements, with its corresponding requirements graph RG. Each state-connected component $C = (R_c, SV_c)$ in RG defines a Connected AG Contract (A_c, G_c), as follows:*

- *$\forall s \in STATES@SV_c, \forall a \in inputs.\, A_c(s, a) = A(a)$.*
- *$G_{Ic} = \bigwedge_{Ri \in (R_c \cap R_{GI})} Ri@SV_c$ is G_c's initial-state guarantee predicate.*

Table 2: Case studies statistics. The "Monolithic" and "Total SCC" columns record the monolithic and compositional (SCC) analysis time in seconds. SCC times are denoted by "N/A" if decomposition was not successful.

Project	**Benchmark**	**#Reqs**	**#SCCs**	**Realizable?**		**Monolithic**		**Total SCC**	
				JSyn	JSyn-vg	JSyn	JSyn-vg	JSyn	JSyn-vg
Example	`liquid_mixer`	12	6	✘	✘	0.50	10.27	2.47	5.85
GPCA	`Infusion_Manager`	26	1	✘	✘	0.40	10.76	N/A	N/A
QFCS	`FCC`	9	7	✔	?	53.31	T/O	4.12	T/O + 6.38
QFCS	`FCC (inlined)`	79	38	✔	✔	1.11	T/O	13.10	16.41
QFCS	`OSAS`	10	2	✘	✘	1.82	T/O	2.96	T/O + 1.12
QFCS	`OSAS (inlined)`	190	21	✘	✘	1.32	T/O	10.83	640.89
LMCPS	`AP`	13	3	?	?	0.40	T/O	1.42	T/O + 8.68
LMCPS	`FSM`	13	3	✘	✘	0.42	1524.82	3.51	6.74
LMCPS	`EB`	5	2	?	?	1.41	1.01	1.62	1.39
LMCPS	`NN`	4	1	?	?	55.80	269.87	N/A	N/A
LMCPS	`REG`	10	5	?	✔	286.14	99.52	422.52	5.72
LMCPS	`TSM`	6	1	✘	✔	3.33	242.67	N/A	N/A

✔: realizable ✘: unrealizable ?: unknown T/O: timeout (12 hours)

- $G_{Tc} = \bigwedge_{Ri \in (R_c \cap R_{GT})} Ri@SV_c$ *is* G_c*'s transitional guarantee predicate.*

By construction, the state variables over which individual connected AG contracts are defined partition the set of system state variables. Hence, connected AG contracts are partial contracts that can be used to decompose realizability checking, according to Theorem 2.

Of the 6 connected components in `liquid_mixer` (Figure 3), only 1 was found to be unrealizable, consisting of requirements [**LM-001**], [**LM-002**] and [**LM-009**]. Thus, we were able to localize and identify the conflict between [**LM-001**] and [**LM-009**], a result consistent with the findings by Lúcio et al. [35].

6 Case Studies

We applied our compositional approach on three multi-component, industrial-level projects[7]. The different components/benchmarks and number of requirements of each project are shown in Table 2. Next, we describe each project.

Generic Infusion Pump (GPCA): The Generic Infusion Pump Research Project [2] is a joint effort by the United States Food and Drug Administration (USFDA), Hutchison China MediTech (Chi-Med), CIMIT [1] and ten universities to identify best software engineering practices in the development of medical devices. The Generic Patient Controlled Analgesic (GPCA) infusion pump has been previously developed and formally analyzed using the AGREE framework [46,45]. This study used the `Infusion_Manager` subcomponent, previously shown unrealizable by Gacek et al. [20] through the use of the JSyn algorithm. The subcomponent contains 12 requirements.

We first translated the original specification into FRETish, and created 26 (as opposed to 12) requirements. This difference is mainly due to our choice of using FRET's inherent support for modes through the `scope` field. The original

[7] Datasets are available upon request. Please email the authors.

contract used a single variable *Current_System_Mode* with 8 different values to model the 8 component modes. In FRET, it is more natural to use 8 different mode variables instead, and avoid mixing properties of different modes in a single requirement. As an example, consider the following requirement:

$$\begin{aligned}\mathbf{G1} \overset{\text{def}}{=}\ & (\mathit{Current\_System\_Mode'} \geq 0)\ \wedge\ (\mathit{Current\_System\_Mode'} \leq 8)\ \wedge \\ & (\mathit{Current\_System\_Mode'} = 0 \Rightarrow\ \mathit{Commanded\_Flow\_Rate'} = 0)\ \wedge \\ & (\mathit{Current\_System\_Mode'} = 1 \Rightarrow\ \mathit{Commanded\_Flow\_Rate'} = 0)\end{aligned}$$

which gets decomposed into three requirements $\mathbf{G1_1}$, $\mathbf{G1_2}$, and $\mathbf{G1_3}$, corresponding to the first, second and third line in the original contract[8]. Requirement $\mathbf{G1_1}$ ensures that the system is in at least one of the 8 modes at any time. Requirements $\mathbf{G1_2}$ and $\mathbf{G1_3}$ define component behavior when in mode 0 and 1, respectively. We added requirements to ensure mutual exclusion between modes, something that was not needed with a single mode variable. We used KIND 2 [10] to show equivalence between our requirements and the original specification.

NASA's Quad-Redundant Flight Control System (QFCS): QFCS is a component in NASA's Transport Class Model aircraft simulation [26]. It is composed of four cross-checking flight control computers (FCC), and contains specifications regarding the control laws and sensing properties of the aircraft. It has been used in the past for the purposes of requirements analysis within the Assume-Guarantee Reasoning Framework (AGREE) [12], both in terms of compositional verification [3] as well as realizability checking and synthesis through JKIND [20,29]. Compared to the latter work, our compositional approach yields new information, that would otherwise be impossible to derive using the monolithic algorithms in JKIND. We present new results on the `FCC` (9 requirements) and Output Signal Analysis and Selection (`OSAS`, 10 requirements) contracts. For this case study, we did not write the requirements in FRET but directly used the provided Lustre specifications to perform decomposition and analysis.

Lockheed Martin CPS Challenge Problems (LMCPS): LMCPS is a set of industrial Simulink model benchmarks and natural language requirements [15,16]. They consist of a set of problems inspired by flight control and vehicle management systems, which are representative of flight critical systems. LMCPS was created by Lockheed Martin Aeronautics to evaluate and improve the state-of-the-art in formal method toolsets. There are two recent research works that study the formalization of the LMCPS requirements and their analysis against the Simulink models. Nejati et al. [47] perform model testing and checking, while Mavridou et al. [42] perform requirement specification and model checking. However, none of these works check consistency or realizability. To perform realizability analysis, we used the FRETISH form of the requirements [41].

We present results for several LMCPS challenges[9]: 1) `6DoF with DeHavilland Beaver Autopilot (AP)`: a simulation of the DeHavilland Beaver airplane with autopilot (13 requirements); 2) `Finite State Machine (FSM)`: an abstraction

[8] We discuss requirements in the original contract notation to make it easy to relate to Gacek et al. [20]

[9] For brevity, we omit challenges for which our work did not yield new information. Additional analysis results can be found in a supplementary technical report [33].

of an advanced autopilot system (13 requirements); 3) `Effector Blender (EB)`: a control allocation method that calculates the optimal effector configuration (5 requirements); 4) `Feedforward Cascade Connectivity Neural Network (NN)`: a predictor neural network (4 requirements); 5) `Control Loop Regulators (REG)`: a regulator's inner loop architecture (10 requirements); 6) `Triplex Signal Monitor (TSM)`: a redundancy management system (6 requirements).

6.1 Analysis Outcomes & Lessons Learned

In Table 2 we summarize analysis outcomes and performance times of the `Infusion Manager` example, and the `GPCA`, `QFCS`, and `LMCPS` projects. For each project, we computed the number of SCCs and applied monolithic and compositional realizability analysis by using both the JSyn and JSyn-vg algorithms. The experiments were run on an Ubuntu VM, 4.5GB RAM, i5-8365U, 4 cores@1.60 GHz. Next, we discuss in detail the analysis results and provide insights.

No decomposition: Our decomposition method returned a single SCC for the `LMCPS NN`, `LMCPS TSM`, and the `Infusion_Manager` specifications. One reason that contributed to the unsuccessful decomposition was that requirements were connected through mode variables. In the future, we plan to study whether large mode-related specifications can be analyzed modularly, by studying the mode-transition logic separately from the intra-mode requirements.

Challenges in realizability analysis: The nested quantifiers in Def. 2 can be particularly challenging for state-of-the art solvers. Furthermore, infinite-state problems are undecidable in general, and the corresponding solvers are not complete. Additionally, many of the `LMCPS` specifications contain non-linear expressions that are not entirely supported by SMT solvers. For instance, the `EB` challenge returned "unknown" due to non-linearities for both JSyn and JSyn-vg and we were not able to get a result even after decomposing the specification into 2 SCCs. Similarly, in the `AP` challenge, the monolithic JSyn approach returned "unknown" due to non-linearities, while JSyn-vg timed out. Even though, the monolithic approach did not yield results for `AP`, by decomposing the specification we were able to get more meaningful results as explained next.

Successful decomposition: Several of our case studies demonstrated how decomposition can effectively reduce problem complexity, surpass some of the aforementioned challenges, and lead to significant performance benefits.

– `LMCPS AP`: Our SCC algorithm decomposed the specification into 3 SCCs. It is worth noting that while we were able to verify realizability of the two SCCs in less than 8.7s, JSyn-vg was not able to solve the last SCC and timed-out due to non-linear expressions. Despite not getting a conclusive answer, decomposition helped us identify and successfully check linear fragments of the specification. `AP` showcases how partial results can be retrieved via decomposition, while identifying fragments for which the solvers fail due to problem complexity.

– `LMCPS FSM`: Monolithic realizability analysis returned unrealizable with both JSyn and JSyn-vg. Decomposition returned 3 SCCs. One was realizable while the other two were unrealizable. This helped us localize the causes of unrealizability within the corresponding SCCs. Additionally, decomposing our original specification allowed us to reduce the total analysis time from 1524.82s to less than 7s with JSyn-vg.

– `LMCPS REG`: This challenge is highly decomposable: for 10 requirements our decomposition approach returned 5 SCCs. `REG` was proven realizable by JSyn-vg through monolithic checking in 99.52s, while compositional checking needed a total time of only 5.72s. On the contrary, JSyn timed out during both the monolithic check and when checking each SCC independently.

– `QFCS`: The `FCC` contract contains only 9 requirements, yet JSyn required 53.31s to declare it as realizable while JSyn-vg could not solve the same problem, even for a timeout value of 12 hours. Using our decomposition method, we partitioned the contract into 7 SCCs, 6 of which contain a single requirement, with the seventh containing 3. We then ran a realizability check over each component individually. The decomposition step resulted in a dramatic performance improvement: JSyn required only 4.12s to solve the entire problem, and JSyn-vg solved the six singletons in 6.38s, and timed out for the seventh SCC (`FCC-7`).

Requirement granularity: To better identify why `FCC-7` was so hard to solve, we examined the requirement definitions: the majority of the requirements in the project (i.e., both `FCC` and `OSAS`) are big conjunctions, where each conjunct corresponds to the application of user-defined, reusable predicate templates, over a disjoint set of state variables (`FCC` contains 6 templates in total, `OSAS` contains 9). For example, requirement **GUARANTEE6** in `FCC` is defined as [10]:

$$\begin{aligned}\textbf{GUARANTEE6} \stackrel{\text{def}}{=} &\, \mathit{range}(\mathit{valid\_acts.TL}, \mathit{acts\_out.TL}', 0.0, 50.0) \land \ldots \\ &\ldots \land \mathit{range}(\mathit{valid\_acts.STEER}, \mathit{acts\_out.STEER}', 0.0, 50.0),\end{aligned}$$

where each conjunct is the application of the template *range* over pairs of variables from *valid_acts* and *acts_out*. While using big conjunctions was, as commented by the project authors, “out of convenience”, it unsurprisingly resulted in performance overhead, as the monolithic algorithms needed to consider all of the conjuncts at the same time (the solver query has 1481 variables), even though each conjunct can be considered as a separate requirement, and therefore be a candidate for decomposition.

As such, we split the initial 9 requirements of `FCC` into subrequirements: for each requirement, and for each application of a template, we derived a subrequirement. The resulting `FCC` contract, i.e., `FCC (inlined)` in Table 2, consists of 79 requirements. The monolithic JSyn run improved significantly (1.11s), but JSyn-vg still timed out. Decomposing the new contract resulted in 38 SCCs, which we individually checked for realizability (30 variables per SCC query, on average). While the compositional run for JSyn was a bit slower (13.10s total, ∼0.34s per SCC), JSyn-vg was finally able to determine the contract as realizable, requiring in total only 16.41s (∼0.43s per SCC).

Similarly, we decomposed the original `OSAS` contract into 2 SCCs, one of which was a singleton. To our surprise, the singleton was unrealizable, and further inspection revealed the cause: the corresponding requirement was declared as a guarantee, yet contained no state variables. Such a guarantee would always be unrealizable as the system cannot control inputs. As with `FCC`, templates

[10] We have shortened the element names in the requirement to reduce the overall size.

are heavily used in OSAS. Deriving new requirements out of templates resulted in a new contract, i.e., OSAS (inlined), with 190 requirements, and 21 SCCs (the monolithic query contained 3035 variables, versus the 151 per SCC query). Through the decomposition, we were able to determine the contract as unrealizable by using both JSyn and JSyn-vg.

To sum up, our compositional approach helped us understand that requirement templates can negatively impact analysis performance and decomposition (e.g., 21 vs. 2 SCCs in OSAS). The QFCS case study stands out from the rest since we did not enter the requirements in FRET but instead directly used the provided Lustre specifications. From our experience, it is not common in practice to write such long requirements (as the ones provided in Lustre) in FRET; usually FRETish sentences are relatively short. For example, take the GPCA case study, for which we created 26 FRETish requirements as opposed to the initial 12 requirements. As shown via the inlined requirements, shorter requirements may enable finer decomposition and thus, return meaningful results.

Algorithm trade-offs: Although the JSyn algorithm is not sound for unrealizability results, it returned a result in several cases (e.g., FCC, OSAS) for which JSyn-vg needed more time or even timed out. We thus realised that the two algorithms can be combined together with our compositional approach to optimize performance. To this end, JSyn can be used for returning fast sound realizable results, while JSyn-vg can be effectively used in the compositional context to determine sound unrealizability without timing out (e.g., OSAS (inlined)).

To conclude, our compositional approach helped us gain significant insights into the challenges of realizability analysis and possible ways to overcome these. For example, we understood that the granularity level of specifications plays an important role since shorter formulas usually enable finer decomposition. Additionally, we realized that decomposition can be particularly helpful for the analysis of specifications that are challenging for state-of-the-art solvers, such as specifications with nested quantifiers and non-linear expressions. In general, our compositional approach helps us overcome challenges in realizability checking since it reduces problem complexity and achieves significant performance gains.

6.2 Comparison of decomposition with Finkbeiner et al. [17]

Finkbeiner et al. recently proposed, in an independent effort to ours, two approaches towards specification decomposition for reactive synthesis. Most relevant to our work is a decomposition algorithm for LTL specifications where, given an LTL formula, each conjunct of its CNF equivalent occurs in exactly one subspecification. Similarly to our work, this is achieved by partitioning the original specification based on dependencies between system variables.

One notable difference between the two approaches is the level at which they are performed. We perform decomposition at the level of FRET requirements, rather than their corresponding LTL formulas. One of the reasons for this choice is that FRET interacts with users at the level of requirements, which promotes diagnosis and repair. Moreover, as observed in our experiments, when expressing requirements in the FRET environment, users tend to write small requirements as opposed to conjoining multiple ones in a single FRETish sentence. Nevertheless, the FRET formula generation algorithms [24] may create large formulas. It would be interesting to explore if we can obtain additional gains by performing decomposition at the level of formulas using Finkbeiner et al.'s algorithms. In fact, we could try to apply Finkbeiner et al.'s algorithms on the connected components identified by our algorithm for further decomposition.

Table 3: Comparison with Finkbeiner et al. [17]. Iden stands for identical.

Benchmark	#SCCs	#Specs [17]	Iden?
`Cockpitboard`	8	8	✔
`Gamelogic`	4	4	✔
`LedMatrix`	3	3	✔
`Radarboard`	11	11	✔
`zoo10`	1	2	✘
`generalized_buffer_2`	2	2	✔
`generalized_buffer_3`	2	2	✔
`shift_8`	8	8	✔
`shift_10`	10	10	✔
`shift_12`	12	12	✔

To compare the approaches we used ten benchmarks from the SYNTCOMP 2020 competition [27] [11], for which the decomposition proposed by Finkbeiner et al. yielded exemplary results [17]. Since neither JSyn nor JSyn-vg support liveness properties, a direct comparison regarding realizability results is not possible. As such, we focused on comparing the quality of decomposition. As shown in Table 3, our decomposition procedure yielded identical results with Finkbeiner et al. for all but the `zoo10` benchmark [12]. We attribute the discrepancy to an optimization in the algorithm by Finkbeiner et al., which yields two subspecifications whose assumptions are not equivalent to each other. It is currently unclear to us whether this decomposition is compatible with our formal framework, and we plan on revisiting this in future work.

7 Related Work

Realizability checking of specifications is a well-established field of research in formal methods, and is strongly tied to the area of *reactive synthesis*. Pnueli and Rosner were the first to show that the complexity of the problem is double-exponential (2-EXPTIME) for propositional specification [50], while further advancements in the General Reactivity of Rank 1 (GR(1)) fragment of LTL showed that a polynomial time algorithm exists [49,6]. In the context of propositional logic, various tools have been proposed towards the realizability analysis of reactive systems, some of which follow a user-guided approach [53], while others serve as general requirements analysis and debugging frameworks [5,37,14]. Our work addresses the same problem, but in the context of potentially infinite-state specifications, where scalability is a major concern.

Specification decomposition is also a research problem of relevance to formal methods and more specifically formal verification, with previous work on procedures that factorize the specification into smaller problems [7,21,48]. The same also applies in the context of synthesis where compositional techniques have been proposed, taking advantage of Binary Decision Diagrams, And-Inverter Graphs and Extended Finite-State Machines to restructure the original problem into

[11] SYNTCOMP 2020 benchmarks: `https://github.com/SYNTCOMP/benchmarks`

[12] The authors provided us with their resulting subspecifications.

factored formulas [28,56,8,43]. In comparison to this work, our proposed algorithm to compute SCCs is orthogonal and does not rely on solvers to perform the decomposition. Furthermore, our approach is not affected by the order in which specification elements are processed and as such does not require the application of sophisticated ordering heuristics [56,44].

The diagnosis of unrealizable specifications has also been extensively explored, primarily in the context of computing minimal sets of conflicting specification elements, commonly referred to as *unrealizable cores*. In this paper, we rely on the model-based diagnosis technique proposed by Kónighofer et al. [32] to compute all minimal unrealizable cores. The technique is modular with respect to the way that a single minimal conflict is computed, allowing for different implementations to be considered. FRET supports both delta debugging [57] and a linear algorithm proposed by Cimatti et al. [11] to compute minimal conflicts. By default, we use delta debugging as it has been shown to perform better, on average. Recently, Maoz et al. proposed QuickCore for computing unrealizable cores, with optimizations based on specific properties of GR(1) specifications [39]. Within the context of GR(1) specifications, QuickCore was shown to perform better than delta debugging. In the future, we intend to evaluate the applicability of QuickCore within FRET.

Akin to realizability checking, prior work exists in other aspects of requirement analysis, such as *rt-inconsistency* [51,52,34], *well-separation* [36,31] and *inherent vacuity* [38,19]. Note that in the case of rt-inconsistency, an unrealizable contract is also rt-inconsistent but not necessarily vice-versa. It would be interesting to explore whether SCC computation could also benefit these types of analysis.

8 Conclusion

We presented a new realizability analysis framework, developed as an extension of FRET. Our partial contracts approach can be applied as a preprocessing step on a variety of realizability analysis tools, by taking advantage of a specification's modularity over disjoint subsets of requirements. We evaluated our approach with state-of-the-art infinite-state realizability algorithms on several industrial case studies. We obtained encouraging results in reducing problem complexity and significantly improving analysis performance.

We focused on conditions under which the realizability of a global contract can equivalently be decomposed into the realizability of its partial contracts. In the future, we plan to study if it is possible to relax the requirement of equivalent partial contract assumptions in Theorem 2. It is worthwhile noting that, for non-interfering contract sets, partial contract realizability always implies global contract realizability, but not vice versa. So to establish a positive realizability result, we can relax the conditions for decomposition, at the expense of losing the capability to call unrealizability. We plan to explore such trade-offs in practice.

References

1. Consortia for improving medicine within innovation and technology, https://cimit.org/home
2. Generic infusion pump research project, https://rtg.cis.upenn.edu/gip/
3. Backes, J., Cofer, D., Miller, S., Whalen, M.W.: Requirements analysis of a quad-redundant flight control system. In: Havelund, K., Holzmann, G., Joshi, R. (eds.) NASA Formal Methods. Lecture Notes in Computer Science, vol. 9058, pp. 82–96. Springer International Publishing (2015). https://doi.org/10.1007/978-3-319-17524-9_7, http://dx.doi.org/10.1007/978-3-319-17524-9_7
4. Benveniste, A., Caillaud, B., Nickovic, D., Passerone, R., Raclet, J.B., Reinkemeier, P., Sangiovanni-Vincentelli, A., Damm, W., Henzinger, T.A., Larsen, K.G., et al.: Contracts for system design (2018)
5. Bloem, R., Cimatti, A., Greimel, K., Hofferek, G., Könighofer, R., Roveri, M., Schuppan, V., Seeber, R.: Ratsy–a new requirements analysis tool with synthesis. In: International Conference on Computer Aided Verification. pp. 425–429. Springer (2010)
6. Bloem, R., Jobstmann, B., Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive (1) designs. Journal of Computer and System Sciences **78**(3), 911–938 (2012)
7. Burch, J.R., Clarke, E.M., Long, D.E.: Representing circuits more efficiently in symbolic model checking. In: Proceedings of the 28th ACM/IEEE Design Automation Conference. pp. 403–407. Association for Computing Machinery, New York, NY, USA (1991). https://doi.org/10.1145/127601.127702, https://doi.org/10.1145/127601.127702
8. Chakraborty, S., Fried, D., Tabajara, L.M., Vardi, M.Y.: Functional synthesis via input-output separation. In: 2018 Formal Methods in Computer Aided Design (FMCAD). pp. 1–9. IEEE (2018)
9. Champion, A., Gurfinkel, A., Kahsai, T., Tinelli, C.: CoCoSpec: A mode-aware contract language for reactive systems. In: International Conference on Software Engineering and Formal Methods. pp. 347–366. Springer (2016)
10. Champion, A., Mebsout, A., Sticksel, C., Tinelli, C.: The Kind 2 model checker. In: International Conference on Computer Aided Verification. pp. 510–517. Springer (2016)
11. Cimatti, A., Roveri, M., Schuppan, V., Tchaltsev, A.: Diagnostic information for realizability. In: International Workshop on Verification, Model Checking, and Abstract Interpretation. pp. 52–67. Springer (2008)
12. Cofer, D., Gacek, A., Miller, S., Whalen, M.W., LaValley, B., Sha, L.: Compositional verification of architectural models. In: NASA Formal Methods Symposium. pp. 126–140. Springer (2012)
13. Damm, W., Hungar, H., Josko, B., Peikenkamp, T., Stierand, I.: Using contract-based component specifications for virtual integration testing and architecture design. In: 2011 Design, Automation & Test in Europe. pp. 1–6. IEEE (2011)
14. Ehlers, R., Raman, V.: Slugs: Extensible GR (1) synthesis. In: International Conference on Computer Aided Verification. pp. 333–339. Springer (2016)
15. Elliott, C.: On example models and challenges ahead for the evaluation of complex cyber-physical systems with state of the art formal methods V&V, Lockheed Martin Skunk Works. In: Laboratory, A.F.R. (ed.) Safe & Secure Systems and Software Symposium (S5) (2015)
16. Elliott, C.: An example set of cyber-physical V&V challenges for S5, Lockheed Martin Skunk Works. In: Laboratory, A.F.R. (ed.) Safe & Secure Systems and Software Symposium (S5) (2016)

17. Finkbeiner, B., Geier, G., Passing, N.: Specification decomposition for reactive synthesis. In: NASA Formal Methods Symposium. pp. 113–130. Springer (2021)
18. Firman, E., Maoz, S., Ringert, J.O.: Performance heuristics for GR (1) synthesis and related algorithms. Acta informatica **57**(1), 37–79 (2020)
19. Fisman, D., Kupferman, O., Sheinvald-Faragy, S., Vardi, M.Y.: A framework for inherent vacuity. In: Haifa Verification Conference. pp. 7–22. Springer (2008)
20. Gacek, A., Katis, A., Whalen, M.W., Backes, J., Cofer, D.: Towards Realizability Checking of Contracts Using Theories. In: NFM. LNCS, vol. 9058, pp. 173–187. Springer (2015)
21. Geist, D., Beer, I.: Efficient model checking by automated ordering of transition relation partitions. In: International Conference on Computer Aided Verification. pp. 299–310. Springer (1994)
22. Giannakopoulou, D., Katis, A., Mavridou, A., Pressburger: Compositional Realizability Checking within FRET. NASA Technical Memorandum (March 2021), https://ti.arc.nasa.gov/publications/20210013008/download/, 32 pages
23. Giannakopoulou, D., Pressburger, T., Mavridou, A., Rhein, J., Schumann, J., Shi, N.: Formal requirements elicitation with FRET. In: Joint Proceedings of REFSQ-2020 Workshops, Doctoral Symposium, Live Studies Track, and Poster Track co-located with the 26th International Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2020), Pisa, Italy, March 24, 2020. CEUR Workshop Proceedings, vol. 2584. CEUR-WS.org (2020), http://ceur-ws.org/Vol-2584/PT-paper4.pdf
24. Giannakopoulou, D., Pressburger, T., Mavridou, A., Schumann, J.: Automated formalization of structured natural language requirements. Information and Software Technology **137**, 106590 (2021). https://doi.org/https://doi.org/10.1016/j.infsof.2021.106590, https://www.sciencedirect.com/science/article/pii/S0950584921000707
25. Hopcroft, J., Tarjan, R.: Algorithm 447: efficient algorithms for graph manipulation. Communications of the ACM **16**(6), 372–378 (1973)
26. Hueschen, R.M.: Development of the transport class model (tcm) aircraft simulation from a sub-scale generic transport model (gtm) simulation (2011)
27. Jacobs, S., Bloem, R., Brenguier, R., Ehlers, R., Hell, T., Könighofer, R., Pérez, G.A., Raskin, J.F., Ryzhyk, L., Sankur, O., et al.: The first reactive synthesis competition (syntcomp 2014). International journal on software tools for technology transfer **19**(3), 367–390 (2017)
28. John, A.K., Shah, S., Chakraborty, S., Trivedi, A., Akshay, S.: Skolem functions for factored formulas. In: 2015 Formal Methods in Computer-Aided Design (FMCAD). pp. 73–80. IEEE (2015)
29. Katis, A., Fedyukovich, G., Guo, H., Gacek, A., Backes, J., Gurfinkel, A., Whalen, M.W.: Validity-guided synthesis of reactive systems from assume-guarantee contracts. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 176–193. Springer (2018)
30. Katis, A., Gacek, A., Whalen, M.W.: Towards synthesis from assume-guarantee contracts involving infinite theories: a preliminary report. In: 4th Intl. Conf. on Formal Methods in Software Engineering (FormaliSE). pp. 36–41. IEEE (2016)
31. Klein, U., Pnueli, A.: Revisiting synthesis of GR (1) specifications. In: Haifa Verification Conference. pp. 161–181. Springer (2010)
32. Könighofer, R., Hofferek, G., Bloem, R.: Debugging formal specifications: a practical approach using model-based diagnosis and counterstrategies. International Journal on Software Tools for Technology Transfer **15**(5-6), 563–583 (2013)

33. Kooi, D., Mavridou, A.: Integrating Realizability Checking in FRET. NASA Technical Memorandum (June 2019), `https://ntrs.nasa.gov/api/citations/20190033980/downloads/20190033980.pdf`, 28 pages
34. Langenfeld, V., Dietsch, D., Westphal, B., Hoenicke, J., Post, A.: Scalable analysis of real-time requirements. In: 2019 IEEE 27th International Requirements Engineering Conference (RE). pp. 234–244 (2019). https://doi.org/10.1109/RE.2019.00033
35. Lúcio, L., Rahman, S., Cheng, C.H., Mavin, A.: Just formal enough? automated analysis of EARS requirements. In: NASA Formal Methods Symposium. pp. 427–434. Springer (2017)
36. Maoz, S., Ringert, J.O.: On well-separation of GR (1) specifications. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 362–372 (2016)
37. Maoz, S., Ringert, J.O.: Spectra: a specification language for reactive systems. arXiv preprint arXiv:1904.06668 (2019)
38. Maoz, S., Shalom, R.: Inherent vacuity for GR (1) specifications. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 99–110 (2020)
39. Maoz, S., Shalom, R.: Unrealizable cores for reactive systems specifications. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). pp. 25–36. IEEE (2021)
40. Mavridou, A., Bourbouh, H., Garoche, P.L., Giannakopoulou, D., Pressburger, T., Schumann, J.: Bridging the Gap Between Requirements and Simulink Model Analysis. In: Joint Proceedings of REFSQ-2020 Workshops, Doctoral Symposium, Live Studies Track, and Poster Track co-located with the 26th International Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2020), Pisa, Italy, March 24, 2020. CEUR Workshop Proceedings, vol. 2584. CEUR-WS.org (2020), `http://ceur-ws.org/Vol-2584/PT-paper9.pdf`
41. Mavridou, A., Bourbouh, H., Garoche, P.L., Hejase, M.: Evaluation of the FRET and CoCoSim tools on the ten Lockheed Martin cyber-physical challenge problems. Tech. rep., NASA (Oct 2019), 84 pages
42. Mavridou, A., Bourbouh, H., Giannakopoulou, D., Pressburger, T., Hejase, M., Garoche, P.L., Schumann, J.: The ten Lockheed Martin cyber-physical challenges: Formalized, analyzed, and explained. In: Proceedings of the 2020 28th IEEE International Requirements Engineering Conference (2020)
43. Mohajerani, S., Malik, R., Fabian, M.: A framework for compositional synthesis of modular nonblocking supervisors. IEEE Transactions on Automatic Control **59**(1), 150–162 (2013)
44. Mohajerani, S., Malik, R., Fabian, M.: Compositional synthesis of supervisors in the form of state machines and state maps. Automatica **76**, 277–281 (2017)
45. Murugesan, A., Sokolsky, O., Rayadurgam, S., Whalen, M., Heimdahl, M., Lee, I.: Linking Abstract Analysis to Concrete Design: A Hierarchical Approach to Verify Medical CPS Safety. Proceedings of ICCPS'14 (April 2014)
46. Murugesan, A., Whalen, M.W., Rayadurgam, S., Heimdahl, M.P.: Compositional verification of a medical device system. In: ACM Int'l Conf. on High Integrity Language Technology (HILT) 2013. ACM (November 2013)
47. Nejati, S., Gaaloul, K., Menghi, C., Briand, L.C., Foster, S., Wolfe, D.: Evaluating model testing and model checking for finding requirements violations in simulink models. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 1015–1025 (2019)

48. Pan, G., Vardi, M.Y.: Symbolic techniques in satisfiability solving. In: SAT 2005, pp. 25–50. Springer (2005)
49. Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of Reactive(1) Designs. In: VMCAI. LNCS, vol. 3855, pp. 364–380. Springer (2006)
50. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 179–190. ACM (1989)
51. Post, A., Hoenicke, J., Podelski, A.: rt-inconsistency: A new property for real-time requirements. In: Giannakopoulou, D., Orejas, F. (eds.) Fundamental Approaches to Software Engineering - 14th International Conference, FASE 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6603, pp. 34–49. Springer (2011). https://doi.org/10.1007/978-3-642-19811-3_4, `https://doi.org/10.1007/978-3-642-19811-3_4`
52. Roth, S.: Erweiterte Konsistenzanalyse für Anforderune (Checking Extended Consistency for Requirements). Master's thesis, Karlsruhe Institute of Technology (2011), see Section 3.2.
53. Ryzhyk, L., Chubb, P., Kuz, I., Le Sueur, E., Heiser, G.: Automatic device driver synthesis with termite. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles. pp. 73–86. ACM (2009)
54. Skiena, S.S.: The algorithm design manual: Text, vol. 1. Springer Science & Business Media (1998)
55. Stachtiari, E., Mavridou, A., Katsaros, P., Bliudze, S., Sifakis, J.: Early validation of system requirements and design through correctness-by-construction. Journal of Systems and Software **145**, 52–78 (2018)
56. Tabajara, L.M., Vardi, M.Y.: Factored boolean functional synthesis. In: 2017 Formal Methods in Computer Aided Design (FMCAD). pp. 124–131. IEEE (2017)
57. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. IEEE Transactions on Software Engineering **28**(2), 183–200 (2002)