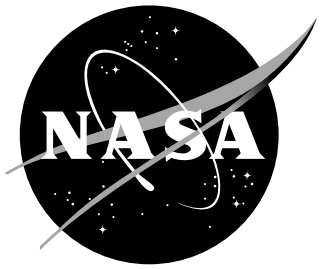
NASA/TM–2021–20210013008



Compositional Realizability Checking within FRET

Dimitra Giannakopoulou
NASA Ames Research Center, Moffett Field, CA 94035, USA

Andreas Katis
KBR, NASA Ames Research Center, Moffett Field, CA 94035, USA

Anastasia Mavridou
KBR, NASA Ames Research Center, Moffett Field, CA 94035, USA

Thomas Pressburger
NASA Ames Research Center, Moffett Field, CA 94035, USA

March 2021

NASA STI Program . . . in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA scientific and technical information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI Program operates under the auspices of the Agency Chief Information O□cer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI Program provides access to the NTRS Registered and its public interface, the NASA Technical Reports Server, thus providing one of the largest collections of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

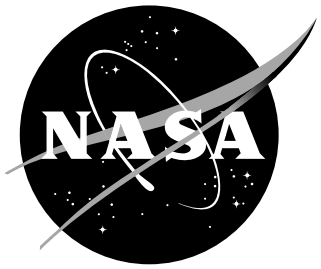
- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers, but having less stringent limitations on manuscript length and extent of graphic presentations.
- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.
- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.
- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.
- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.
- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services also include organizing and publishing research results, distributing specialized research announcements and feeds, providing information desk and personal search support, and enabling data exchange services.

For more information about the NASA STI Program, see the following:

- Access the NASA STI program home page at *http://www.sti.nasa.gov*
- E-mail your question to help@sti.nasa.gov
- Phone the NASA STI Help Desk at 757-864-9658
- Write to:
  NASA STI Information Desk
  Mail Stop 148
  NASA Langley Research Center
  Hampton, VA 23681–2199

NASA/TM–2021–20210013008



Compositional Realizability Checking within FRET

Dimitra Giannakopoulou
NASA Ames Research Center, Moffett Field, CA 94035, USA

Andreas Katis
KBR, NASA Ames Research Center, Moffett Field, CA 94035, USA

Anastasia Mavridou
KBR, NASA Ames Research Center, Moffett Field, CA 94035, USA

Thomas Pressburger
NASA Ames Research Center, Moffett Field, CA 94035, USA

National Aeronautics and
Space Administration

Ames Research Center, Mo□ett Field, CA 94035

March 2021

Acknowledgments

The authors wish to thank David Kooi for reviewing previous drafts of this report, and Mike Whalen for observing problematic decompositions of stateful assumptions. We gratefully acknowledge the NASA Aeronautics Mission Directorate's System-Wide Safety program for funding this work.

This report is available in electronic form at
http://URL

Executive Summary

An Assume-Guarantee contract for a reactive system is realizable if, for any sequence of inputs that satisfy the assumptions on the environment, the guarantees always hold. Realizability checking is essential to ensure that an implementation can be constructed that satisfies the contract. We propose a framework that supports users in the non-trivial task of developing realizable contracts. Our framework uses architectural information to automatically decompose a contract into sub-contracts that can be analyzed separately, and therefore more e□ciently. It then integrates existing algorithms in order to detect unrealizability of (sub)contracts, and to attribute unrealizability to subsets of conflicting guarantees. The latter is key for localizing and correcting the sources of unrealizability. Our approach supports this process by enabling users to interactively visualize and explore the produced conflict sets and counterexamples. We have implemented our framework in the open-source Formal Requirements Elicitation Tool (FRET), and have used it on a variety of industrial-level case studies, showcasing the strengths of our approach in terms of raw performance, as well as diagnostic potential.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Defining requirements for a complex system is a challenging, error-prone task. For a system built as a hierarchical integration of components, approaches exist for ensuring that requirements of higher-level components are achieved through the integration of lower-level ones [3–6]. However, such analyses are meaningless if the leaf components in the hierarchy cannot be constructed because of inconsistent requirements. The focus of this paper is on ensuring consistency of leaf components, thus building a solid foundation for subsequent system-level analysis.

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 [7]. The pursuit of realizable requirements comes with several challenges. While optimal algorithms exist for finite state problems over subsets of Linear Temporal Logic specifications (LTL) [8, 9], scalability issues can make the analysis impractical for realistic systems, and the use of infinite data types can render the entire problem undecidable [10,11]. Finally, upon detecting unrealizability, the designer is still left with the non-trivial task of localizing, understanding, and correcting the sources of the problem.

To address these issues, we have extended our Formal Requirements Elicitation Tool FRET [12] with a framework that: 1) incorporates powerful decision procedures for realizability; 2) decomposes realizability checking into smaller, more tractable problems; 3) traces unrealizability to subsets of requirements; and 4) provides intuitive tools for understanding and correcting unrealizability.

The work presented in this report builds upon previous work by some of the authors [13], which provided the first prototype for checking realizability of FRET requirements. Their prototype exported FRET requirements to an experimental branch of the JKind model checker [14]. That model checker supports two realizability decision procedures, namely JSyn [10,15] and JSyn-vg [11]. Both algorithms handle safety specifications expressed as Assume-Guarantee contracts, and can operate over constraints that admit infinite theories. Kooi and Mavridou also proposed a decomposition of FRET requirements into sets of non-interfering requirements, named *connected components*, which they demonstrated can increase the scalability of realizability checking for some systems.

This report extends [13] by developing a theory of decomposition in terms of assume-guarantee contracts for transition systems, thus making a direct connection with the decision procedures that are used for checking realizability. To this aim, we introduce partial contracts, which describe requirements that observe only a subset of a global system state, and 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. We apply this theory to then decompose global contracts for realizability checking, using connected components similarly to [13], but by also incorporating assumptions into the picture.

We have extended FRET with a user interface through which users can invoke realizability checking, and review the obtained results. Our framework is further enhanced with a diagnostic procedure for unrealizable results, akin to the work by Könighofer et al. [16,17]. For an unrealizable contract, we use a model-based diagnosis algorithm to attribute conflicts to subsets of requirements within the contract. Moreover, the counterexamples we produce pertain to these subsets, and are thus more targeted and easier to understand. These computed artifacts are incorporated in a user-friendly diagram that depicts the dependencies between requirements and conflicts. The diagram is also interactive in such a way that the user can identify clusters of problematic requirements at ease, while observing a counterexample as an execution trace leading to property-violating states.

The new realizability analysis framework of FRET is available, open-source, with FRET's release 2.0, at `https://github.com/NASA-SW-VnV/fret`. We report on the application of our framework to a variety of case studies. The rest of the report is organized as follows. Chapter 2 introduces a liquid mixing system, which will serve as our running example. Chapter 3 provides the formal background that our work builds upon. Chapter 4 presents our theoretical framework for decomposing realizability checking, with Chapter 5 describing the procedure for identifying connected components of a specification. The diagnosis option in FRET is presented in Chapter 6, and we discuss the implementation of our framework in FRET in Chapter 7. We evaluate our approach in Chapter 8, discuss related work in Chapter 9 and close the paper with conclusions and future work in Chapter 10.

Chapter 2

Liquid Mixer System Example

Our running example is the controller of a liquid mixing system [1] (see Figure 2.1), whose behavior must satisfy the 12 requirements shown in Table 2.1. To relate our approach with the EARS-CTRL approach presented in [1], 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 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 2.1. 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.

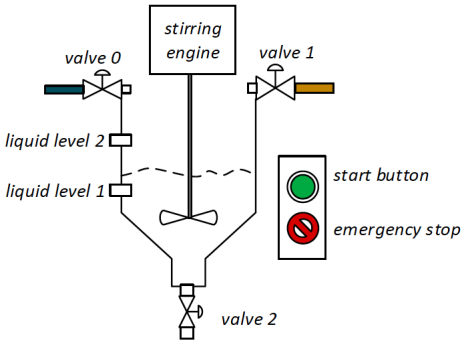


Figure 2.1: Liquid Mixing System (figure taken from Lúcio et al. [1])

Table 2.1: 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

Chapter 3

Checking Realizability

This chapter 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 [10,11]. We use two types *state* and *inputs* for a transition system (I, T) where predicate $I(s)$: *state* $\rightarrow$ *bool* denotes the set of initial states, and predicate $T(s, a, s')$: *state* $\times$ *inputs* $\times$ *state* $\rightarrow$ *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 $A(s, a)$: *state* $\times$ *inputs* $\rightarrow$ *bool* and a guarantee G, made up of two predicates: $G_I(s)$: *state* $\rightarrow$ *bool* and $G_T(s, a, s')$: *state* $\times$ *inputs* $\times$ *state* $\rightarrow$ *bool*, capturing initial-state and transitional guarantees, respectively. In practice, 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 2.1. 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, $A(s, a) = true$ and $G_I(s) = true$. Moreover, $G_T(s, a, s') = true$ if and only if the transition satisfies all requirements of Table 2.1, 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[1] [19] and Lustre [13] code. For this work, we use the Lustre code generation functionality of FRET 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[2].

[1] CoCoSpec [18] is a contract-based extension of the Lustre synchronous language.

[2] The Lustre translation may introduce variables, local to each requirement, to record history of input or state variables, e.g., to record the value of an input in the previous state. These additional variables do not a□ect our decomposition approach and are thus omitted for simplicity.

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. Formally, we can express this by defining a set of *viable* system responses using coinduction [10]:

Definition 1 (Viability of an AG contract)

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

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

Definition 2 (Realizability of an AG contract)

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

When a contract is unrealizable, a *counterexample* can be extracted to provide help in understanding the source(s) of unrealizability. A counterexample consists of a sequence of system input and output variables that demonstrates how the contract is violated (i.e., at least one guarantee is falsified) under a specific scenario.

For realizability checking, we use a combination of two o□-the-shelf algorithms, namely JSyn [10, 15] and JSyn-vg [11]. 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, and particularly the ones exercised for realizability, is not straightforward.

Chapter 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 4.1 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, output[1] variables 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.

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 $\mathit{STATES} \stackrel{\text{def}}{=} T_1 \times \ldots \times T_k$. 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:

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

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:

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

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 Chapter 3, we defined AG contracts (A, G) over the states and inputs of a system. In this chapter, we consider partial contracts (A_i, G_i) that only refer to some state variables $SV_i \subseteq SV$ of the system, i.e.:

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

In our example of Figure 4.1, 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

[1] We remind the reader that output and internal variables are considered state variables

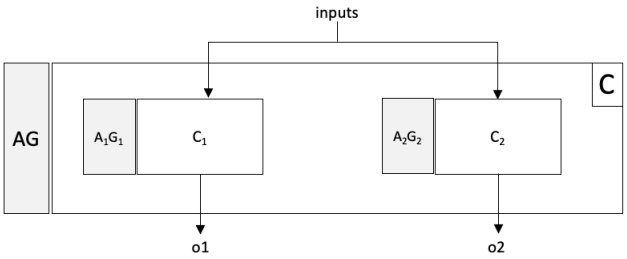


Figure 4.1: Partial Assume-Guarantee Contracts

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 $G = G_1 \wedge G_2$. 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.

Previous work by some of the authors [13] proposed a decomposition approach based on connected components to avoid common state variables. This approach was applied to requirements expressed as sets of guarantees. It was considering a definition of realizability that enforces consistency across all inputs, but was not set in the context of Assume-Guarantee contracts for reactive systems. This work extends that original observation in the context of the Assume-Guarantee contracts presented in Chapter 3, thus also matching the theory of the realizability decision procedures that we use in practice.

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 [13] by examining how to decompose global contracts in the presence of assumptions. We start by introducing a notion of realizability PRealizable$_{\{A_iG_i\}}$ for a *set* of partial contracts $\{(A_i, G_i) : i = 1, \ldots, n\}$. We then show, through Theorem 2, that in the context of contracts that do not share state, PRealizable$_{\{A_iG_i\}}$ is equivalent to ensuring that every subcontract (A_i, G_i) is realizable. Finally, we relate these to the realizability of a contract (A, G), where $A = \wedge_{i=1}^{n} A_i$, and $G = \wedge_{i=1}^{n} G_i$, through Theorem 3.

Let $\{(A_i, G_i) : i = 1, \ldots, n\}$ represent a set of n partial AG contracts for a system over *STATES* and *inputs*. We extend the notions of viability and realizability presented in Chapter 3 for such a set of partial contracts as follows.

Definition 3 (Partial-contract viability)

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

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 $\{A_i, G_i\}$ as:

Definition 4 (Partial-contract Realizability)

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

The following theorem establishes that partial contract realizability ($\mathsf{PRealizable}_{\{A_iG_i\}}$) implies the realizability of a global contract that combines the partial contracts ($\mathsf{Realizable}_{AG}$).

Definition 5 (Combining Predicates) *Let P be a predicate over a set of variables S and let P_i with $i = 1 \ldots n$ be predicates over sets of variables $S_i \subseteq S$. We use $P = \wedge_{i=1}^{n} P_i$ to denote that $\forall s \in S, P(s) = \wedge_{i=1}^{n} P_i(s@S_i)$.*

Theorem 1 *Let (A, G) be an AG contract over state variable set SV, and let $\{A_i, G_i\}$ be a set of partial contracts over state variable sets $SV_i \subseteq SV$, where $A = \wedge_{i=1}^{n} A_i$ and $G = \wedge_{i=1}^{n} G_i$ (i.e., $G_I = \wedge_{i=1}^{n} G_{Ii}$ and $G_T = \wedge_{i=1}^{n} G_{Ti}$). Then* $\mathsf{PRealizable}_{\{A_iG_i\}} \Rightarrow \mathsf{Realizable}_{AG}$.

Proof. We first prove by coinduction that $\mathsf{PViable}_{\{A_iG_i\}}(s) \Rightarrow \mathsf{Viable}_{AG}(s)$. The coinductive hypothesis is:

$$\begin{aligned}&[\mathsf{PViable}_{\{A_iG_i\}}(s') \Rightarrow \mathsf{Viable}_{AG}(s')] \Rightarrow\\&[\mathsf{PViable}_{\{A_iG_i\}}(s) \Rightarrow \mathsf{Viable}_{AG}(s)]\end{aligned} \tag{4.1}$$

(from Definition 3)

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

$\Rightarrow$ (from $((p \vee q) \Rightarrow r) \Rightarrow ((p \wedge q) \Rightarrow r)$)

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

$\Rightarrow$ (from $((p \Rightarrow r) \wedge (q \Rightarrow s)) \Rightarrow ((p \wedge q) \Rightarrow (r \wedge s))$)

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

$\Rightarrow$ (from $(p \Rightarrow (p \Rightarrow q)) \equiv (p \Rightarrow q)$

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

$\Rightarrow$ (from coinductive hypothesis 4.1)

$$\forall a : \textit{inputs}.(\wedge_{i=1}^{n} A_i(s@SV_i, a)) \Rightarrow$$
$$\exists s'.[(\wedge_{i=1}^{n} G_{Ti}(s@SV_i, a, s'@SV_i)) \wedge \mathsf{Viable}_{AG}(s')]$$

$\Rightarrow$ (from $A = \wedge_{i=1}^{n} A_i$ and $G = \wedge_{i=1}^{n} G_i$ and Definition5)

$$\forall a : \textit{inputs}.A(s, a) \Rightarrow \exists s'.G(s, a, s') \wedge \mathsf{Viable}_{AG}(s')$$

$\equiv$ (from Definition 1)

$$\mathsf{Viable}_{AG}(s)$$

We use the above to prove $\mathsf{PRealizable}_{\{A_iG_i\}} \Rightarrow \mathsf{Realizable}_{AG}$ as follows.

(from Definition 4)

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

$\Rightarrow$ (from first part of proof)

$$\exists s.\ \wedge_{i=1}^{n}\ G_{Ii}(s@SV_i) \wedge \mathsf{Viable}_{AG}(s)$$

$\Rightarrow$ (from $G_I = \wedge_{i=1}^{n} G_{Ii}$ and Definition 5)

$$\exists s.\, G_I(s) \wedge \mathsf{Viable}_{AG}(s)$$

$\equiv$ (from Definition 2)

$$\mathsf{Realizable}_{AG}$$

Following our observations of [13], we call a *non-interfering contract set over* SV, a set of partial contracts (A_i, G_i) over SV_i i□ 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 2 below.

Theorem 2 *Let* $\{A_i, G_i\}$ *be a non-interfering contract set. Then:*

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

We first establish the following lemma:

Lemma 1 *Let* $\{A_i, G_i\}$ *be a set of partial contracts over state variable sets* $\{SV_i\}$ *which partition a set of state variables* SV*. Then:*

$$(\wedge_{i=1}^{n} \mathsf{Viable}_{A_iG_i}(s@SV_i)) \Leftrightarrow \mathsf{PViable}_{\{A_iG_i\}}(s)$$

Proof. $\Rightarrow$. Proof by coinduction. We will prove that:

$$[(\wedge_{i=1}^{n}\mathsf{Viable}_{A_iG_i}(s'@SV_i)) \Rightarrow \mathsf{PViable}_{\{A_iG_i\}}(s')] \Rightarrow \\ [(\wedge_{i=1}^{n}\mathsf{Viable}_{A_iG_i}(s@SV_i)) \Rightarrow \mathsf{PViable}_{\{A_iG_i\}}(s)] \tag{4.2}$$

We start with:

$$\wedge_{i=1}^{n}\mathsf{Viable}_{A_iG_i}(s@SV_i) \tag{4.3}$$

Let a : *inputs* and $s \in$ *STATES*. For any contract (A_i, G_i), since $\mathsf{Viable}_{A_iG_i}(s@SV_i)$, we know by Definition 1 that:

$$A_i(s@SV_i, a) \Rightarrow \exists s_i'.\ (G_{Ti}(s@SV_i, a, s_i') \wedge \mathsf{Viable}_{A_iG_i}(s_i')) \tag{4.4}$$

If $A_i(s@SV_i, a)$ holds for (A_i, G_i), we pick s_i' as per Equation 4.4; that equation also guarantees that $\mathsf{Viable}_{A_iG_i}(s_i')$. If $A_i(s@SV_i, a)$ does not hold, we set $s_i' = s@SV_i$; in this case $\mathsf{Viable}_{A_iG_i}(s_i')$ is due to Equation 4.3. Because subsets SV_i partition SV, we can uniquely combine all these s_i' into a state s', such that $s'@SV_i = s_i'$. By construction of s', $\wedge_{i=1}^{n}\mathsf{Viable}_{A_iG_i}(s_i') = \wedge_{i=1}^{n}\mathsf{Viable}_{A_iG_i}(s'@SV_i)$ which, from our Coinduction Hypothesis 4.2, implies $\mathsf{PViable}_{\{A_iG_i\}}(s')$. For s, we have thus constructed an s' for which:

$$(\wedge_{i=1}^{n}(A_i(s@SV_i, a) \Rightarrow G_{Ti}(s@SV_i, a, s'@SV_i))) \wedge \mathsf{PViable}_{\{A_iG_i\}}(s') \tag{4.5}$$

This proves that $\mathsf{PViable}_{\{A_iG_i\}}(s)$ when at least one $A_i(s@SV_i, a)$ holds (see Definition 3). If no $A_i(s@SV_i, a)$ holds, then $\mathsf{PViable}_{\{A_iG_i\}}(s)$ holds trivially.
$\Leftarrow$. The proof is by coinduction. We will prove that:

$$[\mathsf{PViable}_{\{A_iG_i\}}(s') \Rightarrow \wedge_{i=1}^{n}\mathsf{Viable}_{A_iG_i}(s'@SV_i)] \Rightarrow \\ [\mathsf{PViable}_{\{A_iG_i\}}(s) \Rightarrow \wedge_{i=1}^{n}\mathsf{Viable}_{A_iG_i}(s@SV_i)] \tag{4.6}$$

Let us assume that $\wedge_{i=1}^{n}\mathsf{Viable}_{A_iG_i}(s@SV_i)$ is false but $\mathsf{PViable}_{\{A_iG_i\}}(s)$ is true; i.e., a proof by contradiction. Then for some i, $\mathsf{Viable}_{A_iG_i}(s@SV_i)$ is false, which in turn means that for some $a : inputs$:

$$A_i(s@SV_i, a)) \wedge [\nexists s_i'.(G_{Ti}(s@SV_i, a, s_i')) \wedge \mathsf{Viable}_{A_iG_i}(s_i')] \tag{4.7}$$

Since $A_i(s@SV_i, a)$ (Equation 4.7) and $\mathsf{PViable}_{\{A_iG_i\}}(s)$ hold, we know by Definition 3 that:

$$\exists s'.[(A_i(s@SV_i, a) \Rightarrow G_{Ti}(s@SV_i, a, s'@SV_i)) \wedge \mathsf{PViable}_{\{A_iG_i\}}(s')] \tag{4.8}$$

Let us set $s_i' = s'@SV_i$. By the Coinduction Hypothesis 4.6, we have $\mathsf{PViable}_{\{A_iG_i\}}(s') \Rightarrow \mathsf{Viable}_{A_iG_i}(s_i')$. We have thus found an s_i', for which $G_{Ti}(s@SV_i, a, s_i') \wedge \mathsf{Viable}_{A_iG_i}(s_i')$, which contradicts the second conjunct of Equation 4.7.

We now use Lemma 1 to prove Theorem 2.

Proof. $\Rightarrow$. We start with $\wedge_{i=1}^{n}\mathsf{Realizable}_{A_iG_i}$. From Definition 2, for each (A_i, G_i), $\exists s_i$ such that $G_{Ii}(s_i) \wedge \mathsf{Viable}_{A_iG_i}(s_i)$. Because subsets SV_i partition SV, we can uniquely combine all these s_i into a state s, such that $s@SV_i = s_i$. For this s, we thus have that $\wedge_{i=1}^{n}\mathsf{Viable}_{A_iG_i}(s_i) = \wedge_{i=1}^{n}\mathsf{Viable}_{A_iG_i}(s@SV_i)$, and from Lemma 1, we conclude that $\mathsf{PViable}_{\{A_iG_i\}}(s)$. Moreover, we have established that $\wedge_{i=1}^{n}G_{Ii}(s_i) = \wedge_{i=1}^{n}G_{Ii}(s@SV_i)$. The existence of s proves $\mathsf{PRealizable}_{\{A_iG_i\}}$ (see Definition 4).
$\Leftarrow$. We start with $\mathsf{PRealizable}_{\{A_iG_i\}}$, i.e. $\exists s.\ \wedge_{i=1}^{n}G_{Ii}(s@SV_i) \wedge \mathsf{PViable}_{\{A_iG_i\}}(s)$ (see Def. 4). For each (A_i, G_i), we set $s_i = s@SV_i$. From Lemma 1, $\mathsf{PViable}_{\{A_iG_i\}}(s)$ implies that $\mathsf{Viable}_{A_iG_i}(s@SV_i) = \mathsf{Viable}_{A_iG_i}(s_i)$. Therefore, for each (A_i, G_i), we have established an s_i such that $G_{Ii}(s_i) \wedge \mathsf{Viable}_{A_iG_i}(s_i)$, which implies $\mathsf{Realizable}_{A_iG_i}$ (see Definition 2).

In other words, for non-interfering contract sets, partial-contract realizability is equivalently decomposed into realizability of the individual partial contracts. By combining Theorem 1 and Theorem 2, we obtain the following Corollary:

Corollary 1 *Let (A, G) be an Assume-Guarantee contract, and let (A_i, G_i) be a non-interfering contract set where $A = \wedge_{i=1}^{n} A_i$ and $G = \wedge_{i=1}^{n} G_i$ (i.e., $G_I = \wedge_{i=1}^{n} G_{Ii}$ and $G_T = \wedge_{i=1}^{n} G_{Ti}$). Then:*

$$(\wedge_{i=1}^{n} \mathsf{Realizable}_{A_i G_i}) \Rightarrow \mathsf{Realizable}_{AG}$$

In other words, checking realizability of non-interfering partial contracts can be used to infer realizability of their conjunction in a global contract. Unfortunately, unrealizability of a partial contract does not imply unrealizability of the corresponding global contract, in the general case. It therefore remains for us to discover conditions under which partial-contract realizability actually coincides with global contract realizability. The latter would allow us to also infer unrealizability of a global contract when a partial contract is unrealizable. By comparing the definitions of Viable_{AG} and $\mathsf{PViable}_{\{A_i G_i\}}$, 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 $A_1 \equiv A_2 \equiv \ldots \equiv A_n$ (from Boolean algebra) [2]. Additionally, since $A = \wedge_{i=1}^{n} A_i$, $A \equiv A_1 \equiv A_2 \equiv \ldots \equiv A_n$.

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 $A(s, a)$: *state* $\times$ *inputs* $\rightarrow$ *bool* is considered *independent of state*, if $\forall s_1, s_2 \in$ *state*, $\forall a \in$ *inputs*. $A(s_1, a) = A(s_2, a)$. We abbreviate $A(*, a)$ by $A(a)$. The following theorem captures these observations.

Theorem 3 *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* $\mathsf{Realizable}_{AG} \equiv (\wedge_{i=1}^{n} \mathsf{Realizable}_{AG_i})$.

Proof. We first prove by coinduction that $(\wedge_{i=1}^{n} \mathsf{Viable}_{AG_i}(s@SV_i)) \equiv \mathsf{Viable}_{AG}(s)$. The coinductive hypothesis is:

$$\begin{aligned}[(\wedge_{i=1}^{n} \mathsf{Viable}_{AG_i}(s'@SV_i)) \equiv \mathsf{Viable}_{AG}(s')] \Rightarrow \\ [(\wedge_{i=1}^{n} \mathsf{Viable}_{AG_i}(s@SV_i)) \equiv \mathsf{Viable}_{AG}(s)]\end{aligned} \tag{4.9}$$

(from Definition 1)

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

(given: $G = \wedge_{i=1}^{n} G_i$)

$$\equiv \forall a.\ (A(a) \Rightarrow \exists s'.(\wedge_{i=1}^{n} G_{Ti}(s@SV_i, a, s'@SV_i)) \wedge \mathsf{Viable}_{AG}(s'))$$

(from coinductive hypothesis 4.9 and Lemma 1)

$$\equiv \forall a.\ (A(a) \Rightarrow \exists s'.(\wedge_{i=1}^{n} G_{Ti}(s@SV_i, a, s'@SV_i)) \wedge \mathsf{PViable}_{\{AG_i\}}(s'))$$

[2] In the future, we plan to study if it is possible to weaken this condition.

(from $(p \Rightarrow q) \equiv p \Rightarrow (p \Rightarrow q)$ and distributing $\Rightarrow$ over $\wedge$)

$$\equiv \forall a.\ (A(a) \Rightarrow \exists s'.(\wedge_{i=1}^{n}(A(a) \Rightarrow G_{Ti}(s@SV_i, a, s'@SV_i))) \wedge \mathsf{PViable}_{\{AG_i\}}(s'))$$

(from Definition 3)

$$\equiv \mathsf{PViable}_{\{AG_i\}}(s)$$

(from Lemma 1)

$$\equiv (\wedge_{i=1}^{n}\mathsf{Viable}_{AG_i}(s@SV_i))$$

We use the above to prove $\mathsf{Realizable}_{AG} \equiv (\wedge_{i=1}^{n}\mathsf{Realizable}_{AG_i})$ as follows.

(from Definition 2)

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

(from $G_I = \wedge_{i=1}^{n} G_{Ii}$ and the fact that $(\wedge_{i=1}^{n}\mathsf{Viable}_{AG_i}(s@SV_i)) \equiv \mathsf{Viable}_{AG}(s)$)

$$\equiv \exists s.\ (\wedge_{i=1}^{n} G_{Ii}(s@SV_i)) \wedge (\wedge_{i=1}^{n}\mathsf{Viable}_{AG_i}(s@SV_i))$$

(after rearranging conjuncts)

$$\equiv \exists s.\ (\wedge_{i=1}^{n}(G_{Ii}(s@SV_i) \wedge \mathsf{Viable}_{AG_i}(s@SV_i)))$$

(from the fact that each (A, G_i) is defined over SV_i and SV_i partition SV)

$$\equiv \wedge_{i=1}^{n}(\exists s@SV_i.\ G_{Ii}(s@SV_i) \wedge \mathsf{Viable}_{AG_i}(s@SV_i))$$

(from Definition 2)

$$\equiv \wedge_{i=1}^{n}\mathsf{Realizable}_{AG_i}$$

Chapter 5

Connected Components

In this chapter, 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 3. As discussed, Theorem 3 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 3 by splitting the guarantees G based on the notion of connected components [20, 21] for undirected graphs. As seen in Figure 5.1, 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 Chapter 3, AG contracts (A, G) are typically expressed as sets of assumption and guarantee predicates, with A and G corresponding to their conjunctions. We use R to represent all guarantee predicates involved in G, without differentiating between initial state and transitional guarantees. We refer to predicates in 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 5.1 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 requirement [LM-005]), *timer_120sec_start* (referenced only by requirement [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. 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 $\forall s_1, s_2 \in \mathit{STATES}.\, (s_1@SV_{Ri} = s_2@SV_{Ri}) \Rightarrow R_i(s_1) = R_i(s_2)$. For each R_i we can therefore define the predicate $R_i@SV_{Ri}$ that behaves as R_i, but has lower dimensionality when $SV_{Ri} \subset SV$:

$$\forall s_i \in \mathit{STATES}@SV_{Ri}.$$
$$R_i@SV_{Ri}(s_i) \stackrel{\text{def}}{=} \exists s \in \mathit{STATES}.\, ((s@SV_{Ri} = s_i) \wedge R_i(s)).$$

The above notations and definitions naturally extend to transitional guarantees.

Definition 6 (Requirements Graph) *A* requirements graph RG *is an undirected graph*

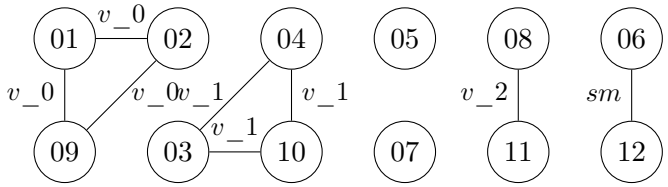


Figure 5.1: `liquid_mixer` connected components. We use the last two digits of req. names, and abbreviate `valve_x` to `v_x`, `stirring_motor` to `sm`

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 the requirements graph.

Definition 7 (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, $SV_c = \bigcup_{R_i \in R_c} SV_{Ri}$.*

State-connected components can be computed with a connected component algorithm. For example, our framework implements Tarjan's classic connected components algorithm [20] with an $O(|V| + |E|)$ complexity. State-connected components can then be used to create AG contracts as follows.

Definition 8 (Connected AG Contract) *Let (A, G) be an AG contract over state variables SV, and RG be its corresponding requirements graph, where A is independent of state. 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_c is made up of $G_{Ic} = \wedge_{i=1}^{m} R_i@SV_c$, and $G_{Tc} = \wedge_{j=1}^{n} R_j@SV_c$, where $R_i \in R_c$ are initial state guarantees, and $R_j \in R_c$ are transitional guarantees.*

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 3.

Of the six connected components in `liquid_mixer` (Figure 5.1), only one was found to be unrealizable, consisting of requirements [LM-001], [LM-002] and [LM-009]. Using the diagnosis algorithm over the unrealizable connected component yielded a single minimal conflict between [LM-001] and [LM-009], a result consistent with the findings by Lúcio et al. [1].

Chapter 6

Diagnosing Unrealizability

Realizability checking can save developers from embarking on an impossible task. But how does one proceed with the information that a set of requirements is unrealizable? Connected components may reduce the number of requirements involved in realizability checking. However, for realistic systems, one may still end up having to debug a large number of requirements. In this chapter, we present a framework for localizing and understanding the reasons of unrealizability. The framework: 1) combines existing realizability detection and diagnosis algorithms, and 2) proposes a visualization approach to help developers with "debugging" unrealizability.

6.1 Finding Minimal Conflicts

Given a set of unrealizable requirements, we use the algorithms presented by Könighofer et al [16,17] to compute all minimal unrealizable sets of requirements, called *minimal conflicts*. The approach is based on the concept of *model-based diagnosis* [22]. In the context of realizability, model-based diagnosis produces all possible options, named *hitting sets*, available for fixing a set U of unrealizable requirements. If M is the set of minimal conflicts in U, a hitting set H of requirements is such that $\forall m \in M.\, H \cap m \neq \varnothing$. In our experience, this information is not useful in practice. On the other hand, the computation of *all* minimal conflicts within U, which is computed in the process, is valuable for localizing sources of unrealizability.

Model-based diagnosis relies on external "services" to: 1) decide if a set of requirements is realizable, and 2) obtain a minimal conflict for a set of unrealizable requirements. Our implementation uses JSyn-vg as the main realizability decision procedure. For the production of minimal conflicts, we use *delta-debugging* [23].

Given a set of unrealizable requirements, delta-debugging detects a minimal conflict in a divide-and-conquer fashion. We describe the algorithm at a high level, through an example run on the `liquid_mixer` component that contains Requirements [LM-001], [LM-002], [LM-009].

The algorithm uses a granularity parameter n, initially $n = 2$, to dictate the number of partitions (denoted by $\Delta_1 \ldots \Delta_n$) that the original set of requirements is split into. In Steps 1 and 2 (Table 6.1), the algorithm checks the realizability of the initial partitions $\{1, 2\}$ and $\{9\}$, as well as their complements (denoted by ∇). Since $\Delta_1 = \nabla_2$ and $\Delta_2 = \nabla_1$, the complements do not need to be examined. Note that in this example, we do not have assumptions. In the general case, all assumptions participate in every partition generated

Table 6.1: Delta Debugging on SCC ([LM-001], [LM-002], [LM-009])

Step	Test case	Reqs	Realizable?
1	$\Delta_1 = \nabla_2$	1 2 .	4
2	$\Delta_2 = \nabla_1$	. . 9	4
3	Δ_1	1 . .	4
4	Δ_2	. 2 .	4
5	Δ_3	. . 9	4
6	∇_1	. 2 9	4
7	$\nabla_{\mathbf{2}}$	1 . 9	8
8	∇_3	1 2 .	4
9	$\Delta_1 = \nabla_2$	1 . .	4
10	$\Delta_2 = \nabla_1$	. . 9	4
Result		1 . 9	

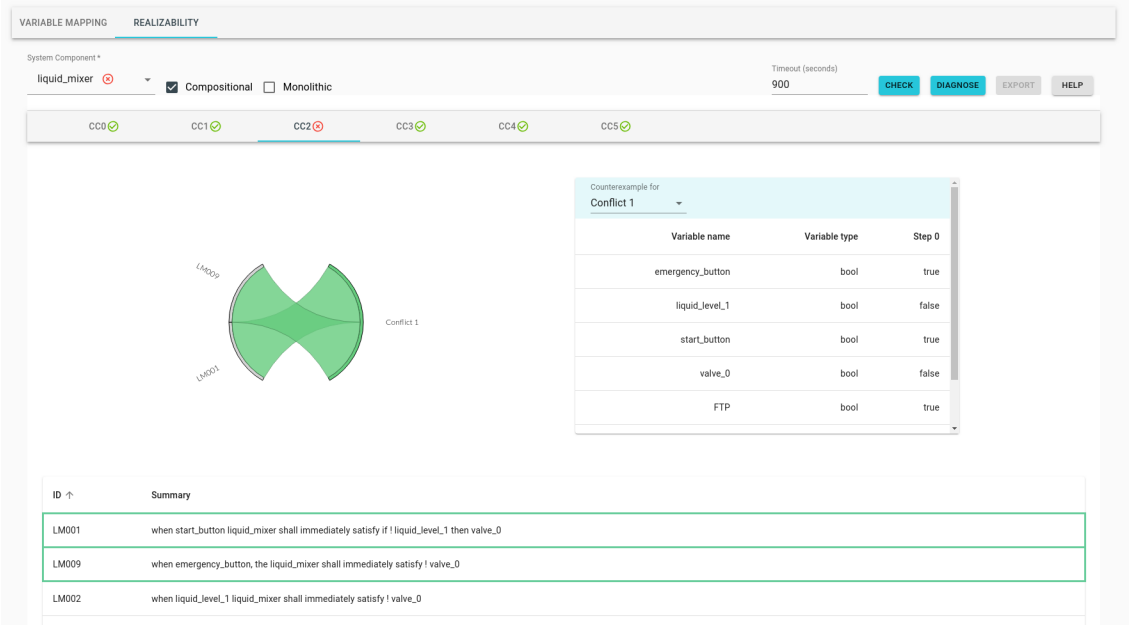


Figure 6.1: Focused view of `liquid_mixer` diagram on conflict [LM-001, LM-009]

during the delta-debugging algorithm.

Since all these sets are realizable, the algorithm increases the granularity to $n = 3$, creating partitions $\{1\}$, $\{2\}$, $\{9\}$, as well as the complements $\{2, 9\}$, $\{1, 9\}$, and $\{1, 2\}$. These are checked for realizability in Steps 3–8. In our implementation, we cache intermediate results to avoid redundant checks, so Step 8 simply reuses the results of Step 1. The set $\{1, 9\}$ is found to be unrealizable and the algorithm focuses on that set, reiterating the steps of partitioning and checking. It soon discovers that no unrealizable partition of $\{1, 9\}$ exists, and returns $\{1, 9\}$ as the minimal conflict.

The model-based diagnosis algorithm generates and maintains a *hitting set tree*, where nodes are minimal conflicts and edges are labeled with a requirement that appears in the source node's label. The invariant is that each node is labeled with a conflict that does not contain requirements appearing on the edges that define the path towards the root.

Initially, an unlabeled node for the tree's root is generated. This node is labeled with a minimal conflict returned by delta-debugging. In the next step, a child node of the newly labeled node is created for each requirement appearing in the parent's label. From here on, the tree is expanded in a breadth-first fashion, by labeling each frontier node with a minimal conflict that satisfies the invariant of the hitting set tree. These labels are obtained by running delta-debugging over a factored version of the requirement set, where requirements that are part of the labeled path (edges) leading to the current node are removed.

Since only one minimal conflict exists in `liquid_mixer`, the final tree is composed of three nodes, with the root labeled by the minimal conflict and the two leaves labeled by the fact that the rest of the specification is realizable when either [LM-001] or [LM-009] are not part of it. Note that the computation of SCCs has a direct impact to the performance of the model-based diagnosis algorithm, as it can dramatically reduce the number of requirements that are provided as input. For `liquid_mixer`, this number corresponds to just three requirements out of the original twelve that would be considered, had SCCs not been computed.

6.2 Monotonicity of Unrealizability

The delta-debugging algorithm presented in the previous section is applicable when the test that is performed is monotonic. In our context, this means that, unrealizability of any sub-contract targeted by delta-debugging must imply unrealizability of the original contract being diagnosed. We remind the reader that, in Chapter 4, we observed that unrealizability of a partial contract does not imply unrealizability of the corresponding global contract, in the general case. We prove here that when the partial contract and the global contract have the same assumption, as is the case with our delta-debugging implementation, monotonicity holds.

Lemma 2 *Let* (A, G) *and* (A, G_{new}) *be Assume-Guarantee contracts over state variables* SV *and* SV_{new}*, respectively, and let* $(A, G_{\supset})$ *be an Assume-Guarantee contract where* $G_{\supset} = G \wedge G_{new}$ *over state variables* $SV_{\supset} = SV \cup SV_{new}$*. Then :*

$$\mathsf{Viable}_{AG_{\supset}}(s) \Rightarrow (\mathsf{Viable}_{AG}(s@SV) \wedge \mathsf{Viable}_{AGnew}(s@SV_{new}))$$

Proof. To prove this, we use coinduction:

$$\begin{array}{l}[\mathsf{Viable}_{AG_{\supset}}(s') \Rightarrow (\mathsf{Viable}_{AG}(s'@SV) \wedge \mathsf{Viable}_{AGnew}(s'@SV_{new}))] \Rightarrow \\ [\mathsf{Viable}_{AG_{\supset}}(s) \Rightarrow (\mathsf{Viable}_{AG}(s@SV) \wedge \mathsf{Viable}_{AGnew}(s@SV_{new}))]\end{array} \tag{6.1}$$

(from Definition 1)

$$\mathsf{Viable}_{AG_{\supset}}(s) \stackrel{\text{def}}{=} \forall a.A(s,a) \Rightarrow \exists s'.G_{T\supset}(s,a,s') \wedge \mathsf{Viable}_{AG_{\supset}}(s')$$

$\equiv$ (from the fact that $G_{\supset} = G \wedge G_{\mathsf{new}}$ and $SV_{\supset} = SV \cup SV_{new}$)

$$\begin{array}{l}\forall a.A(s,a) \Rightarrow \\ \exists s'.G_T(s@SV, a, s'@SV) \wedge G_{Tnew}(s@SV, a, s'@SV_{new}) \wedge \mathsf{Viable}_{AG_{\supset}}(s')\end{array}$$

$\Rightarrow$ (from the coinductive hypothesis 6.1)

$$\begin{array}{l}\forall a.\ A(s,a) \Rightarrow \exists s'.\ [G_T(s@SV, a, s'@SV) \wedge G_{Tnew}(s@SV_{new}, a, s'@SV_{new}) \wedge \\ \qquad\qquad \mathsf{Viable}_{AG}(s'@SV) \wedge \mathsf{Viable}_{AGnew}(s'@SV_{new})]\end{array}$$

$\Rightarrow$ (distribute $\exists$ over $\wedge$; note this is an implication not an equivalence)

$$\begin{array}{l}\forall a.A(s,a) \Rightarrow [(\exists s'@SV.\ G_T(s@SV, a, s'@SV) \wedge \mathsf{Viable}_{AG}(s'@SV)) \wedge \\ (\exists s'@SV_{new}.\ G_{Tnew}(s@SV_{new}, a, s'@SV_{new}) \wedge \mathsf{Viable}_{AGnew}(s'@SV_{new}))]\end{array}$$

$\equiv$ (from the fact that $p \Rightarrow (q \wedge r) \equiv (p \Rightarrow q) \wedge (p \Rightarrow r)$)

$$[\forall a.A(s@SV,a) \Rightarrow \exists s'@SV.\ G_T(s@SV,a,s'@SV) \wedge \mathsf{Viable}_{AG}(s'@SV)] \wedge$$
$$[\forall a.A(s@SV_{new},a) \Rightarrow \exists s'@SV_{new}.\ G_{Tnew}(s@SV_{new},a,s'@SV_{new}) \wedge$$
$$\mathsf{Viable}_{AGnew}(s'@SV_{new})]$$

$\equiv$ (from Definition 1)

$$\mathsf{Viable}_{AG}(s@SV) \wedge \mathsf{Viable}_{AGnew}(s@SV_{new})$$

Theorem 4 (Monotonicity of unrealizability in an Assume-Guarantee contract) *Let (A,G) and (A,G_{new}) be Assume-Guarantee contracts over state variables SV and SV_{new}, respectively, and let $(A,G_{\supset})$ be an Assume-Guarantee contract where $G_{\supset} = G \wedge G_{new}$ over state variables $SV_{\supset} = SV \cup SV_{new}$. Then:*

$$\neg\mathsf{Realizable}_{AG} \Rightarrow \neg\mathsf{Realizable}_{AG_{\supset}}$$

Proof. (by contradiction) Assume that:

$$\neg\mathsf{Realizable}_{AG} \wedge \mathsf{Realizable}_{AG_{\supset}}$$

(from Definition 2)

$$\mathsf{Realizable}_{AG_{\supset}} \stackrel{\text{def}}{=} \exists s.\ G_{I\supset}(s) \wedge \mathsf{Viable}_{AG_{\supset}}(s)$$

$\equiv$ (from the fact that $G_{\supset} = G \wedge G_{\mathsf{new}}$ and that $SV_{\supset} = SV \cup SV_{new}$)

$$\exists s.\ G_I(s@SV) \wedge G_{Inew}(s@SV_{new}) \wedge \mathsf{Viable}_{AG_{\supset}}(s)$$

$\Rightarrow$ (from Lemma 2)

$$\exists s.\ G_I(s@SV) \wedge G_{Inew}(s@SV_{new}) \wedge \mathsf{Viable}_{AG}(s@SV) \wedge \mathsf{Viable}_{AGnew}(s@SV_{new})$$

$\Rightarrow$ (distribute $\exists$ over $\wedge$)

$$(\exists s@SV.\ G_I(s@SV) \wedge \mathsf{Viable}_{AG}(s@SV)) \wedge$$
$$(\exists s@SV_{new}.\ G_{Inew}(s@SV_{new}) \wedge \mathsf{Viable}_{AGnew}(s@SV_{new}))$$

$\equiv$ (from Definition 2)

$$\mathsf{Realizable}_{AG} \wedge \mathsf{Realizable}_{AGnew}$$

which contradicts our hypothesis that $\neg\mathsf{Realizable}_{AG}$.

6.3 Visualizing Unrealizability

The artifacts provided by the aforementioned techniques for specification decomposition and diagnosis of unrealizable contracts are not elegant enough to be presented to the engineer in their original form. Visualization of results is often required in order to truly understand the dependencies between requirements. This is also particularly true for scenarios where the engineer has to identify the reason why the given contract is declared unrealizable. To that end, we created a graphical user interface (GUI) that precisely captures essential information through a compact representation of the dependencies between requirements and minimal conflicts. Furthermore, the interface allows the user to interact with the visualization, in order to focus on particular conflicts, or dependencies.

We believe that the most e□cient way to achieve the above is through the use of *chord diagrams* [24]. A chord diagram is a graphic representation of interrelationships between data, where each individual element is placed along the perimeter of a circular construct and relationships are depicted through edges between elements. An important feature of chord diagrams is the ability to maintain a clear representation of dependencies through *hierarchical edge bundling* [25], even when the size of data is relatively large, e.g., contracts with tens or hundreds of requirements.

Figure 8.1a shows the chord diagram that is generated for the `Infusion_Manager` system, an unrealizable contract in our case studies with 26 requirements and 8 minimal conflicts. These requirements and conflicts define the input data to the chord diagram, which depicts each set using a distinguishable arc on the circular pattern (left and right arc, respectively). Chords, i.e. edges, connect each requirement to the conflicts that it appears in, with each edge being assigned a distinct color that matches the color-coded conflicts. While hierarchical edge bundling helps us maintain a clear total view, it may be the case that the engineer would like to focus on a particular subset of dependencies, related to either a particular requirement or a specific conflict. We enable this through interactive means where parts of the diagram that are not related to the selected element can be filtered out. Two examples are presented in Figures 8.1b and 8.1c, which depict a focused view on a specific conflict and requirement from the chord diagram in Figure 8.1a, respectively.

Figure 6.1 provides a snapshot of the overall GUI for realizability checking in FRET, using the `liquid_mixer` as example. As soon as the system component is selected, the SCCs are computed. If multiple SCCs exist, a focused view is provided for each one ('CCn' tabs in Figure 6.1), where the user can initally see which requirements participate in each SCC via a table that is dynamically graying out unrelated requirements. As soon as the SCCs are computed, the realizability checking options become available. The user can then choose to run the analysis either compositionally or monolithically. In the cases where a compositional analysis is possible, a realizability check is performed over each SCC separately.

When the necessary checks are completed the realizability result is displayed for the system component. In the case of compositional analysis, each SCC has its own result. When the system component is unrealizable, the user has the option to diagnose the specification, focusing on a particular SCC (if applicable). As soon as the diagnosis step is complete, the SCC tab is updated with a chord diagram. The user can then interact with the diagram to observe dependencies between requirements and conflicts, as well as a counterexample for each conflict. As the user is interacting with the diagram, the table of requirements is dynamically sorted such that the related requirements appear on the top and are outlined with the color that corresponds to the one used for the conflict within the diagram. The user can then focus on particular conflicts, fix issues and repeat the process. SCCs are computed

anew each time a requirement is updated in the system database.

We believe that the above features optimize the ability of an engineer to quickly transition from requirements to conflicts and focus on system elements that appear to contribute the most to the source of unrealizability. The end product is a minimalistic but concise representation of the problem at hand, especially when compared against the set of diagnoses returned by previous work [16].

Chapter 7

Implementation in FRET

We have implemented our approach to analyzing realizability of Assume-Guarantee contracts as a native feature of FRET. The realizability check can be performed either monolithically on the complete set of assumptions and guarantees, or compositionally, on computed partial contracts. Realizability analysis is made available to the users through the FRET analysis portal, previously dedicated to exporting specifications to the CoCoSim model checker [26]. The variable mapping supported in the portal enables users to indicate whether variables appearing in FRETish requirements are inputs, outputs, or internal. The formulas generated by FRET are then transformed into specifications that can be used by the JSyn and JSyn-vg algorithms, as implemented by the first FRET prototype [13].

As FRET does not distinguish between assumptions and guarantees, we initially treated all requirements as guarantees. However, as observed in [13], if requirements that only refer to system inputs are treated as guarantees, they become by default unrealizable. This is because inputs are not controllable by the system. Such requirements were previously excluded from realizability analysis. In our new implementation, and given the theory of Chapter 4, we automatically flag such requirements as assumptions for the purpose of realizability analysis. In the future, we plan to enable FRET users to explicitly indicate if a FRETish sentence corresponds to an assumption or a guarantee, thus allowing them to also formulate stateful assumptions.

Decomposition of requirements is performed at the FRETish level, as opposed to the level of their corresponding formulas. If requirements contain stateful assumptions, i.e., assumptions that include both input and state variables, FRET does not perform decomposition. The current implementation analyzes such contracts monolithically by taking into account the complete set of requirements. Our decomposition algorithm can be used inside or outside FRET. When used within FRET, the decomposition is applied directly at the level of FRETish requirements. When used externally, it takes as input a Lustre specification file and it parses it to gather all the necessary information for the decomposition. Due to this flexibility of our implementation, we were able to apply the decomposition and perform analysis not only on case studies for which the requirements were written directly in FRET, but also for case studies for which we received the contracts directly in Lustre.

The implementation of the decomposition algorithm is written in JavaScript, while the visualization of results relies on the use of Material-UI React components and the D3 library [27, 28]. External dependencies include the Jkind model checker (which includes the JSyn and JSyn-vg algorithms), as well as the Z3 and AE-VAL SMT solvers [29, 30].

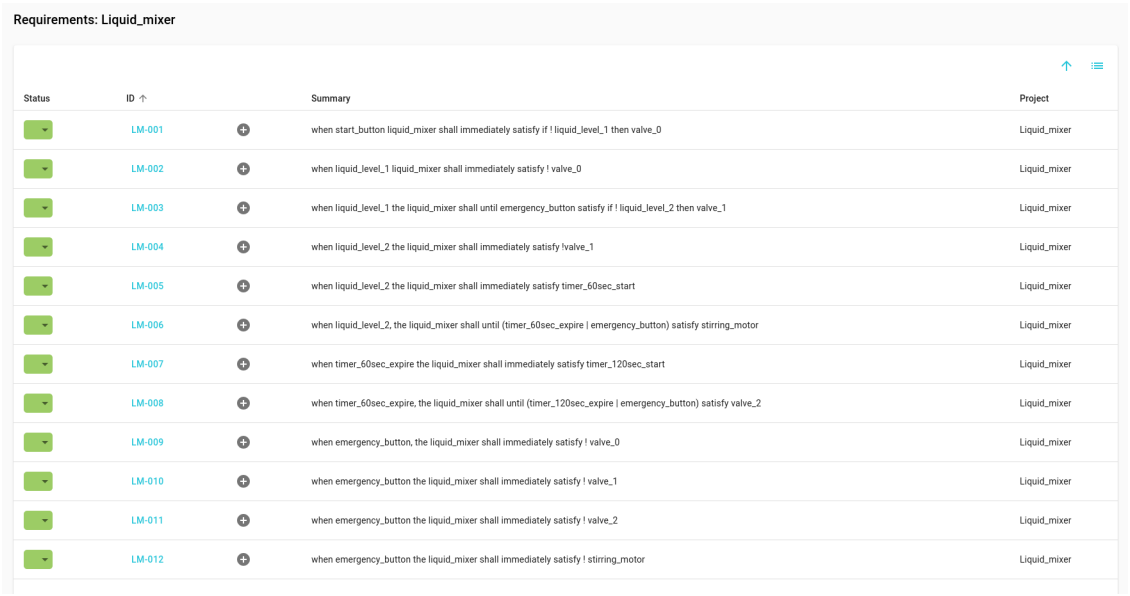


Figure 7.1: The Liquid Mixer requirements as they appear in FRET

7.1 Application to the Liquid Mixer example

In this section, we use the Liquid Mixer example to provide a pictorial demonstration of the process of analyzing and diagnosing requirements with regards to their (un)realizability. Figure 7.1 depicts the twelve requirements as they were written and saved within FRET. Figure 7.2 shows the realizability checking portal that we created.

Given a specific FRET project, the user chooses a particular system component (in this case, Liquid Mixer). The view is then updated with a list of the project requirements that relate to the chosen component. As soon as a system component is picked, the SCC algorithm immediately is ran over the corresponding requirements, computing thus a list of connected components (CCX tabs in Figure 7.2). As soon as the connected components are computed, each SCC tab is updated to focus on the requirements that are relevant to the SCC, by filtering out irrelevant ones (greyed-out requirements in table of Figure 7.2) the user can choose to run realizability either compositionally (i.e. one check per SCC) or monolithically (i.e. one check over all requirements at once).

Assuming that the user opted for the compositional check, Figure 7.3 shows how the result is being displayed. The system component, and all its connected components are updated with a symbol that designates whether the corresponding set of requirements is realizable or not (4 and 8 symbols in Figure 7.3). As we can see the Liquid Mixer system is deemed as unrealizable, with the third connected component (CC2) being the only unrealizable set of requirements.

For the realizable connected components, the user needs to do nothing more. On the other hand, an unrealizable set of requirements needs to be diagnosed. Figure 7.4 shows how the CC2 tab is chosen, and the diagnosis option becomes available. Clicking the "Diagnose" button calls the diagnosis algorithm over CC2. As soon as the analysis is done, the chord diagram for the connected component is displayed. As mentioned before, the diagram summarizes the number of minimal conflicts and corresponding conflicting requirements. The user can then interact with the diagram, focusing on either requirements or conflicts. Figure 6.1 refers to the particular case for Liquid Mixer where the user clicks on the dia-

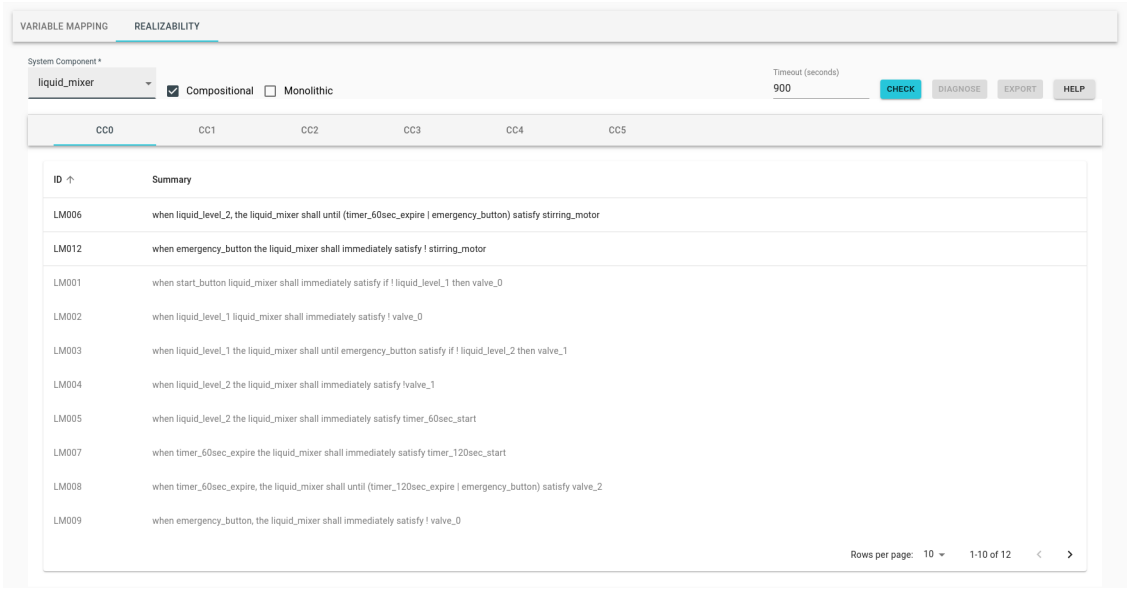


Figure 7.2: The Realizability Checking portal in FRET

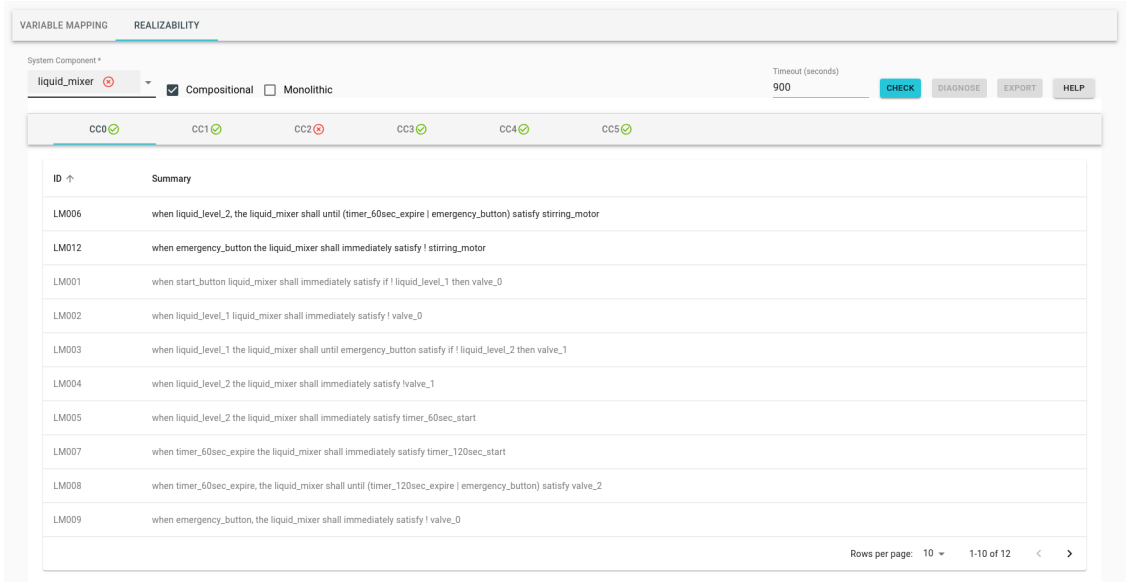


Figure 7.3: The results of running the compositional realizability check over Liquid Mixer

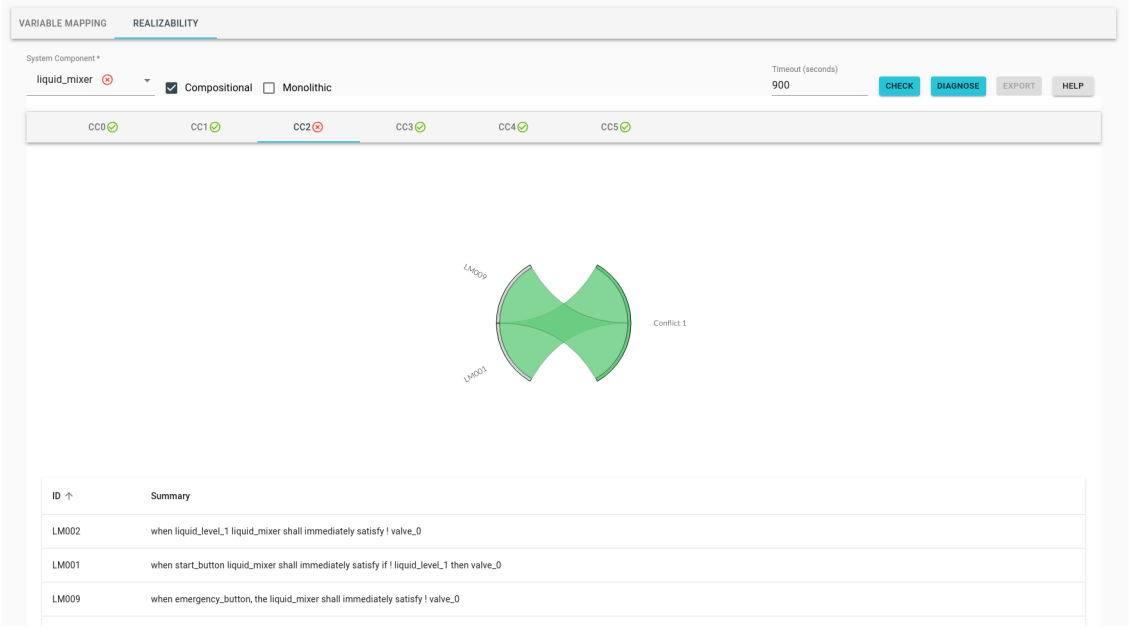


Figure 7.4: Diagnosis result on the unrealizable connected component for Liquid Mixer

gram, and the counterexample appears for the conflict between requirements [LM-001] and [LM-009].

With the minimal explanations of unrealizability at hand, the user is then left to interpret the counterexamples and edit the conflicting requirements as needed. Since the new set of requirements may introduce new connected components, and even other unrealizable fragments, a new analysis cycle needs to be performed.

Chapter 8

Case Studies

To assess the effectiveness of the connected components decomposition and the impact of the diagnosis interface, we relied upon several multi-component, industrial-level projects [1]. 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. In Table 8.1 we present the realizability analysis outcomes as well as performance times on the following case studies: our running example, the Generic Patient Controlled Analgesic (GPCA), the NASA Quad Redundant Flight Control (QFCS), and six sub-challenges of the Lockheed Martin Cyber Physical System Challenges (LMCPS). The experiments were run on an Ubuntu VM, 4.5gb RAM, i5-8365U, 4 cores@1.60 GHz. When a project's requirements are realizable, a diagnosis is not applicable (N/A); also when a project's requirements form a single SCC as in GPCA, the SCC times are the same as the monolithic times. We write N/A for such cases.

8.1 Generic Infusion Pump

The Generic Infusion Pump Research Project [31] is a joint effort by the United States Food and Drug Administration (USFDA), Hutchison China MediTech (Chi-Med), CIMIT [32] 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 [33, 34]. This study used the `Infusion_Manager` subcomponent, previously shown unrealizable by Gacek et al. [10] through the use of the JSyn algorithm. The subcomponent contains 12 requirements.

Formalization. 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 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

[1] Our dataset is available at `https://figshare.com/s/1ae972372dc67afec854`.

Table 8.1: Case studies statistics. The "Monolithic", "Total SCC" and "Diagnosis" columns record the monolithic, compositional (SCC) and diagnosis analysis time in seconds. Total SCC times are denoted by "N/A" if decomposition was not successful. Diagnosis times are denoted by "N/A" for realizable or unknown contracts, or when the SCC time is equal to the monolithic time (no decomposition)

Project	Benchmark	# Reqs	# SCCs	Realizable?		Monolithic		Total SCC		Diagnosis	
				JSyn	JSyn-vg	JSyn	JSyn-vg	JSyn	JSyn-vg	Mono	SCC
Example	liquid_mixer	12	6	8	8	0.50	10.27	2.47	5.85	19.7	1.01
GPCA	Infusion_Manager	26	1	8	8	0.40	10.76	N/A	N/A	343.28	N/A
QFCS	FCC	9	7	4	?	53.31	T/O	4.12	T/O + 6.38	N/A	N/A
QFCS	FCC (inlined)	79	38	4	4	1.11	T/O	13.10	16.41	N/A	N/A
QFCS	OSAS	10	2	8	8	1.82	T/O	2.96	T/O + 1.12	T/O	T/O
QFCS	OSAS (inlined)	190	21	8	8	1.32	T/O	10.83	640.89	T/O	2451.00
LMCPS	AP	13	3	?	?	0.40	T/O	1.42	T/O + 8.68	N/A	N/A
LMCPS	FSM	13	3	8	8	0.42	1524.82	3.51	6.74	T/O	34.60
LMCPS	EB	5	2	?	?	1.41	1.01	1.62	1.39	N/A	N/A
LMCPS	NN	4	1	?	?	55.80	269.87	N/A	N/A	N/A	N/A
LMCPS	REG	10	5	?	4	286.14	99.52	422.52	5.72	N/A	N/A
LMCPS	TSM	6	1	8	4	3.33	242.67	N/A	N/A	N/A	N/A

4: realizable 8: unrealizable ?: unknown T/O: timeout (12 hours)

following requirement in the original contract:

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

This requirement 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[2]. Requirement $\mathbf{G1_1}$ ensures that the system must be 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 additionally introduced requirements to ensure mutual exclusion between modes, something that was not needed with a single mode variable. We used KIND 2 [35] to show equivalence between our requirements and the original specification.

Decomposition. Our FRETish specification formed a single SCC. This was rather expected, as the requirements mainly describe mode logic, and are connected through mode variables. In the future, we plan to study whether large mode-related specifications can be analyzed modularly, potentially by studying the mode-transition logic separately from the intra-mode requirements.

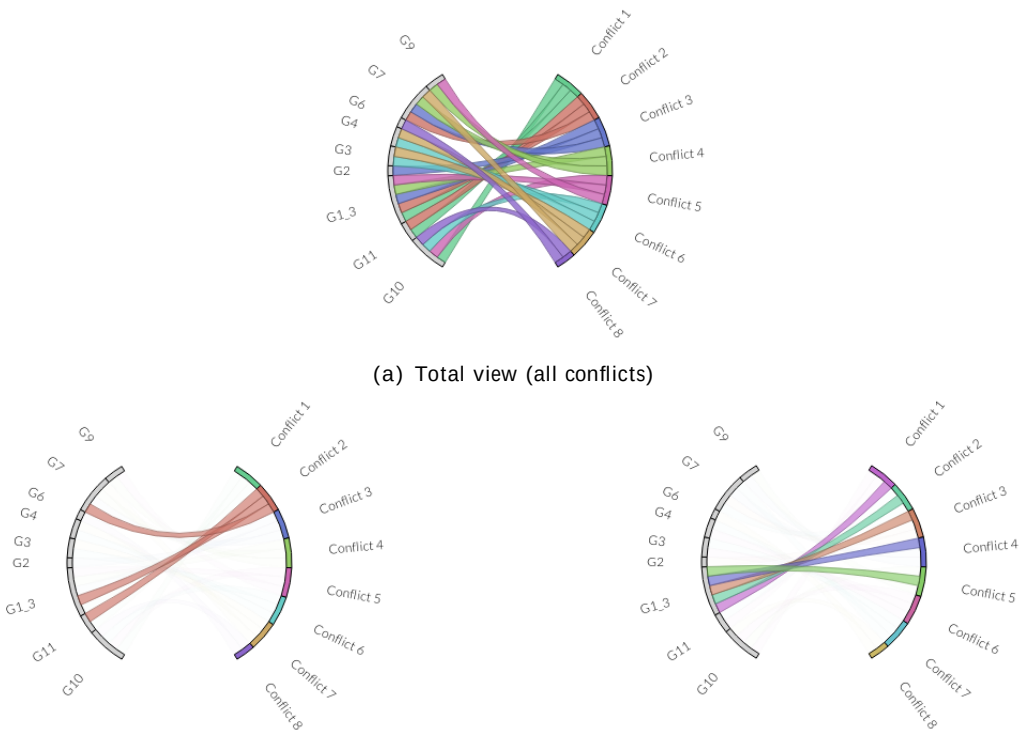
Diagnosis. Gacek et al. [10] manually identified a single pair of conflicting requirements, i.e., G1 and G7, as the source of unrealizability, where G7 is defined as:

$$\begin{aligned} \text{G7} \stackrel{\text{def}}{=} & (\textit{System\_On} \wedge \textit{Highest\_Level\_Alarm} = 3) \Rightarrow \\ & (\textit{Commanded\_Flow\_Rate'} = \textit{Flow\_Rate\_KVO}) \end{aligned}$$

The authors attributed the conflict to the fact that, when *System_On* is true and *Highest_Level_Alarm* = 3, whether *Current_System_Mode* is 0 or 1, the requirements disagree on the value of output *Commanded_Flow_Rate*. However, our diagnostic procedure of Chapter 6 unveiled a different picture. Specifically, it found 8 minimal conflicts between requirements, 7 triplets and one pair (see Figure 8.1a). Both pairs of requirements $\{\mathbf{G1_2}, \mathbf{G7}\}$ and $\{\mathbf{G1_3}, \mathbf{G7}\}$ are in fact realizable. How? By implementing a system that never enters modes 0, or 1, respectively (note that mode variables are state variables). The true source of unrealizability manifests when $\{\mathbf{G1_3}, \mathbf{G7}\}$ are combined with a requirement such as G11, which makes the system enter mode 1, thus activating the conflict:

$$\text{G11} \stackrel{\text{def}}{=} (\textit{System\_On} \wedge \textit{Configured} < 1) \Rightarrow \textit{Current\_System\_Mode'} = 1$$

[2] We discuss the requirements in the original contract notation to make it easy to relate to Gacek et al. [10]



(a) Total view (all conflicts)

(b) Focused view (one conflict)

(c) Focused view (one requirement)

Figure 8.1: Chord Diagram for `Infusion_Manager`

Interestingly, requirement $\{\mathbf{G1_2}\}$ does not contribute to any conflict because no requirement makes the system enter mode 0.

Figure 8.1a depicts FRET's visual explanation of unrealizability in `Infusion_Manager`, using a chord diagram. Since eight conflicts exist in the system, the resulting diagram is dense, reflecting the fact that a lot of requirements in the system participate in a lot of conflicts. Figure 8.1b shows a focused view of the same diagram over the conflict between requirements $\mathbf{G1_3}$, G7 and G11. On the other hand, Figure 8.1c provides a focused view on $\mathbf{G1_3}$, showing how it participates in five different conflicts.

8.2 NASA QFCS

The nested quantifiers in Eq. 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. The Quad-Redundant Flight Control System (QFCS) case study demonstrates how decomposition can effectively reduce problem complexity and lead to significant performance benefits.

QFCS is a component in NASA's Transport Class Model aircraft simulation [36]. 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) [3], both in terms of compositional verification [37] as well as realizability checking and synthesis through `JKind` [10,11]. Compared to the latter work, our proposed approach to compute connected components yields new information regarding the project, that would otherwise be impossible to derive using the monolithic algorithms in `JKind`.

We experimented over the FCC and Output Signal Analysis and Selection (OSAS) contracts, for which prior results already existed for the monolithic approaches. The FCC contract contains nine requirements, yet the resulting check is surprisingly hard, as JSyn required 53.31s to declare it as realizable while JSyn-vg could not solve the same problem, even for a timeout value of twelve hours. Using our connected components algorithm, we decomposed the contract into seven SCCs, six of which contain a single requirement over their corresponding outputs, with the seventh containing three. After partitioning the contract, we 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).

As a last resort towards identifying why FCC-7 was so hard for JSyn-vg to solve, we examined the requirement definitions. Soon enough we found the reason: 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 [3]:

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

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 checked independently from the rest.

As such, we split the initial 9 requirements of FCC into subrequirements. In particular, for each requirement and for each application of a template we derived a subrequirement. The resulting FCC contract 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, $\sim$0.34s per SCC), JSyn-vg was finally able to determine the contract as realizable, requiring in total only 16.41s ($\sim$0.43s per SCC).

For the OSAS contract we used its unrealizable version from Gacek et al. [10], consisting of 10 requirements. The authors had originally attributed the unrealizability of the contract to a pair of conflicting requirements. Contrary to that, our decomposition and diagnosis procedure exposed more issues with the specification.

Decomposing the original contract yielded 2 SCCs, one of which was a singleton. To our surprise, the singleton was unrealizable because the corresponding requirement was mistakenly defined as a guarantee rather than as an assumption (it only referenced input variables). Diagnosis over the non-singleton SCC was initially failing due to timeouts. As previously mentioned, templates are heavily used in OSAS. Deriving new requirements out of templates resulted in a new contract with 190 requirements, and 21 SCCs (the monolithic query contained 3035 variables, versus the 151 per SCC query). The new decomposition led to the following findings:

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

- The assumption $ccdl\_failed \Rightarrow \neg osas\_failed$ proposed by Gacek et al. for requirements OSAS-170 and OSAS-240 was insu□cient to fix the conflict. Instead, we resolved it by strengthening the assumption, requiring that the two system inputs (*ccdl_failed*, *osas_failed*) are mutually exclusive.

- Our fix for {OSAS-170,OSAS-240} was also applicable to another, undisclosed conflict between requirements OSAS-180 and OSAS-240.

- Another conflict existed over requirements OSAS-180 and OSAS-250. Following a similar pattern by strengthening assumptions, we resolved the conflict by requiring that input *act_claw_fails* cannot be true when *ccdl_failed* is true.

- Another singleton conflict was found regarding OSAS-210-220-230. After inspection, we identified a typographical error (a pair of parentheses was closed at the wrong location) which essentially resulted in the requirement being defined as $isas\_fail \Rightarrow false$, where *isas_fail* is an input.

- One final conflict was over requirements OSAS-120, OSAS-140, OSAS-170 and OSAS-190. The problem was related to output *acts_out*, which OSAS-140 required to be 0 when output *acts_fail* is true, while OSAS-190 expected it to be equal to input *acts_in*. The issue is exposed only when all four requirements are considered. We resolved this by changing OSAS-190 to require that $acts\_fail \Rightarrow acts\_out = 0$.

Given the aforementioned fixes, we were able to prove the modified contract realizable by checking each SCC separately. The same could not be replicated monolithically since analysis timed out.

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 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.

8.3 Lockheed Martin CPS Challenges (LMCPS)

LMCPS is a set of industrial Simulink model benchmarks and natural language requirements [38, 39]. 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. [40] perform model testing and checking, while Mavridou et al. [41] 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 [42].

We present results on the following challenges: 1) `6DoF with DeHavilland Beaver Autopilot (AP)`: a realistic simulation of the DeHavilland Beaver airplane with autopilot (13 requirements); 2) `Finite State Machine (FSM)`: an abstraction of an ad-

vanced 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). For brevity, we omit challenges for which our work did not yield new information.

The LMCPS challenges were also used to demonstrate the first prototype for checking realizability of FRET requirements [13]. The results reported here were obtained by re-running the experiments with our new implementation presented in Chapter 7. As mentioned, the new implementation treats requirements that only refer to input variables as assumptions, and includes them as such in the contracts that it analyzes. Additionally, we set longer time outs and used an updated version of the JSyn and JSyn-vg algorithms, as well as the AE-VAL solver, in which some of the issues reported in [13] were fixed.

Assumption handling: Among the LMCPS case studies, only AP contains requirements that are interpreted as assumptions by our implementation. These assumptions appear in all 3 partial contracts that are computed by our algorithms of Chapter 5.

No decomposition: Our decomposition method was not successful for the NN and TSM requirement specifications. It returned a single SCC. The reason behind the unsuccessful decomposition was that the number of requirements for these case studies is relatively small (there are 4 requirements in NN and 6 requirements in TSM) and additionally, these requirements are highly connected through state variables.

Analysis challenges: 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. By decomposing AP we were able to get more meaningful results as explained next.

Successful decomposition: Several of the LMCPS case studies demonstrated how decomposition can effectively reduce problem complexity, and surpass some of the aforementioned challenges.

- 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. Note that these results are the same as obtained in [13]. For the two realizable assume-guarantee contracts this is as expected, since adding assumptions cannot turn a realizable contract into unrealizable. Including assumptions in the third contract did not help obtain a conclusive result.

- 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.

- 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 and diagnose them in a more efficient manner. In fact, diagnosis was performed in a total time of 34.6s. More importantly, decomposing our original specification allowed us to reduce the total realizability time from 1524.82s to less than 7s with JSyn-vg.

Monotonicity of unrealizability: When our analysis did not yield results, we used the result of Theorem 4 to perform realizability analysis at the level of singleton requirements (including assumptions when applicable). The idea was that if we find an unrealizable guarantee, then we can conclude about the unrealizability of the SCC it belongs to. To this end, we performed analysis of singleton requirements for the LMCPS_AP and LMCPS_NN case studies. For AP we analyzed the SCC for which JSyn-vg timed out. This SCC contains 4 requirements, 2 of which were realizable while the other 2 returned 'unknown'. Similarly, we used the JSyn-vg algorithm to analyze independently each of the requirements of the LMCPS_NN case study. JSyn-vg returned 'unrealizable' result for 2 of these requirements. However, as it turned out the unrealizable result was due to a bug in the JSyn-vg algorithm. More specifically, an unsound region of validity was being produced by AE-VAL due to nonlinearities in the requirements, leading to the erroneous result. We have communicated this issue to the AE-VAL developers.

Algorithm trade-offs: As shown in Table 8.1 the JSyn algorithm returned a result in several cases for which JSyn-vg needed more time or even timed out (e.g., in the QFCS_FCC and QFCS_OSAS case studies). Additionally, since the implementation of the JSyn algorithm is an approximation of the algorithm presented in [10], unrealizable JSyn results are not sound. In fact, the analysis of the LMCPS_TSM specification produced such an unsound unrealizable result with JSyn versus the sound (realizable) result of JSyn-vg. By decomposing the original specification we were able to determine sound results with JSyn-vg in cases where the monolithic analysis 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.

8.4 Formula decomposition for reactive synthesis

Specification decomposition has also been studied, independently, in a recent work by Finkbeiner et al. [2], 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.

Finkbeiner et al. propose two approaches towards specification decomposition for reactive synthesis, one at the level of automata, and one at the level of LTL formulas. The latter, which is most relevant to our work, presents a decomposition algorithm for LTL specifications where, given an LTL formula, each conjunct of its CNF equivalent occurs in exactly

one sub-specification. 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, and returns diagnosis results at that level as well. 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, our formula generation algorithms [43] 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. We are currently in contact with the authors of [2] and we would like to try their decomposition approach on cases studies for which our algorithms did not yield results, e.g., `LMCPS EB`, `AP`, and `NN`.

To compare the approaches we used ten benchmarks from the SYNTCOMP 2020 competition [44] [4], for which the decomposition proposed by Finkbeiner et al. yielded exemplary results [2]. The original benchmarks are written in Temporal Logic Synthesis Format (TLSF) [45] and consist of safety and liveness properties. 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 8.2, our decomposition procedure yielded identical results with Finkbeiner et al. for all but the `zoo10` benchmark [5]. 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.

Table 8.2: Comparison with Finkbeiner et al. [2]

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

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

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

Chapter 9

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*. Pnuelli and Rosner were the first to show that the complexity of the problem is double-exponential (2-EXPTIME) for propositional specification [7], while further advancements in the General Reactivity of Rank 1 (GR(1)) fragment of LTL showed that a polynomial time algorithm exists [8, 46]. 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 [47], while others attempt to serve as general requirements analysis and debugging frameworks [48–50]. Our proposed work is particularly relevant with the aforementioned analysis tools, as we attempt to solve the same problem, i.e., to tame the increasing difficulty of information inferral from artifacts produced by the underlying formal mechanisms. The most significant difference is that, at the cost of complexity, we are able to practically operate over a set of safety specifications that may admit not only Boolean constructs, but also infinite arithmetic theories.

To diagnose unrealizability, we rely on the work by Könighofer et al. [16] to compute minimal conflicts. To explain unrealizability, past work mainly focused on the presentation of analysis byproducts, in the form of typical counterexamples, and even counterstrategies [17, 51, 52]. To the best of our knowledge, no previous attempts exist on visualizing unrealizability at a compositional level. We believe that visualization can ease the debugging cycle, especially since the application of formal analyses in practice is usually met with reluctance by engineers with no background in formal methods.

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 that preserve the overall soundness of results [53–55]. The same also applies in the context of realizability checking, and more specifically Boolean synthesis where decomposition techniques have been proposed, taking advantage of Binary Decision Diagrams and And-Inverter Graphs to restructure the original problem into factored formulas [56–58]. In comparison to this work, our proposed algorithm to compute SCCs is orthogonal to them and does not rely on the underlying solvers to perform the decomposition. Furthermore, our approach is not affected by the order in which variables appear in the contract and as such does not require the application of sophisticated ordering heuristics [57].

Akin to realizability checking, prior work exists in other aspects of requirement analysis, such as *rt-inconsistency* [59], *well-separation* [60, 61] and *inherent vacuity* [62, 63]. The compositional approach for rt-inconsistency studied by Simon Roth [64] has similarities

with our SCC approach, but since then research on analysis of rt-inconsistency has studied reduction to program analysis [65]. Note that an unrealizable contract is also rt-inconsistent but not necessarily vice-versa. In the future, we are interested in exploring the applicability of SCCs in the context of the aforementioned requirement analysis techniques.

Chapter 10

Conclusion

We presented a new feature of the FRET tool that supports realizability checking, and diagnosis of unrealizability, of the requirements of a project. Our work improves performance of previous infinite-state realizability checking algorithms, by taking advantage of a specification's modularity over disjoint subsets of requirements. We developed a theory of decomposition for Assume-Guarantee contracts for reactive systems, together with algorithms for performing such decompositions. We also provided a solution that helps requirement engineers localize and visualize unrealizability in their specifications. For a variety of industrial-level models, our realizability analysis yielded significant improvements in performance and ease of diagnosis.

Our framework utilizes raw counterexamples provided by the underlying solvers to depict unrealizability. Even though execution traces to violations can be insightful, in the future we would like to explore additional techniques to make counterexamples easier to digest. We also plan to extend the FRET simulator to replay counterexamples on unrealizable sets of requirements, to further help with diagnosis.

Our theoretical observations and experience in practice inspire several directions for future work. In developing conditions under which checking realizability of a global contract is equivalent to partial contract realizability, we proved intermediate results that can be useful in practice. For example, realizability of non-interfering partial contracts implies realizability of their global contract even when assumptions are partitioned. Moreover, we have proven that unrealizability is monotonic when assumptions are unchanged. For more challenging case studies, we also found it useful to play with different decision procedures and to trade their respective advantages in soundness and performance in order to achieve conclusive results.

In reality, realizability checking remains a challenging task, and one needs to throw as much ingenuity as possible in addressing the problem. In the future we plan to explore ways of combining our observations into building heuristics that may help detect realizability or unrealizability in a more efficient manner.

Bibliography

1. Lúcio, L.; Rahman, S.; Cheng, C.-H.; and Mavin, A.: Just formal enough? automated analysis of EARS requirements. *NASA Formal Methods Symposium*, Springer, 2017, pp. 427–434.

2. Finkbeiner, B.; Geier, G.; and Passing, N.: Specification Decomposition for Reactive Synthesis. *arXiv preprint arXiv:2103.08459*, 2021. A version to be published in NASA Formal Methods Symposium (NFM 2021), May 26-28, 2021.

3. Cofer, D.; Gacek, A.; Miller, S.; Whalen, M. W.; LaValley, B.; and Sha, L.: Compositional verification of architectural models. *NASA Formal Methods Symposium*, Springer, 2012, pp. 126–140.

4. Damm, W.; Hungar, H.; Josko, B.; Peikenkamp, T.; and Stierand, I.: Using contract-based component specifications for virtual integration testing and architecture design. *2011 Design, Automation & Test in Europe*, IEEE, 2011, pp. 1–6.

5. Stachtiari, E.; Mavridou, A.; Katsaros, P.; Bliudze, S.; and Sifakis, J.: Early validation of system requirements and design through correctness-by-construction. *Journal of Systems and Software*, vol. 145, 2018, pp. 52–78.

6. 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.

7. Pnueli, A.; and Rosner, R.: On the synthesis of a reactive module. *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM, 1989, pp. 179–190.

8. Piterman, N.; Pnueli, A.; and Sa'ar, Y.: Synthesis of Reactive(1) Designs. *VMCAI*, Springer, vol. 3855 of *LNCS*, 2006, pp. 364–380.

9. Firman, E.; Maoz, S.; and Ringert, J. O.: Performance heuristics for GR (1) synthesis and related algorithms. *Acta informatica*, vol. 57, no. 1, 2020, pp. 37–79.

10. Gacek, A.; Katis, A.; Whalen, M. W.; Backes, J.; and Cofer, D.: Towards Realizability Checking of Contracts Using Theories. *NFM*, Springer, vol. 9058 of *LNCS*, 2015, pp. 173–187.

11. Katis, A.; Fedyukovich, G.; Guo, H.; Gacek, A.; Backes, J.; Gurfinkel, A.; and Whalen, M. W.: Validity-guided synthesis of reactive systems from assume-guarantee contracts. *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2018, pp. 176–193.

12. Giannakopoulou, D.; Pressburger, T.; Mavridou, A.; Rhein, J.; Schumann, J.; and Shi, N.: Formal Requirements Elicitation with FRET. *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-WS.org, vol. 2584 of *CEUR Workshop Proceedings*, 2020. URL http://ceur-ws.org/Vol-2584/PT-paper4.pdf.

13. Kooi, D.; and Mavridou, A.: Integrating Realizability Checking in FRET. *NASA Technical Memorandum*, June 2019. URL https://ntrs.nasa.gov/api/citations/20190033980/downloads/20190033980.pdf, 28 pages.

14. Gacek, A.: JKind – An infinite-state model checker for safety properties in Lustre. http://loonwerks.com/tools/jkind.html, 2016.

15. Katis, A.; Gacek, A.; and Whalen, M. W.: Towards synthesis from assume-guarantee contracts involving infinite theories: a preliminary report. *4th Intl. Conf. on Formal Methods in Software Engineering (FormaliSE)*, IEEE, 2016, pp. 36–41.

16. Könighofer, R.; Ho□erek, G.; and Bloem, R.: Debugging unrealizable specifications with model-based diagnosis. *Haifa Verification Conference*, Springer, 2010, pp. 29–45.

17. Könighofer, R.; Ho□erek, G.; and Bloem, R.: Debugging formal specifications: a practical approach using model-based diagnosis and counterstrategies. *International Journal on Software Tools for Technology Transfer*, vol. 15, no. 5-6, 2013, pp. 563–583.

18. Champion, A.; Gurfinkel, A.; Kahsai, T.; and Tinelli, C.: CoCoSpec: A mode-aware contract language for reactive systems. *International Conference on Software Engineering and Formal Methods*, Springer, 2016, pp. 347–366.

19. Mavridou, A.; Bourbouh, H.; Garoche, P.-L.; Giannakopoulou, D.; Pressburger, T.; and Schumann, J.: Bridging the Gap Between Requirements and Simulink Model Analysis. *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-WS.org, vol. 2584 of *CEUR Workshop Proceedings*, 2020. URL http://ceur-ws.org/Vol-2584/PT-paper9.pdf.

20. Hopcroft, J.; and Tarjan, R.: Algorithm 447: e□cient algorithms for graph manipulation. *Communications of the ACM*, vol. 16, no. 6, 1973, pp. 372–378.

21. Skiena, S. S.: *The algorithm design manual: Text*, vol. 1. Springer Science & Business Media, 1998.

22. Reiter, R.: A theory of diagnosis from first principles. *Artificial Intelligence*, vol. 32, no. 1, 1987, pp. 57–95.

23. Zeller, A.; and Hildebrandt, R.: Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, vol. 28, no. 2, 2002, pp. 183–200.

24. Chord Diagram. https://www.data-to-viz.com/graph/chord.html.

25. Holten, D.: Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE Transactions on visualization and computer graphics*, vol. 12, no. 5, 2006, pp. 741–748.

26. Mavridou, A.; Bourbouh, H.; Garoche, P.; Giannakopoulou, D.; Pressburger, T.; and Schumann, J.: Bridging the Gap Between Requirements and Simulink Model Analysis. *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*, M. Sabetzadeh, A. Vogelsang, S. Abualhaija, M. Borg, F. Dalpiaz, M. Daneva, N. Condori-Fernández, X. Franch, D. Fucci, V. Gervasi, E. C. Groen, R. S. S. Guizzardi, A. Herrmann, J. Horko□, L. Mich, A. Perini, and A. Susi, eds., CEUR-WS.org, vol. 2584 of *CEUR Workshop Proceedings*, 2020. URL http://ceur-ws.org/Vol-2584/PT-paper9.pdf.

27. Material-UI : React components for faster and easier web development (version 4). https://material-ui.com/.

28. Data-Driven Documents : D3.js. https://d3js.org/.

29. De Moura, L.; and Bjørner, N.: Z3: An e□cient SMT solver. *TACAS*, Springer, 2008, pp. 337–340.

30. Fedyukovich, G.; Gurfinkel, A.; and Sharygina, N.: Automated Discovery of Simulation Between Programs. *LPAR*, Springer, vol. 9450 of *LNCS*, 2015, pp. 606–621.

31. Generic Infusion Pump Research Project. https://rtg.cis.upenn.edu/gip/.

32. Consortia for Improving Medicine within Innovation and Technology. https://cimit.org/home.

33. Murugesan, A.; Whalen, M. W.; Rayadurgam, S.; and Heimdahl, M. P.: Compositional Verification of a Medical Device System. *ACM Int'l Conf. on High Integrity Language Technology (HILT) 2013*, ACM, November 2013.

34. Murugesan, A.; Sokolsky, O.; Rayadurgam, S.; Whalen, M.; Heimdahl, M.; and Lee, I.: Linking Abstract Analysis to Concrete Design: A Hierarchical Approach to Verify Medical CPS Safety. *Proceedings of ICCPS'14*, April 2014.

35. Champion, A.; Mebsout, A.; Sticksel, C.; and Tinelli, C.: The Kind 2 model checker. *International Conference on Computer Aided Verification*, Springer, 2016, pp. 510–517.

36. Hueschen, R. M.: *Development of the Transport Class Model (TCM) aircraft simulation from a sub-scale Generic Transport Model (GTM) simulation*. National Aeronautics and Space Administration, Langley Research Center, 2011.

37. Backes, J.; Cofer, D.; Miller, S.; and Whalen, M. W.: Requirements Analysis of a Quad-Redundant Flight Control System. *NASA Formal Methods*, K. Havelund, G. Holzmann, and R. Joshi, eds., Springer International Publishing, vol. 9058 of *Lecture Notes in Computer Science*, 2015, pp. 82–96. URL http://dx.doi.org/10.1007/978-3-319-17524-9_7.

38. 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. *Safe & Secure Systems and Software Symposium (S5)*, A. F. R. Laboratory, ed., 2015.

39. Elliott, C.: An Example Set of Cyber-Physical V&V Challenges for S5, Lockheed Martin Skunk Works. *Safe & Secure Systems and Software Symposium (S5)*, A. F. R. Laboratory, ed., 2016.

40. Nejati, S.; Gaaloul, K.; Menghi, C.; Briand, L. C.; Foster, S.; and Wolfe, D.: Evaluating model testing and model checking for finding requirements violations in Simulink models. *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 1015–1025.

41. Mavridou, A.; Bourbouh, H.; Giannakopoulou, D.; Pressburger, T.; Hejase, M.; Garoche, P.-L.; and Schumann, J.: The Ten Lockheed Martin Cyber-Physical Challenges: Formalized, Analyzed, and Explained. *Proceedings of the 2020 28th IEEE International Requirements Engineering Conference*, 2020.

42. Mavridou, A.; Bourbouh, H.; Garoche, P.-L.; and Hejase, M.: Evaluation of the FRET and CoCoSim tools on the Ten Lockheed Martin Cyber-Physical Challenge Problems. , NASA, Oct 2019. URL `https://drive.google.com/open?id=1DhEk-A0PgDBU9nU1Jhlb-UFYC8Af71W2`, 84 pages.

43. Giannakopoulou, D.; Pressburger, T.; Mavridou, A.; and Schumann, J.: Automated formalization of structured natural language requirements. *Information and Software Technology*, 2021, p. 106590. URL `https://www.sciencedirect.com/science/article/pii/S0950584921000707`.

44. 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*, vol. 19, no. 3, 2017, pp. 367–390.

45. Jacobs, S.; Klein, F.; and Schirmer, S.: A high-level LTL synthesis format: TLSF v1. 1. *arXiv preprint arXiv:1604.02284*, 2016.

46. Bloem, R.; Jobstmann, B.; Piterman, N.; Pnueli, A.; and Sa'ar, Y.: Synthesis of reactive (1) designs. *Journal of Computer and System Sciences*, vol. 78, no. 3, 2012, pp. 911–938.

47. Ryzhyk, L.; Chubb, P.; Kuz, I.; Le Sueur, E.; and Heiser, G.: Automatic device driver synthesis with Termite. *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ACM, 2009, pp. 73–86.

48. Bloem, R.; Cimatti, A.; Greimel, K.; Ho□erek, G.; Könighofer, R.; Roveri, M.; Schuppan, V.; and Seeber, R.: RATSY–a new requirements analysis tool with synthesis. *International Conference on Computer Aided Verification*, Springer, 2010, pp. 425–429.

49. Maoz, S.; and Ringert, J. O.: Spectra: a specification language for reactive systems. *arXiv preprint arXiv:1904.06668*, 2019.

50. Ehlers, R.; and Raman, V.: Slugs: Extensible GR (1) synthesis. *International Conference on Computer Aided Verification*, Springer, 2016, pp. 333–339.

51. Kuvent, A.; Maoz, S.; and Ringert, J. O.: A symbolic justice violations transition system for unrealizable GR (1) specifications. *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 362–372.

52. Maoz, S.; and Sa'ar, Y.: Counter play-out: executing unrealizable scenario-based specifications. *2013 35th International Conference on Software Engineering (ICSE)*, IEEE, 2013, pp. 242–251.

53. Burch, J. R.; Clarke, E. M.; and Long, D. E.: Representing Circuits More E□ciently in Symbolic Model Checking. *Proceedings of the 28th ACM/IEEE Design Automation Conference*, Association for Computing Machinery, New York, NY, USA, 1991, pp. 403–407. URL https://doi.org/10.1145/127601.127702.

54. Geist, D.; and Beer, I.: E□cient model checking by automated ordering of transition relation partitions. *International Conference on Computer Aided Verification*, Springer, 1994, pp. 299–310.

55. Pan, G.; and Vardi, M. Y.: Symbolic techniques in satisfiability solving. *SAT 2005*, Springer, 2005, pp. 25–50.

56. John, A. K.; Shah, S.; Chakraborty, S.; Trivedi, A.; and Akshay, S.: Skolem functions for factored formulas. *2015 Formal Methods in Computer-Aided Design (FMCAD)*, IEEE, 2015, pp. 73–80.

57. Tabajara, L. M.; and Vardi, M. Y.: Factored Boolean functional synthesis. *2017 Formal Methods in Computer Aided Design (FMCAD)*, IEEE, 2017, pp. 124–131.

58. Chakraborty, S.; Fried, D.; Tabajara, L. M.; and Vardi, M. Y.: Functional synthesis via input-output separation. *2018 Formal Methods in Computer Aided Design (FMCAD)*, IEEE, 2018, pp. 1–9.

59. Post, A.; Hoenicke, J.; and Podelski, A.: rt-Inconsistency: A New Property for Real-Time Requirements. *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*, D. Giannakopoulou and F. Orejas, eds., Springer, vol. 6603 of *Lecture Notes in Computer Science*, 2011, pp. 34–49. URL https://doi.org/10.1007/978-3-642-19811-3_4.

60. Maoz, S.; and Ringert, J. O.: On well-separation of GR (1) specifications. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 362–372.

61. Klein, U.; and Pnueli, A.: Revisiting synthesis of GR (1) specifications. *Haifa Verification Conference*, Springer, 2010, pp. 161–181.

62. Maoz, S.; and Shalom, R.: Inherent vacuity for GR (1) specifications. *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 99–110.

63. Fisman, D.; Kupferman, O.; Sheinvald-Faragy, S.; and Vardi, M. Y.: A framework for inherent vacuity. *Haifa Verification Conference*, Springer, 2008, pp. 7–22.

64. Roth, S.: Erweiterte Konsistenzanalyse für Anforderune (Checking Extended Consistency for Requirements). Master's Thesis, Karlsruhe Institute of Technology, 2011. See Section 3.2.

65. Langenfeld, V.; Dietsch, D.; Westphal, B.; Hoenicke, J.; and Post, A.: Scalable Analysis of Real-Time Requirements. *2019 IEEE 27th International Requirements Engineering Conference (RE)*, 2019, pp. 234–244.

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704–0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
01-03-2021	Technical Memorandum	

4. TITLE AND SUBTITLE
Compositional Realizability Checking within FRET

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Dimitra Giannakopoulou, Andreas Katis, Anastasia Mavridou, Thomas Pressburger

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NASA Ames Research Center, Mo□ett Field, CA 94035

8. PERFORMING ORGANIZATION REPORT NUMBER
L–

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSOR/MONITOR'S ACRONYM(S)
NASA

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
NASA/TM–2021–20210013008

12. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified-Unlimited
Subject Category 59
Availability: NASA STI Program (757) 864-9658

13. SUPPLEMENTARY NOTES
An electronic version can be found at http://ntrs.nasa.gov.

14. ABSTRACT
An Assume-Guarantee contract for a reactive system is realizable if, for any sequence of inputs that satisfy the assumptions on the environment, the guarantees always hold. Realizability checking is essential to ensure that an implementation can be constructed that satisfies the contract. We propose a framework that supports users in the non-trivial task of developing realizable contracts. Our framework uses architectural information to automatically decompose a contract into sub-contracts that can be analyzed separately, and therefore more e□ciently. It then integrates existing algorithms in order to detect unrealizability of (sub)contracts, and to attribute unrealizability to subsets of conflicting guarantees. The latter is key for localizing and correcting the sources of unrealizability. Our approach supports this process by enabling users to interactively visualize and explore the produced conflict sets and counterexamples. We have implemented our framework in the open-source Formal Requirements Elicitation Tool (FRET), and have used it on a variety of industrial-level case studies, showcasing the strengths of our approach in terms of raw performance, as well as diagnostic potential.

15. SUBJECT TERMS
realizability checking, FRET, requirement analysis, compositional

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			STI Information Desk (email: help@sti.nasa.gov)
U	U	U	UU		19b. TELEPHONE NUMBER *(Include area code)* (757) 864-9658