Capture, Analyze, Diagnose: Realizability Checking Of Requirements in FRET

Andreas Katis[1(✉)], Anastasia Mavridou[1], Dimitra Giannakopoulou[2], Thomas Pressburger[2(✉)], and Johann Schumann[1]

[1] Employed by KBR; NASA Ames Research Center, Moffett Field, CA, USA
andreas.katis@nasa.gov
[2] NASA Ames Research Center, Moffett Field, CA, USA
tom.pressburger@nasa.gov

Abstract. Requirements formalization has become increasingly popular in industrial settings as an effort to disambiguate designs and optimize development time and costs for critical system components. Formal requirements elicitation also enables the employment of analysis tools to prove important properties, such as consistency and realizability. In this paper, we present the realizability analysis framework that we developed as part of the Formal Requirements Elicitation Tool (FRET). Our framework prioritizes usability, and employs state-of-the-art analysis algorithms that support infinite theories. We demonstrate the workflow for realizability checking, showcase the diagnosis process that supports visualization of conflicts between requirements and simulation of counterexamples, and discuss results from industrial-level case studies.

1 Introduction

Requirements elicitation is a proactive process which, by capturing the intended behavior of a system at an early stage, safeguards against decisions that could lead to increased development costs and even catastrophic failures. Formal requirements analysis can solidify engineers' confidence in the expressed specification. Our work is concerned with ensuring requirements consistency for system components, as a pre-requisite for subsequent system-level analysis. In particular, we focus on the notion of *realizability*: a realizable set of requirements guarantees that an implementation exists, such that it always behaves in a manner consistent with the specification, no matter what input it receives from its environment. The notion of realizability, first described as implementability by Pnueli and Rosner [47], has since then shaped an entire research area over the specification and synthesis of *reactive systems*.

This paper presents the realizability analysis framework that we have developed as part of NASA's open source tool FRET [3] for writing, understanding, and formalizing requirements. FRET is designed with a strong focus on usability, and is used by several NASA projects to explore the benefits of writing

Dimitra Giannakopoulou contributed to this work prior to joining AWS.

https://doi.org/10.1007/978-3-031-13188-2_24

requirements that can be processed by formal analysis tools [10,17,42,45]. Additionally, FRET has been used by external (to NASA) industrial and research teams, e.g., for the formalization of aircraft engine controller requirements [19]. FRET's realizability framework has two main goals: 1) to implement efficient algorithms for checking realizability, and 2) to provide user support in understanding and correcting sources of unrealizability. With these features, FRET provides an end-to-end solution to capturing, analyzing, and diagnosing requirements.

FRET's realizability framework provides a user-friendly interface for analyzing the requirements of system components. We have designed a graphic environment, in which the user can observe a (potentially) decomposed version of the specification that is sound with respect to realizability, as well as further dive into the task of *diagnosing* unrealizable requirements. Compositional analysis is based on our theoretical framework for checking realizability of a global specification through smaller, more tractable parts [25,43]. The diagnosis process is based on the theoretical work by Könighofer et al. [33,34] on generating minimal conflicts of unrealizability. We adjusted the diagnosis algorithm to support the discovery of all minimal conflicts in a contract, accompanied by a counterexample of unrealizability. The computed artifacts can be visualized as an interactive diagram that depicts the dependencies between requirements and conflicts. Counterexample traces that originate from these conflicts can also be simulated to enhance the understanding of unrealizability sources. For the analysis, we have integrated in FRET state-of-the-art tools with respect to realizability checking modulo infinite theories.

In particular, the contributions of this work are:

- The design and implementation of a realizability checking framework in FRET that tightly integrates the JKind [23] and Kind 2 [35] analysis tools;
- a diagnosis feature for unrealizability that returns all minimal conflicts and their counterexamples in an easy-to-use, graphical user interface;
- the extension of the simulator component in FRET, to be used for the simulation of conflicting requirements in unrealizable specifications; and
- improvements of the algorithms in our in-house fork of the JKind model checker, following recent work from the Kind 2 and GenSys [48] tools.

2 Related Work

Table 1 provides a comparison between prominent requirements specification tools that support realizability checking with respect to various aspects, such as support for liveness properties, specification decomposition, algorithms.

Spectra Tools [37] and RATSY [8] are requirements specification tools for reactive synthesis over the General Reactivity of Rank 1 (GR(1)) fragment of LTL. The GR(1) fragment is particularly appealing, because it subsumes a subset of requirements that may appear in real world problems, adheres to the popular Assume-Guarantee paradigm, and a polynomial-time synthesis algorithm exists for it [9,46]. Both tools are limited to finite-state problems, and provide the ability to diagnose unrealizable specifications, primarily through the computation of minimal unrealizable cores [33,40] and counterstrategy synthesis, where an

Table 1. Comparison of requirements specification tools w.r.t. realizability checking.

Tool	Finite State	Infinite State	Decomposition	Liveness	Unrealizable Cores	Algorithms	Backend	Other features
Spectra	✔	✘	✘	✔	✔	BDD-based fixpoint	CUDD + JTLV	Well-separation, Vacuity Checking, Counterstrategies
SpeAR	✔	✔	✘	✘	✘	k-induction	JKind	N/A
AGREE	✔	✔	✘	✘	✘	k-induction	JKind	N/A
RATSY	✔	✘	✘	✔	✔	BDD-based fixpoint	CUDD+ NuSMV	Counterstrategies
EARS-CTRL	✔	✘	✘	✘	✘	BDD-based fixpoint	autoCode4	N/A
FRET	✔	✔	✔	✘	✔	k-induction, SMT-based fixpoint	JKind, Kind 2	Simulation of conflicting requirements

implementation for the environment is generated, such that its actions always lead to the violation of the specification [34,39]. Furthermore, Spectra Tools provide the ability to repair unrealizable specifications [38].

SpeAR [22] and AGREE [14] are tools developed at Collins Aerospace for the purpose of requirements specification and analysis. Realizability checking is provided as a feature in both tools with limited support. Both tools depend on JKind's k-induction algorithm for realizability checking, which supports infinite-state problems, but is not sound with respect to unrealizable results [24].

EARS-CTRL [36] is yet another requirements specification platform that enables analysis of requirements written in Easy Approach to Requirements Syntax (EARS) [41]. Its realizability checking implementation relies upon autoCode4 [13], and is limited to the GXW subset of LTL [12]. Similar to Spectra Tools and RATSY, its analysis is limited to finite-state problems.

FRET's realizability-checking framework encapsulates desirable features of the aforementioned tools into an interface that is designed for users of varying backgrounds in formal methods. Additionally, it is the only requirements specification tool that provides a powerful decomposition approach to help with analysis performance [25,43]. FRET's realizability framework is powered by the algorithms in JKind and Kind 2. As such, it can analyze requirements that are as expressive as arbitrary discrete past-time metric LTL (pmLTL) formulas, and which may involve arithmetic expressions over the Linear Integer and Real Arithmetic SMT-LIB logics [7]. In practice, the framework targets analysis of formulas corresponding to requirements written in FRETish, as presented in the next section. FRETish requirements correspond to templates that form only a subset of all pmLTL formulas. As long as future FRETish extensions can be translated into pmLTL, analysis will be supported by the realizability backend.

3 The FRETish Language

In FRET, requirements are written in a restricted natural language called FRETish [27]. FRET formalizes FRETish requirements in pmLTL and then

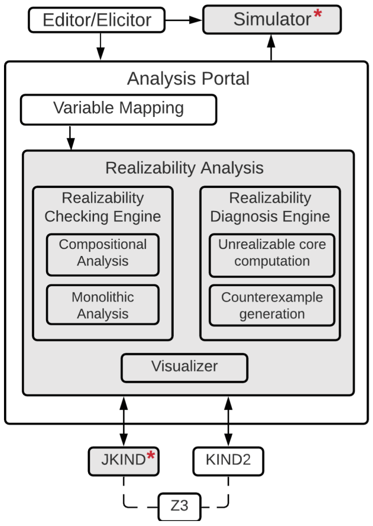
Table 2. Two FSM requirements in FRETISH and pmLTL from Katis et al. [32].

[FSM-006]	FSM `shall` for 5 ticks satisfy (state = 2 & standby & good) => STATE = 3 `H ((O[<=5] (! (Y TRUE))) -> (state = 2 & standby & good) -> STATE = 3`
[FSM-007]	FSM `shall` within 5 ticks satisfy (state = 2 & supported & good) => STATE = 0 `H ((H (! (state = 2 & supported & good) -> STATE = 0)) -> (O[<5] (! (Y TRUE))))`

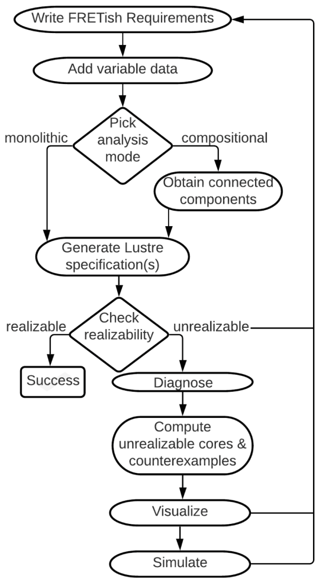
into Lustre. A FRETISH requirement is described using up to six distinct fields (the * symbol designates mandatory fields): 1) `scope` specifies the time intervals where the requirement is enforced, 2) `condition` is a Boolean expression that triggers the `response` to occur at the time the expression's value becomes true, or is true at the beginning of the scope interval, 3) `component*` is the system component that the requirement is levied upon, 4) `shall*` is used to express that the component's behavior must conform to the requirement, 5) `timing` specifies when the response shall happen, subject to the constraints defined in `scope` and `condition` and 6) `response*` is the Boolean expression that the component's behavior must satisfy.

FRETISH provides 8 scopes: *global*, *in*, *before*, *after*, *notin*, *only in*, *only before*, and *only after*. The scope *global* means *always*; the others are with respect to when the system is in a mode or satisfies a Boolean expression. For example, *In mode M* means the requirement is enforced when the system is in mode M, as determined by the Boolean variable M. Also allowed for scope in place of a single Boolean variable is a Boolean expression, except for *in* which in the expression case is written with *while*; e.g., *While vehicle_mode = hover*. In FRETISH, the optional condition field is introduced by the words *upon*, *when*, or *if*, which are synonymous in FRETISH, or the word *unless*, which is the same as *when !*. FRETISH provides 10 timings: *immediately*, *at the next timepoint*, *always*, *eventually*, *never*, *for* N time steps, *within* N time steps, *after* N time steps, *until* *bool_expr*, and *before* *bool_expr*. When the scope is omitted it is taken as *global*; when the condition is omitted, it is taken as `true`; when the timing is omitted, it is taken as *eventually*. If we consider the condition being omitted as a separate case, there are $8 \times 2 \times 10 = 160$ possible combinations of $\langle$*scope*, *condition*, *timing*$\rangle$, each formalized as a distinct pmLTL formula template. The templates are generated by an algorithm that has been formally proven to generate formalizations with the intended semantics [15].

Boolean expressions can use the standard logical connectives (!, &, |) and can involve arithmetic relations (`=`,`!=`,`<`,`<=`,`>`,`>=`) and operators ($+,-,*,/$) over integer and real variables. There are two predefined predicates *preInt* and *preReal* that refer to previous values: the expression $preInt(init, n)$, for integer expression n, returns the value of n at the previous timepoint; if at the beginning of the trace where there is no previous value, then the value of *init* is returned. Currently, FRETISH does not allow arbitrary nesting of temporal operators, e.g. "*In mode m, before q the system shall ...*". Timed operators with intermediate bounds are also not currently expressible; e.g., the equivalent of `H[i,j] p`, where `i` $\neq$ `0`.



(a) Architectural view.



(b) Flow of usage view.

Fig. 1. Implementation views for realizability checking in FRET.

For the remainder of the paper we use a running example, namely `Finite State Machine (FSM)`, to demonstrate the various aspects of our framework. FSM contains 13 requirements for an abstracted version of an advanced autopilot system, and is part of the Lockheed-Martin Cyber-Physical Challenge Problems [18,32,42]. The requirements capture safety expectations with regards to the autopilot system's state transitions. Table 2 contains two `FSM` requirements written in FRETISH and their pmLTL formulas, which are generated by FRET.

4 Implementation

Figure 1a shows the architectural components of FRET that communicate with or belong to the *Realizability Analysis* framework.[1] Grayed components illustrate the contributions of this paper. The asterisks in *Simulator* and *JKind* indicate that their existing implementation and features were considerably extended for this work. Arrows show the flow of data between components. All components are implemented in JavaScript using the React, Material-UI and D3 libraries [2,5,6].

FRET requirements are written using the *Editor/Elicitor* component, which also provides semantic explanations in various forms to assist users to clarify subtle semantic issues. The Simulator component provides an interactive visualizer based on graphical signal representation. Given a FRET requirement, it shows

[1] The FRET architecture is described in previous work by Giannakopoulou et al. [26].

temporal traces of each of the variables involved as well as the valuation of the requirement for each point in time. The user can interactively modify the input signals, which results in automatically updating the valuation of the requirement and thus, visually inspecting the temporal behavior of the requirement. As part of this work, we extended the Simulator with the following features: 1) the ability to import and export simulation traces, 2) support for numerical expressions, and 3) simultaneous visualization of multiple requirements. We integrated the Simulator in our realizability analysis workflow, to provide the ability to inspect and interact with counterexample traces in unrealizable specifications.

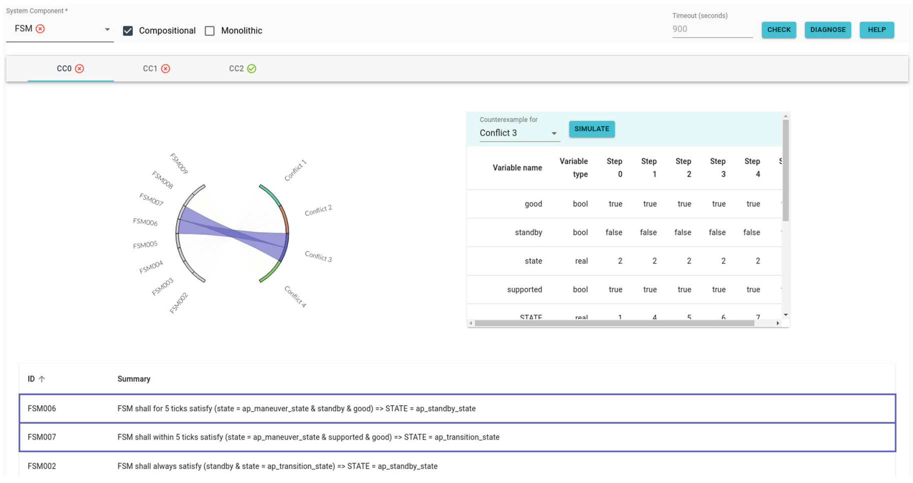
The *Variable Mapping* component collects essential information provided by the user regarding the variables of the requirements, e.g., data types and correspondence to system inputs or outputs. *Realizability Analysis* consists of three sub-components. The *Realizability Checking Engine* is responsible for checking realizability of requirement sets either monolithically or compositionally. Given an unrealizable set of requirements, the *Realizability Diagnosis Engine* implements the algorithm proposed by Könighofer et al. [33,34] to compute all minimal unrealizable sets of requirements, called *minimal unrealizable cores*. For each such core, a counterexample trace is computed that depicts a case under which the environment can lead the system into a deadlocking state. For the computation of minimal conflicts, our implementation uses the *delta-debugging* algorithm [49]. The *Visualizer* implements the user interface that displays analysis results as well as diagnostic results in the case of unrealizable specifications. These results are typically hard to digest in their original form. As such, the visualizer translates the information into an interactive diagram that allows the user to focus on unrealizable cores and inspect or simulate conflicting requirements.

We have integrated into FRET the JKind [23] and Kind 2 [11] tools for checking realizability. We actively maintain a fork of JKind [30], because the original repository lacks an implementation for the fixpoint algorithm by Katis et al. [31]. Formerly, the fork implementation relied on the AE-VAL solver's Model-Based Projection algorithm to perform quantifier elimination over forall-exists formulas [20,21]. As part of this work, we have improved its performance by utilizing Z3's [16] quantifier elimination tactics. For instance, for the analysis of FSM the version of JKind using AE-VAL took 1524.82 s [43], whereas our optimization through Z3 dramatically decreased the time to 0.6 s.

The flow of usage of our framework is as follows (Fig. 1b). Once requirements are written in FRETish and variable information is provided, the user may start the analysis. Realizability can be performed through two different modes: 1) monolithic and 2) compositional, i.e., through the computation of independent sub-specifications, namely connected components. Each connected component is an undirected dependency graph with requirements as vertices and system outputs as edges. Compositional analysis has been proved faster and more prone to return result, compared to the monolithic option [43]. At the next step, the specification is translated to Lustre [29] and fed into JKind and Kind 2 to perform realizability checking. If the specification is unrealizable, the user can diagnose it using the generated counterexamples, and the FRET simulator.

5 Features Walkthrough

We next demonstrate the features of framework through our running example.



Variable name	Variable type	Step 0	Step 1	Step 2	Step 3	Step 4
good	bool	true	true	true	true	true
standby	bool	false	false	false	false	false
state	real	2	2	2	2	2
supported	bool	true	true	true	true	true
STATE	real	1	4	5	6	7

ID ↑	Summary
FSM006	FSM shall for 5 ticks satisfy (state = ap_maneuver_state & standby & good) => STATE = ap_standby_state
FSM007	FSM shall within 5 ticks satisfy (state = ap_maneuver_state & supported & good) => STATE = ap_transition_state
FSM002	FSM shall always satisfy (standby & state = ap_transition_state) => STATE = ap_standby_state

Fig. 2. The realizability checking interface in FRET.

Realizability Checking. Figure 2 provides a snapshot of the overall graphical user interface (GUI) for realizability checking in FRET. As soon as the system component is selected, its connected components (CC) are computed. In the case of `FSM`, three CCs are identified. The GUI provides a focused view for each one (‘CCX’ tabs, with X being the corresponding index value), where the user can see which requirements participate in each CC via a table that dynamically grays out unrelated requirements. As soon as the CCs are computed, the realizability checking options become available, i.e., compositional and monolithic.

To check realizability, the user clicks the ‘Check’ button. Depending on the input specification, four possible answers may be given i.e., the specification is realizable, unrealizable, inconsistent, or the analysis is inconclusive (“unknown” result). Figure 2 shows the results of a compositional check for `FSM`, where connected components CC0 and CC1 are unrealizable, and CC2 is realizable.

Diagnosing Unrealizability. The compositional results above suggest that the `FSM` requirements are, as a whole, unrealizable. The next step in the process is to try and understand the source(s) of unrealizability. Since only CC0 and CC1 are unrealizable, it suffices to diagnose these independently. Following Fig. 2, the user selects the ‘CC0’ tab and clicks the ‘Diagnose’ button. The computation of minimal unrealizable cores kicks in, as outlined in Sect. 4, identifying 4 cores.

Visualizing Unrealizability. The raw artifacts produced by realizability checking and diagnosis are difficult for the users to digest. Therefore, the ability to visualize data in a user-friendly format is necessary, especially for unrealizable specifications. The core of our proposed solution to visualize unrealizability relies on the use of *chord diagrams* [1]. 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* [28], even when the size of data is large.

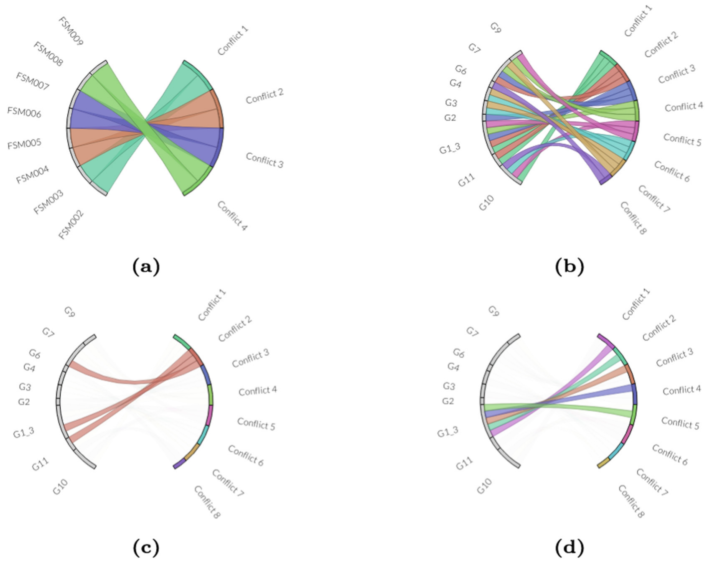


Fig. 3. (a) Chord Diagram for connected component CC0 in `FSM`. (b) Chord Diagram for `Infusion_Manager`. (c) Focused view (one core) for `Infusion_Manager`. (d) Focused view (one requirement) for `Infusion_Manager`.

Figure 3a shows the chord diagram that is generated for connected component CC0 in `FSM`. Requirements and conflicts (i.e., unrealizable cores) 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 interface that are not

Table 3. Counterexample for conflicting requirements **[FSM-006]** and **[FSM-007]**.

Variable name	Variable type	Step 0	Step 1	Step 2	Step 3	Step 4	Step 5
good	bool	true	true	true	true	true	true
standby	bool	false	false	false	false	false	true
state	int	2	2	2	2	2	2
supported	bool	true	true	true	true	true	true
STATE	int	1	4	5	6	7	0
FSM-006	bool	true	true	true	true	true	false
FSM-007	bool	true	true	true	true	true	true

related to the selected element can be filtered out. Figure 2 shows an instance where the user has already interacted with the chord diagram for CC0, focusing on the unrealizable core containing [**FSM-006**] and [**FSM-007**]. The table of requirements is dynamically sorted so that relevant requirements appear on the top, and are outlined with the color of the corresponding conflict. Additionally, a counterexample witnessing the unrealizability of the conflict is displayed. Table 3 shows the counterexample for requirements [**FSM-006**] and [**FSM-007**].

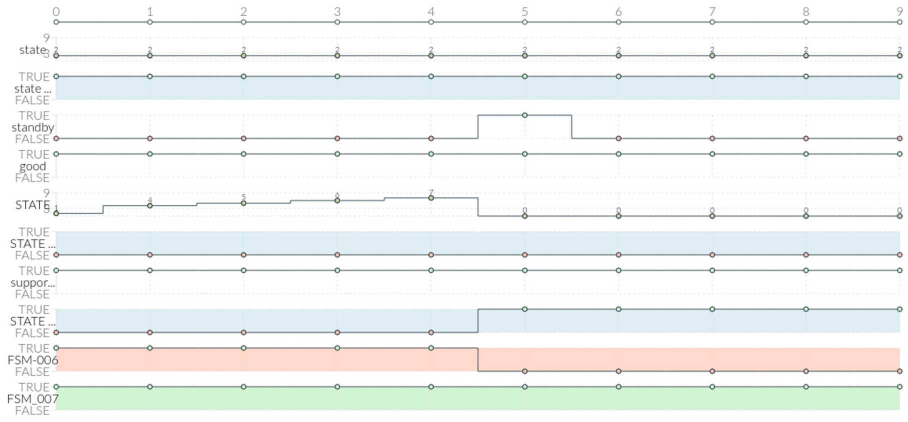


Fig. 4. Simulation of conflicting requirements [**FSM-006**] and [**FSM-007**].

Simulating Conflicting Requirements. Our experience with counterexamples has indicated that a single execution trace is not enough to truly understand interactions between requirements. Therefore, we provide the ability for the user to interact with the set of conflicting requirements by using the FRET simulator, which we have substantially extended to meet our needs in visualizing conflicting requirements. Figure 4 shows how the counterexample (Table 3) for [**FSM-006**] and [**FSM-007**] is displayed in the simulator window: each line shows the values of the input signals as well as the valuation of each of the requirements.

The counterexample in Table 3 is not the only witness to the unrealizability of these requirements. Another example is a trace where requirement [**FSM-006**] holds for 5 consecutive ticks, leading to a violation of requirement [**FSM-007**] at the last tick, assuming that the antecedent of the latter was true at least once within the last 5 ticks. By modifying the values of the input variables, a user may identify additional witnesses to unrealizability causes. Combined with the ability to store and review traces, the simulator makes for an integral element towards understanding and repairing unrealizable specifications.

6 Case Studies

6.1 Lift Plus Cruise Aircraft

This study reports preliminary results on requirements for an autonomous 'lift plus cruise' concept aircraft.[2] This aircraft has a hovering vehicle mode, using its lifting rotors. From the hover mode, it can transition to a flying forward mode, eventually using its rear pusher propeller, and where lift is provided by the wing instead of the lifting rotors. Inbetween the hover and forward modes is a transitional mode which is a phase of concern for the aircraft engineers.

Table 4. FRETISH requirements for Lift Plus Cruise from Katis et al. [32].

[LPC01]	The vehicle `shall` immediately satisfy vehicle_mode = hover `H ((! (Y TRUE)) -> vehicle_mode = hover)`
[LPC02]	While vehicle_mode = hover, the vehicle `shall` never satisfy gndspeed > 20.0 `H ((vehicle_mode = hover) -> (! (gndspeed > 20.0)))`
[LPC03]	While vehicle_mode = hover, the vehicle `shall` eventually satisfy ! rear_propeller `(H (((! (vehicle_mode = hover)) & (Y (vehicle_mode = hover))) -> (Y (! ((! (! rear_propeller)) S ((! (! rear_propeller)) & ((vehicle_mode = hover) & ((! (Y TRUE)) | (Y (! (vehicle_mode = hover)))))))))) & (((! ((! (vehicle_mode = hover)) & (Y (vehicle_mode = hover)))) S ((! ((! (vehicle_mode = hover)) & (Y (vehicle_mode = hover)))) & ((vehicle_mode = hover) & ((! (Y TRUE)) | (Y (! (vehicle_mode = hover))))))) -> (! ((! (! rear_propeller)) S ((! (! rear_propeller)) & ((vehicle_mode = hover) & ((! (Y TRUE)) | (Y (! (vehicle_mode = hover)))))))))`
[LPC04]	The vehicle `shall` always satisfy if (preInt(hover,vehicle_mode) = hover & preReal(0.0,gndspeed) > 15.0) then vehicle_mode = transitional `(H (((preInt(hover,vehicle_mode) = hover) & (preReal(0.0,gndspeed) > 15.0)) -> vehicle_mode = transitional))`
[LPC09]	The vehicle `shall` always satisfy if (preInt(hover,vehicle_mode) = transitional & preReal(0.0,airspeed) > 100.0) then vehicle_mode = forward `(H (((preInt(hover,vehicle_mode) = transitional) & (preReal(0.0,airspeed) > 100.0)) -> (vehicle_mode = forward)))`

As of this paper, 11 requirements have been formalized in FRET [32]. A subset is shown in Table 4, describing the transition relations and constraints among various vehicle modes and vehicle motion. Requirement **[LPC01]** states that the vehicle starts in hover mode. Requirement **[LPC04]** specifies that if the previous mode is hover, and ground speed is greater than 15 knots, then the vehicle enters transitional mode. Requirement **[LPC09]** states the conditions for transitioning to forward mode. Variables `hover`, `transitional` and `forward` are specified as distinct integer constants. All of the other variables, e.g., `airspeed`, `rear_propeller`, are outputs.

[2] We acknowledge discussions with John Kaneshige, Michael Feary and the Revolutionary Vertical Lift Technology team.

The first complete set of FRETISH requirements raised concerns, as realizability checking yielded non-sensical counterexamples, where at least one requirement between [**LPC04**] and [**LPC09**] was violated in the initial state. We quickly identified the issue: both requirements were written using a version of the 'previous' operator `pre` which is undefined at the initial state. We addressed this by introducing the `preInt` and `preReal` operators, which at the initial state return the value of their first argument.

The resulting 11 requirements are in one CC, so we ran analysis in monolithic mode. The requirements are shown to be realizable in about 8 s. As a sanity check for realizability, we experimented with various subsets of the original requirements, as well as adding contradictions. A notable example was omitting [**LPC01**], while modifying [**LPC03**] so that in hover mode, the vehicle must fly faster than 30 knots. This experiment, unexpectedly to us, led to realizability. Further inspection quickly revealed how omitting [**LPC01**] allows the controlled variable `vehicle_mode` to never enter the hover mode. Including [**LPC01**] led to unrealizability with minimal conflict [**LPC01**], [**LPC02**] and [**LPC03**].

6.2 Generic Infusion Pump

This study explores 12 formalized requirements, proven unrealizable by Gacek et al. [24], of the `Infusion_Manager` subcomponent for a Generic Patient Controlled Analgesic (GPCA) infusion pump [44]. The GPCA system originates from the Generic Infusion Pump Research project, a joint effort to identify best software engineering practices in the development of medical devices [4].

Taking advantage of FRETISH's support for system modes (`scope` field), we derived 26 requirements, as opposed to the original 12 [32]. The increased number is a direct product of the declaration of 8 distinct modes, stemming from the system variable *Current_System_Mode*, which was originally of integer type. For example, requirement **G1** from Gacek et al.:

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

was rewritten into three requirements: $\mathbf{G1_1}$ ensures that the system is in at least one of the 8 modes at any time, while requirements $\mathbf{G1_2}$ and $\mathbf{G1_3}$ ensure that the pump's flow rate is equal to 0 when the system is in mode 0 or 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 to show equivalence between our requirements and the original specification.

Gacek et al. had already shown that the `Infusion_Manager` requirements are unrealizable, verbally attributing unrealizability to a conflict between **G1** and requirement **G7**:

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

The authors claimed that the requirements are unrealizable because they disagree on the value of output *Commanded_Flow_Rate* under specific conditions. However, FRET's diagnostic procedure provided a different answer, identifying 8 minimal unrealizable cores. Furthermore, the assumed conflict between requirements **G1** and **G7** does not really exist. While the two requirements do disagree on the value for the system output *Commanded_Flow_Rate* under specific circumstances, a realization still exists: one which would never exercise modes 0 or 1! Nevertheless, the report by Gacek et al. was still on the right track, as part of **G1** (FRETish requirement $\mathbf{G1_3}$) and **G7** participate in at least one minimal unrealizable core with requirement **G11**, the latter enforcing the system to enter mode 1, given specific system input values:

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

Figure 3b shows the chord diagram for `Infusion_Manager`, depicting the 8 minimal unrealizable cores. Figures 3c and 3d show resulting states of the diagram after the user interacted with it in order to focus on a specific core, or a specific requirement, respectively.

7 Conclusion

We presented the realizability analysis framework in FRET and demonstrated its interactive GUI, which helps users diagnose unrealizable specifications through visualizations and simulation of conflicts. The framework employs state-of-the-art analysis algorithms that support infinite theories. In the future, we plan to extend the tool with recommendations in the form of environment assumptions.

References

1. Chord diagram. https://www.data-to-viz.com/graph/chord.html
2. D3.js: Data-driven documents. https://d3js.org/
3. FRET: Formal requirements elicitation tool. https://tinyurl.com/ycxe9fv4
4. Generic infusion pump research project. https://rtg.cis.upenn.edu/gip/
5. Material-UI. https://mui.com/
6. React: a javascript library for building user interfaces. https://reactjs.org/
7. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB) (2016). www.SMT-LIB.org
8. Bloem, R., et al.: RATSY – a new requirements analysis tool with synthesis. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 425–429. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_37
9. Bloem, R., Jobstmann, B., Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive (1) designs. J. Comput. Syst. Sci. **78**(3), 911–938 (2012)
10. Bourbouh, H., et al.: Integrating formal verification and assurance: an inspection rover case study. In: Dutle, A., Moscato, M.M., Titolo, L., Muñoz, C.A., Perez, I. (eds.) NFM 2021. LNCS, vol. 12673, pp. 53–71. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-76384-8_4

11. Champion, A., Mebsout, A., Sticksel, C., Tinelli, C.: The Kind 2 model checker. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 510–517. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_29
12. Cheng, C.-H., Hamza, Y., Ruess, H.: Structural synthesis for GXW specifications. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 95–117. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_6
13. Cheng, C.-H., Lee, E.A., Ruess, H.: autoCode4: structural controller synthesis. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 398–404. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_23
14. Cofer, D., Gacek, A., Miller, S., Whalen, M.W., LaValley, B., Sha, L.: Compositional verification of architectural models. In: Goodloe, A.E., Person, S. (eds.) NFM 2012. LNCS, vol. 7226, pp. 126–140. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28891-3_13
15. Conrad, E., Titolo, L., Giannakopoulou, D., Pressburger, T., Dutle, A.: A compositional proof framework for FRETish requirements. In: Popescu, A., Zdancewic, S. (eds.) CPP 2022, pp. 68–81. ACM (2022). https://doi.org/10.1145/3497775.3503685
16. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
17. Dutle, A., et al.: From requirements to autonomous flight: an overview of the monitoring ICAROUS project. In: Luckuck, M., Farrell, M. (eds.) FMAS 2020. EPTCS, vol. 329, pp. 23–30. Open Publishing Association (2016). https://doi.org/10.4204/EPTCS.329.3
18. Elliott, C.: An example set of cyber-physical V&V challenges for S5, Lockheed Martin Skunk Works. In: Safe & Secure Systems and Software Symposium (S5) 2016, AFRL (2016). http://mys5.org/Proceedings/2016/Day_2/2016-S5-Day2_0945_Elliott.pdf
19. Farrell, M., Luckcuck, M., Sheridan, O., Monahan, R.: FRETting about requirements: formalised requirements for an aircraft engine controller. In: Gervasi, V., Vogelsang, A. (eds.) Requirements Engineering: Foundation for Software Quality. REFSQ 2022. LNCS, vol. 13216. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-98464-9_9
20. Fedyukovich, G., Gurfinkel, A., Gupta, A.: Lazy but effective functional synthesis. In: Enea, C., Piskac, R. (eds.) VMCAI 2019. LNCS, vol. 11388, pp. 92–113. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-11245-5_5
21. Fedyukovich, G., Gurfinkel, A., Sharygina, N.: Automated discovery of simulation between programs. In: Davis, M., Fehnker, A., McIver, A., Voronkov, A. (eds.) LPAR 2015. LNCS, vol. 9450, pp. 606–621. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48899-7_42
22. Fifarek, A.W., Wagner, L.G., Hoffman, J.A., Rodes, B.D., Aiello, M.A., Davis, J.A.: SpeAR v2.0: formalized past LTL specification and analysis of requirements. In: Barrett, C., Davies, M., Kahsai, T. (eds.) NFM 2017. LNCS, vol. 10227, pp. 420–426. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57288-8_30
23. Gacek, A., Backes, J., Whalen, M., Wagner, L., Ghassabani, E.: The JKind model checker. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10982, pp. 20–27. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96142-2_3
24. Gacek, A., Katis, A., Whalen, M.W., Backes, J., Cofer, D.: Towards realizability checking of contracts using theories. In: Havelund, K., Holzmann, G., Joshi, R. (eds.) NFM 2015. LNCS, vol. 9058, pp. 173–187. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-17524-9_13

25. Giannakopoulou, D., Katis, A., Mavridou, A., Pressburger, T.: Compositional Realizability Checking within FRET. NASA Technical Memorandum, March 2021
26. Giannakopoulou, D., Pressburger, T., Mavridou, A., Rhein, J., Schumann, J., Shi, N.: Formal requirements elicitation with FRET. In: Mehrdad Sabetzadeh, M., Vogelsang, A., et al. (eds.) REFSQ 2020. CEUR Workshop Proceedings, vol. 2584 (2020)
27. Giannakopoulou, D., Pressburger, T., Mavridou, A., Schumann, J.: Automated formalization of structured natural language requirements. Inf. Softw. Technol. **137**, 106590 (2021)
28. Holten, D.: Hierarchical edge bundles: visualization of adjacency relations in hierarchical data. IEEE Trans. Visual. Comput. Graph. **12**(5), 741–748 (2006)
29. Jahier, E., Raymond, P., Halbwachs, N.: The Lustre V6 reference manual
30. Katis, A.: JKind fork. https://github.com/andreaskatis/jkind-1
31. Katis, A., et al.: Validity-guided synthesis of reactive systems from assume-guarantee contracts. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10806, pp. 176–193. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89963-3_10
32. Katis, A., Mavridou, A., Giannakopoulou, D., Pressburger, T.: Realizability checking of requirements in FRET. NASA Technical Memorandum, June 2021
33. Könighofer, R., Hofferek, G., Bloem, R.: Debugging unrealizable specifications with model-based diagnosis. In: Barner, S., Harris, I., Kroening, D., Raz, O. (eds.) HVC 2010. LNCS, vol. 6504, pp. 29–45. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19583-9_8
34. Könighofer, R., Hofferek, G., Bloem, R.: Debugging formal specifications: a practical approach using model-based diagnosis and counterstrategies. Int. J. Softw. Tools Technol. Transfer **15**(5–6), 563–583 (2013)
35. Larraz, D., Tinelli, C.: Realizability checking of contracts with Kind 2 (2022). https://doi.org/10.48550/ARXIV.2205.09082
36. Lúcio, L., Rahman, S., Cheng, C.-H., Mavin, A.: Just formal enough? Automated analysis of EARS requirements. In: Barrett, C., Davies, M., Kahsai, T. (eds.) NFM 2017. LNCS, vol. 10227, pp. 427–434. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57288-8_31
37. Maoz, S., Ringert, J.O.: Spectra: a specification language for reactive systems. Softw. Syst. Model. **20**(5), 1553–1586 (2021)
38. Maoz, S., Ringert, J.O., Shalom, R.: Symbolic repairs for GR(1) specifications. In: Atlee, J.M., Bultan, T, Whittle, J. (eds.) ICSE 2019, pp. 1016–1026. IEEE/ACM (2019). https://doi.org/10.1109/ICSE.2019.00106
39. Maoz, S., Sa'ar, Y.: Counter play-out: executing unrealizable scenario-based specifications. In: Notkin, D., Cheng, B.H.C., Pohl, K. (eds.) ICSE 2013, pp. 242–251. IEEE (2013). https://doi.org/10.1109/ICSE.2013.6606570
40. Maoz, S., Shalom, R.: Unrealizable cores for reactive systems specifications. In: ICSE 2021, pp. 25–36. IEEE (2021). https://doi.org/10.1109/ICSE43902.2021.00016
41. Mavin, A., Wilkinson, P., Harwood, A., Novak, M.: Easy approach to requirements syntax (EARS). In: RE (2009)
42. Mavridou, A., et al: The ten Lockheed Martin cyber-physical challenges: formalized, analyzed, and explained. In: RE (2020)
43. Mavridou, A., Katis, A., Giannakopoulou, D., Kooi, D., Pressburger, T., Whalen, M.W.: From partial to global assume-guarantee contracts: compositional realizability analysis in FRET. In: Huisman, M., Păsăreanu, C., Zhan, N. (eds.) FM

2021. LNCS, vol. 13047, pp. 503–523. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-90870-6_27
44. Murugesan, A., Sokolsky, O., Rayadurgam, S., Whalen, M., Heimdahl, M., Lee, I.: Linking abstract analysis to concrete design: a hierarchical approach to verify medical CPS safety. In: ICCPS 2014, pp. 139–150. IEEE (2014). https://doi.org/10.1109/ICCPS.2014.6843718
45. Perez, I., Mavridou, A., Pressburger, T., Goodloe, A., Giannakopoulou, D.: Automated translation of natural language requirements to runtime monitors. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2022. LNCS, vol. 13243. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_21
46. Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive(1) designs. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 364–380. Springer, Heidelberg (2006). https://doi.org/10.1007/11609773_24
47. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: POPL 1989, pp. 179–190. ACM (1989). https://doi.org/10.1145/75277.75293
48. Samuel, S., D'Souza, D., Komondoor, R.: GenSys: a scalable fixed-point engine for maximal controller synthesis over infinite state spaces. In: ESEC/FSE 2021, pp. 1585–1589. ACM (2021). https://doi.org/10.1145/3468264.3473126
49. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. IEEE Trans. Softw. Eng. **28**(2), 183–200 (2002)