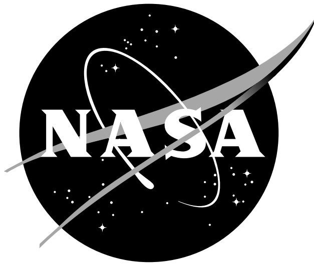
NASA/TM–2019–220365



Integrating Realizability Checking in FRET

David Kooi
University of California Santa Cruz, Santa Cruz, CA 95064, USA

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

June 2019

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 Officer. 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:

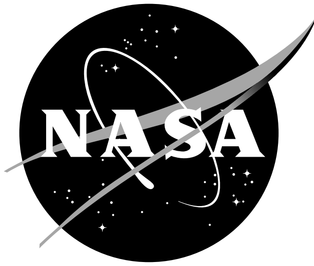
- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference 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–2019–220365



Integrating Realizability Checking in FRET

David Kooi
University of California Santa Cruz, Santa Cruz, CA 95064, USA

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

National Aeronautics and
Space Administration

Ames Research Center, Moffett Field, CA 94035
Onera, The French Aerospace Lab, FR

June 2019

Acknowledgments

I would like to acknowledge all those who guided and assisted me during this summer internship task. In particular Anastasia Mavridou for enthusiastic mentorship and guidance, Tom Pressburger for advice with the connected components and compositional testing, and Dimitra Giannakopoulou for questions, feedback, and advice. Additionally, correspondence with Grigory Fedyukovich was helpful in understanding the AE-VAL solver. Finally, a big thanks to Andreas Katis for continuous and instructive feedback on using JKind's realizability engine.

The use of trademarks or names of manufacturers in this report is for accurate reporting and does not constitute an offical endorsement, either expressed or implied, of such products or manufacturers by the National Aeronautics and Space Administration.

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

Executive Summary

Realizability (alternatively referred to as *relative consistency* or *implementability*), i.e., checking whether a specification can be implemented by an open system, has been the subject of extensive study. Realizability can be viewed as a stronger analysis method when compared to consistency checking in that it does not only check whether the system works in *some* environment, but instead whether the system works in *all* environments. In this report, we experiment and build on the approach presented by Gacek et al. that supports checking realizability of contracts involving infinite theories using SMT solvers. In particular, we 1) extend the Formal Requirements Elicitation Tool (FRET) to support generation of Lustre contracts that can be checked for realizability with the JKind k-induction and fix point generation engines; 2) study a compositional way for checking realizability based on the notion of connected components; and 3) test our approach and the capabilities of the JKind realizability engines on a large subset of the Lockheed Martin Cyber Physical System Challenge Problems.

Contents

1 Preliminaries **1**

2 Realizability Testing with FRET and JKind **2**
2.1 Generation of Lustre Code Through FRET 2
2.2 Realizability Testing Through JKind 3
2.3 Background 3
2.4 Fixpoint 4
2.5 K-Induction 4
2.6 Notes 5

3 Compositional Realizability Testing **6**
3.1 Output Dependency Graphs 6
3.2 Connected Components 7
3.3 Computing The Set of Connected Components 8
3.3.1 Algorithm Complexity 9
3.3.2 Example: Regulator 9
3.3.3 Example: Autopilot 10
3.3.4 Example: FSM 11
3.3.5 Example: Effector Blender 11
3.3.6 Example: Triplex Signal Monitor 12
3.3.7 Example: Tustin Integrator 12
3.3.8 Example: Neural Network 13

4 Conclusion **14**
4.1 Identified Issues 14
4.2 Compositional Testing 15

5 Appendix **16**
A Results 17
A.1 Realizability Statistics 17
A.2 Fixpoint and K-Indution Non-Linear Performance 17
A.3 TSM 17
A.4 TUI 18
A.5 NN 18
A.6 AP 18
B List of Manual Realizability Checking Results 19
B.1 Triplex Signal Monitor (TSM) 19
B.2 Finite State Machine (FSM) 20

B.3 **[FSM_0/Autopilot] Not Realizable** 21
B.4 **[FSM_1/Autopilot] Not Realizable** 21
B.5 **[FSM_2/Autopilot]: Not Realizable** 22
B.6 **[FSM_4/Autopilot]: Not Realizable** 23
B.7 **[FSM_0/Sensor]: Not Realizable** 24
B.8 Tustin Integrator (TUI) 25
B.9 Neural Network (NN) 25
B.10 Autopilot (AP) 26
B.11 **[AP_2]: Not Realizable** 26
B.12 **[AP_4]: Realizable** 27
B.13 Roll Autopilot (RAP) 27

List of Figures

2.1 Lustre Template Example . . . 2
2.2 Realizability Annotations Generated from Figure 2.1 . . . 3

3.1 Dependency Graph of FSM . . . 6
3.2 Connected Components of REG . . . 9
3.3 Connected Components of AP . . . 10
3.4 Connected Components of FSM . . . 11
3.5 Connected Components of EB . . . 11
3.6 Connected Components of TSM . . . 12
3.7 Connected Components of TUI . . . 12
3.8 Connected Components of NN . . . 13

B1 FSM/Autopilot . . . 20

List of Tables

3.1 REG Compositional Testing 10
3.2 REG Monolithic Testing 10
3.3 AP Compositional Testing 10
3.4 AP Monolithic Testing 10
3.5 FSM Compositional Testing 11
3.6 FSM Monolithic Testing 11
3.7 EB Compositional Testing 11
3.8 EB Monolithic Testing 12
3.9 TSM Monolithic Testing 12
3.10 TUI Monolithic Testing 13
3.11 NN Monolithic Testing 13

B1 [**FSM_0**] Counter Example 21
B2 [**FSM_1/Autopilot**] Counter Example 22
B3 [**FSM_2**] Counter Example 23
B4 [**FSM_4**] Counter Example 24
B5 [**FSM_0/Sensor**] Counter Example 24
B6 [**AP_2**] Counter Example 26

Chapter 1

Preliminaries

Realizabilty checks whether for all inputs there exists a system that satisfies a set of given properties. Let $\vec{x} = \{x_1, x_2, ..., x_n\} \in \mathcal{X}^n$ be n inputs to a system with each input's domain being $\mathbb{X}_1, \mathbb{X}_2, \dots, \mathbb{X}_n$ respectively. Let $\vec{y} = \{y_1, y_2, ..., y_m\} \in \mathcal{Y}^m$ be m outputs to a system with each output's domain being $\mathbb{Y}_1, \mathbb{Y}_2, \dots, \mathbb{Y}_n$ respectively.

We define a system property property as a function $p : \mathcal{X}^n \times \mathcal{Y}^m \mapsto \ \mathbb{B} = \{\top, \bot\}$. $\mathbb{P}$ is a set of k properties. $\mathbb{P}(\vec{x}, \vec{y})$ is an alias for the conjunction of k properties. That is, $\mathbb{P}(\vec{x}, \vec{y}) = p_1(\vec{x_1}, \vec{y_1}) \wedge p_2(\vec{x_2}, \vec{y_2}) \wedge ... \wedge p_k(\vec{x_k}, \vec{y_k})$, where $\bigcup_{i=1}^{k} \vec{x}_i = \vec{x}$ and $\bigcup_{i=1}^{k} \vec{y}_i = \vec{y}$ respectively. The domain for each subvector $\vec{x}_i$ and $\vec{y}_i$ is $\mathcal{X}_i \subseteq \mathcal{X}^n$ and $\mathcal{Y}_i \subseteq \mathcal{Y}^m$, respectively. Then, realizability can be defined as follows:

Definition 1.01 (Realizability)**.** *Given a set of possible inputs and a set of possible outputs, a set of properties is realizable iff*

$$\forall \vec{x}.\ \exists \vec{y}.\ \mathbb{P}(\vec{x}, \vec{y}) \tag{1.1}$$

Consistency checking is weaker than realizability checking. Consistency checks whether for at least an input there exists a system that satisfies a set of given properties and can be defined

Definition 1.02 (Consistency)**.** *Given a set of possible inputs and a set of possible outputs, a set of properties is consistent iff*

$$\exists \vec{x}.\ \exists \vec{y}.\ \mathbb{P}(\vec{x}, \vec{y}) \tag{1.2}$$

Chapter 2

Realizability Testing with FRET and JKind

The Formal Requirements Elicitation and Testing (FRET) tool enables the formalization and elicitatation of system requirements. Requirements can be given as input to FRET by using an intuitive control natural language (CNL). Each CNL statement is translated into a temporal logic formula. These temporal logic formulas are used to generate CoCoSpec [1]. CoCoSpec is an extension of the Lustre [2] synchronous dataflow programming language. CoCoSpec contracts may be used for model checking or creating monitors.

However, the JKind realizability engine only supports requirement specifications written in Lustre. JKind is an infinite model checking library [3]. The generation of Lustre code through FRET is explained before summarizing realizability testing in JKind.

2.1 Generation of Lustre Code Through FRET

Support for Lustre code generation was added to FRET in order to enable realizability testing. JKind uses Lustre with annotations to denote properties and inputs. Support for Lustre code generation was added to FRET by creating Lustre-specific templates. Information from FRET's CNL is input into these templates to create valid Lustre code. Figure 2.2 shows the annotations required at the end of a Lustre node that are needed for JKind realizability testing.

```
1  <% for (i=0; i < properties.length; i++) {%>
2  (* <%-properties[i].fullText -%>  *)
3  <%var newid = properties[i].reqid.replace(/-/g,'')%>
4  <%- newid -%> = <%- properties[i].value -%>;
5
6  <%}%>
7  <% for(i=0; i < properties.length; i++){%>
8  <%var newid = properties[i].reqid.replace(/-/g,'')%> --%PROPERTY <%- newid %>; <%}%>
9  <%for(i=0; i < inputVariables.length;i++){%><%if(i==0){%>--%REALIZABLE <%}%><%- input
   Variables[i].name -%><%if(i<inputVariables.length-1){%>,<%}else{%>;<%}%>
10 <%}%>
11
12 tel
```

Figure 2.1: Lustre Template Example

Since FRET already supported the generation of CoCoSpec code, we had to identify the syntactic differences between CoCoSpec and Lustre in order to support the generation of JKind Lustre code. These syntactic differences include:

```
 --%PROPERTY FSM001v2;
 --%PROPERTY FSM001v3;
 --%PROPERTY FSM001v1;
--%REALIZABLE apfail, limits, standby, supported;

tel
```

Figure 2.2: Realizability Annotations Generated from Figure 2.1 Template

- Lustre for JKind does not support trailing semicolons in node declarations;
- Lustre for JKind does not support const modifiers for node arguments;
- Lustre for JKind requires that all (input and output) variables are declared as inputs (node arguments);
- Lustre for JKind differentiates between input and output variables by adding the input variables in the `%REALIZABLE` annotation.

2.2 Realizability Testing Through JKind

Two methods from JKind were used to compute realizability of a set of properties.

1. Fixpoint using AE-VAL [4]
2. K-Induction using z3 [5]

These will be referred to as FP and KI, respectively.

2.3 Background

JKind checks realizability using Assume-Guarantee contracts.

Definition 2.31 (Transition System)**.** *A transition system is* (I, T) *where* $I : state \mapsto \mathbb{B}$ *defines the possible initial states of the system and* $T : state \times input \times state \mapsto \mathbb{B}$ *defines the possible transitions from one state to another.*

Definition 2.32 (Assume-Guarantee Contract)**.** (A, G) *is an Assume-Guarantee contract.* $A : state \times input \mapsto \mathbb{B}$ *defines what inputs are valid for a given state.* $G = (G_I, G_T)$ *where* $G_I : state \mapsto \mathbb{B}$ *defines the possible states a system must start in and* $G_T : state \times input \times state \mapsto \mathbb{B}$ *defines the transitions a state is allowed to take.*

Definition 2.33 (Viable State)**.** *A viable state wrt a contract* (A, G) *is a state from which the system can continue forever. Formally:*

$$Viable(s) = \forall i. A(s, i) \implies \exists s'. G_T(s, i, s') \wedge Viable(s') \tag{2.1}$$

Definition 2.34 (Finite Viable State)**.** *A finite viable state wrt a contract* (A, G) *is a state from which the system can continue for at least n steps. Formally:*

$$\begin{aligned} Viable_n(s) =& \forall i_1. A(s, i_1) \implies \exists s_1. G_T(s, i_1, s_1) \wedge \\ & ... \wedge \forall i_n. A(s_{n-1}, i_n) \implies \exists s_n. G_T(s_{n-1}, i_n, s_n) \end{aligned} \tag{2.2}$$

Authors of [5] prove that a necessary and sufficient condition for realizability is the following:

$$\text{Contract } (A, G) \text{ is realizable } \iff \exists s. G_I(s) \wedge Viable(s) \tag{2.3}$$

2.4 Fixpoint

The fixpoint realizability algorithm is built on the ∀∃-formula solver AE-VAL [6]. Authors of [4] take the realizability formula $\forall \vec{x}.\ \exists \vec{y}.\ \mathbb{P}(\vec{x}, \vec{y})$ and form the equivalence

$$\forall \vec{x}.\ \exists \vec{y}.\ \mathbb{P}(\vec{x}, \vec{y}) \equiv \forall \vec{x}.\ S(\vec{x}) \implies \exists \vec{y}.\ T(\vec{x}, \vec{y}) \tag{2.4}$$

AE-VAL is used to extract regions in $dom\ \vec{x}$ that satisfy $S(\vec{x})$. JKind iteratively computes the maximal region that satisfies $S(\vec{x})$. If such a region exists then for all $\vec{x}$ in that region the original formula is realizable.

2.5 K-Induction

K-Induction is a generalization of standard induction and can be used to prove realizability of a set of properties. First, we describe standard induction.

Standard induction is defined as follows:

$$P(0) \wedge \forall n.\ (P(n) \implies P(n+1)) \implies \forall n.\ P(n) \tag{2.5}$$

There are three possibilities for standard induction.

1. The base case and inductive step are true $\implies$ the property is always true
2. The base case if false $\implies$ the property is not always true
3. The base case is true but the inductive step is false $\implies$ no conclusion can be made

Since software states evolve over time it may be necessary to perform multiple inductive iterations. K-Induction is a generalization of standard induction and is defined [7]:

$$I(k) := \left(\bigwedge_{i=0}^{k-1} P(i)\right) \wedge \forall n.\ \left(\left(\bigwedge_{i=0}^{k-1} P(n+i)\right) \implies P(n+k)\right) \tag{2.6}$$

$$I(k) \implies \forall n.\ P(n) \tag{2.7}$$

Authors of [5] define a base check and an inductive step:

$$\mathbf{BaseCheck(n)} = \exists s. G_I(s) \wedge \text{Viable}_n(s) \tag{2.8}$$

$$\mathbf{ExtendCheck(n)} = \forall s. Extend_n(s) \tag{2.9}$$

It follows from the definition of K-Induction in Equation 2.6 that if $\exists n.\mathbf{BaseCheck(n)} \wedge \mathbf{ExtendCheck(n)}$ then the contract is realizable.

2.6 Notes

The implementation of the K-Induction algorithm is an approximation of the algorithm presented in [5]. One consequence of this approximation is that unrealizable results from the K-Induction algorithm are not sound. For example section 3.3.6 shows an unsound unrealizable result returned by the K-Induction implementation.

Chapter 3

Compositional Realizability Testing

Compositionally testing the realizability of a set of properties has certain benefits. For instance, compositional testing may offer higher resolution in realizability results and can provide speedup in test time.

However, care must be taken during the decomposition because existential quantification does not distribute over conjunction:

$$\begin{aligned} &\forall \vec{x}.\ \exists \vec{y}.\ \mathbb{P}(\vec{x}, \vec{y}) \not\equiv \\ &\qquad \forall \vec{x}.\ \left(\exists \vec{y}_1.\ p_1(\vec{x}_1, \vec{y}_1) \wedge \exists \vec{y}_2.\ p_2(\vec{x}_2, \vec{y}_2) \wedge ... \wedge \exists \vec{y}_k.\ p_k(\vec{x}_k, \vec{y}_k) \right) \end{aligned} \tag{3.1}$$

However, we will next prove that properties can be decomposed if their outputs are independent. To achieve this result we develop an output dependency graph from which we can collect disjoint sets of properties. We prove that these pairwise disjoint sets of properties can be used to compositionally check for realizability.

3.1 Output Dependency Graphs

An output dependency graph shows the dependencies between requirements and output variables, i.e., on which outputs properties depend on. Let $D = (V, E)$ be a dependency graph. Vertices may either be properties or output variables: $V = \{p_i | p_i \in \mathbb{P}\} \cup \{y_i | \forall y_i \in \vec{y}\}$. Edges are defined as connections between properties and output variables: $E \subset \{(p, y_i) | (p, y_i) \in V^2,\ p \in P,\ y_i \in \vec{y}\}$.

Figure 3.1 shows three outputs of the FSM component, SENSTATE, STATE, pullup, and the properties that depend on them.

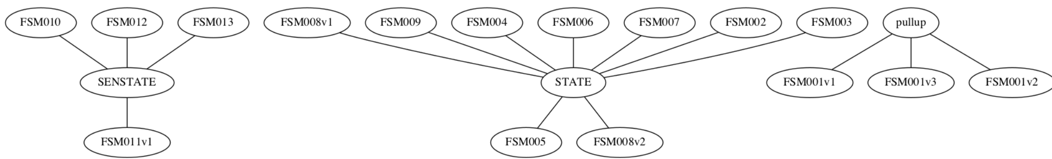


Figure 3.1: Dependency Graph of FSM

3.2 Connected Components

Let $O(p_i) : \mathbb{P} \mapsto \mathcal{Y}_i \subseteq \mathcal{Y}^m$ be a function that maps a proposition to the set of output variables it depends on.

An output connected component is $C_k = (P_k, \vec{y}_k)$ where $P_k = \textbf{maximal}(\{\bar{P} \in 2^{\mathbb{P}} | \forall p_i, p_j \in \bar{P}.\ i \neq j,\ O(p_i) \cap O(p_j) \neq \varnothing\})$. $\textbf{maximal}(\cdot)$ takes the maximal set of properties from all combinations of subsets. $\vec{y}_k = \{O(p_i) | \forall p_i \in P_k\}$ is the set of all output variables in the output connected component.

Let the number of elements in $\mathbb{C}$ be $|\mathbb{C}|$. Then the set of all output connected components is collected in $\mathbb{C} = \{C_1,\ C_2,\ ...,\ C_{|\mathbb{C}|}\}$. These output connected components are pairwise disjoint.

Definition 3.21 (Pairwise Disjoint Connected Components)**.** *All the elements within a set $\mathbb{C}$ of output connected components are pairwise disjoint if*

- $\forall (P_k,\ \vec{y}_k),\ (P_j,\ \vec{y}_j) \in \mathbb{C}$ *and* $k \neq j,\ P_k \cap P_j = \varnothing,\ \vec{y}_k \cap \vec{y}_j = \varnothing$
- $\mathbb{P} = \bigcup_{P_k \in \mathbb{C}} P_k$ *and* $\vec{y} = \bigcup_{\vec{y}_k \in \mathbb{C}} \vec{y}_k$

Let $C_k(\vec{x},\ \vec{y}_k) = \bigwedge_{i=1}^{|\mathbb{P}|} p_i(\vec{x},\ \vec{y}_k)$ denote the conjunction of all properties in an output connected component.

Lemma 3.21. *A set of properties $\mathbb{P}$, and an output vector $\vec{y}$ with $|\mathbb{C}|$ output connected components can be written as the conjunction of each output connected component.*

$$\mathbb{P}(\vec{x}, \vec{y}) = \bigwedge_{k=1}^{|\mathbb{C}|} C_k(\vec{x}, \vec{y}_k) \tag{3.2}$$

Proof. Follows directly from Definition 3.21 of Pairwise Disjoint Connected Components.

Theorem 3.22. *Let $\mathbb{P}$ be a set of properties that consists of $|\mathbb{C}|$ output connected components. The realizability of $\mathbb{P}(\vec{x},\ \vec{y})$ is the conjunction of the realizability of each output connected component.*

$$\forall \vec{x}.\ \exists \vec{y}.\ \mathbb{P}(\vec{x}, \vec{y}) \equiv \forall \vec{x}.\ \bigwedge_{k=1}^{|\mathbb{C}|} \exists \vec{y}_k.\ C_k(\vec{x},\ \vec{y}_k) \tag{3.3}$$

Proof. By induction.

Base Case (Two Outputs, Two connected components)
The two connected components are $\mathbb{C} = \{(P_1, \vec{y}_1), (P_2, \vec{y}_2)\}$ and by Lemma 3.21 we have $\mathbb{P}(\vec{x}, \vec{y}) = C_1(\vec{x}, y_1) \wedge C_2(\vec{x}, y_2)$ and $\vec{y} = (y_1, y_2)$

$$\begin{aligned}
&\forall \vec{x}.\ \exists \vec{y}.\ \mathbb{P}(\vec{x}, \vec{y}) = \\
&\forall \vec{x}.\ \exists y_1,\ \exists y_2.\ C_1(\vec{x},\ y_1) \wedge C_2(\vec{x},\ y_2) = \\
&\forall \vec{x}.\ \exists y_1.\ (\exists y_2.\ C_1(\vec{x},\ y_1) \wedge C_2(\vec{x},\ y_2)) = \\
&\forall \vec{x}.\ \exists y_1.\ C_1(\vec{x},\ y_1) \wedge C_2(\vec{x},\ f_2(\vec{x})) = && \text{(By skolemization)} \\
&\forall \vec{x}.\ C_1(\vec{x},\ f_1(\vec{x})) \wedge C_2(\vec{x},\ f_2(\vec{x})) = && \text{(By skolemization)} \\
&\forall \vec{x}.\ (\exists y_1.\ C_1(\vec{x},\ y_1)) \wedge (\exists y_2.\ C_2(\vec{x},\ y_2)) = && \text{(By existential generalization)} \\
&\forall \vec{x}.\ \bigwedge_{k=1}^{2} \exists y_k.\ C_k(\vec{x},\ y_k)
\end{aligned}$$

Assumption of $|\mathbb{C}|^{th}$ Case

$$\forall \vec{x}.\ \exists \vec{y}.\ \mathbb{P}(\vec{x},\ \vec{y}) = \forall \vec{x}.\ \bigwedge_{k=1}^{|\mathbb{C}|} \exists \vec{y_k}.\ C_k(\vec{x},\ \vec{y_k})$$

Inductive case (m outputs, $|\mathbb{C}|+1 \leq m$ connected components)
By Lemma 3.21 we have $\mathbb{P}(\vec{x},\vec{y}) = C_1(\vec{x},\ \vec{y_1}) \wedge C_2(\vec{x},\vec{y_2}) \wedge ... \wedge C_{|\mathbb{C}|+1}(\vec{x},\vec{y}_{|\mathbb{C}|+1})$ and by Definition 3.21 we have $\vec{y} = (\vec{y_1},\ \vec{y_2},\ ...,\ \vec{y}_{|\mathbb{C}|+1})$.

$$\forall \vec{x}.\ \exists \vec{y_1}, \exists \vec{y_2}, ..., \exists \vec{y}_{|\mathbb{C}|},\ \exists \vec{y}_{|\mathbb{C}|+1}.\ C_1(\vec{x},\vec{y_1}) \wedge C_2(\vec{x},\ \vec{y_2}) \wedge ... \wedge C_{|\mathbb{C}|}(\vec{x},\ \vec{y}_{|\mathbb{C}|}),\ C_{|\mathbb{C}|+1}(\vec{x},\ \vec{y}_{|\mathbb{C}|+1}) =$$

$$\forall \vec{x}.\ \bigwedge_{k=1}^{|\mathbb{C}|} \exists \vec{y_k}.\ C_k(\vec{x},\ \vec{y_k}) \wedge \exists \vec{y}_{|\mathbb{C}|+1}.\ C_{|\mathbb{C}|+1}(\vec{x},\vec{y}_{|\mathbb{C}|+1}) =$$

$$\forall \vec{x}.\ \bigwedge_{k=1}^{|\mathbb{C}|+1} \exists \vec{y_k}.\ C_k(\vec{x},\ \vec{y_k}) \qquad \text{(Assuming the } |\mathbb{C}|^{th} \text{ case and existential elimination)}$$

□

Skolemization and Existential Generalization

Theorem 3.22 distributes existential quantifiers over conjunction due to skolemization and existential generalization.

Definition 3.22 (Skolemization)**.** *Skolemization is a form of quantifier elimination:*

$$\forall x.\ \exists y.\ P(x,y) \equiv \forall x.\ P(x, f(x))$$

where $f(x)$ is a function that makes $P(x, f(x))$ true.

Definition 3.23 (Existential Generalization)**.** *Existential Generalization is a quantifier rule in first order logic that says:*

$$A(t) \equiv \exists x.\ A(x)$$

for some term t that makes A true.

3.3 Computing The Set of Connected Components

After JKind parses a Lustre node it forms a dependency map that maps variable names to the set equations and variables it depends on. The set of connected components $\mathbb{C}$ is created by Algorithm 1. Algorithm 1 takes as input a set of properties $\mathbb{P}$ and a function $O(p_i) : \mathbb{P} \mapsto \mathcal{Y}_i \subseteq \mathcal{Y}^m$ that maps a proposition to the outputs it depends on. In practice O is a hash map with proposition names as keys and sets of outputs as values.

The algorithm iterates through all properties and adds a property to a connected component if the property shares at least one output with the connected component.

Algorithm 1: Compute Connected Components

```
input  : ℙ, O
output: ℂ
ℂ = ∅;
for p_i ∈ ℙ do
    if ℂ = ∅ then
        C_new = ({p_i}, O(p_i));
        ℂ.add(C_new);
    else
        has_intersection = FALSE;
        for C_k ∈ ℂ do
            (P_k, y⃗_k) = C_k;
            if y⃗_k ∩ O(p_i) ≠ ∅ then
                P_k = P_k ∪ p_i;
                y⃗_k = y⃗_k ∪ O(p_i);
                has_intersection = TRUE;
                break;
            end
        end
        if !has_intersection then
            C_new = ({p_i}, O(p_i));
            ℂ.add(C_new);
        end
    end
end
```

3.3.1 Algorithm Complexity

The worst case complexity occurs when no properties have overlapping outputs. Set additions, intersection, and union operations are assumed to be constant time. The upper bound of complexity is $O(|\mathbb{P}|^2)$.

3.3.2 Example: Regulator

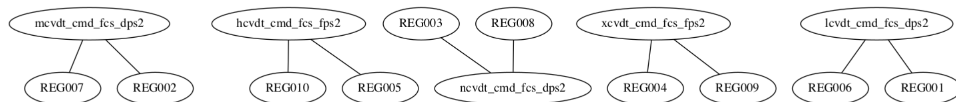


Figure 3.2: Connected Components of REG

Figure 3.2 shows five connected components. Due to Theorem 3.22 the conjunction of the realizability results is equivalent to realizability of the monolithic component and the connected components can be individually tested for realizability. We also see improvement in test time.

Properties Tested	Result(KI)	Time	Result(FP)	Time
REG007, REG002	Unknown	1m 40s*	Realizable	0.4s
REG010, REG005	Unknown	1m 40s*	Realizable	0.4s
REG003, REG008	Unknown	1m 40s*	Realizable	0.4s
REG004, REG009	Unknown	1m 40s*	Realizable	0.4s
REG006, REG001	Unknown	1m 40s*	Realizable	0.4s
	Total Time	3m 20s	Total Time	2s

Table 3.1: REG Compositional Testing

* Timeout

Properties Tested	Result(KI)	Time	Result(FP)	Time
All Properties	Unknown	26.4s	Realizable	1m 3s

Table 3.2: REG Monolithic Testing

3.3.3 Example: Autopilot

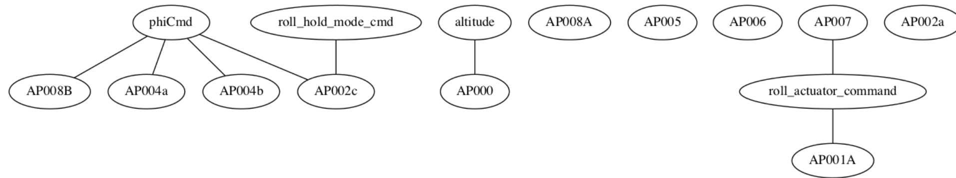


Figure 3.3: Connected Components of AP

Input-only requirements AP005, AP006, are considered assertions. Requirements AP008A and AP002a are mode requirements and will be merged with their counterparts AP008B and AP002a.

The Autopilot contains three connected components. We see that of compositional testing results in more information. Additionally feedback can be provided to the user about input-only requirements.

Properties Tested	Result(KI)	Time	Result(FP)	Time
AP004a, AP004b, AP002c, AP008B	Unknown	0.2s	Unknown	48h*
AP000	Realizable	0.1s	Realizable	0.3s
AP001A, AP007	Realizable	0.3s	Realizable	3.97s
	Total Time	0.6s		48h 4s

Table 3.3: AP Compositional Testing

Properties Tested	Result(KI)	Time	Result(FP)	Time
All Properties	Unknown	0.2	Unknown	1m 40s*

Table 3.4: AP Monolithic Testing

*Timeout

3.3.4 Example: FSM

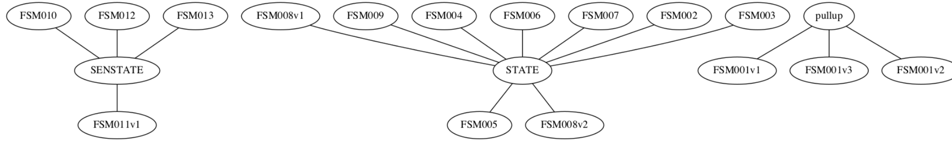


Figure 3.4: Connected Components of FSM

Properties Tested	Result(KI)	Time	Result(FP)	Time
Subset 1	Realizable	1.54s	Realizable	2.04s
Subset 2	Unrealizable	0.1s	Unrealizable	1.54s
Subset 3	Unrealizable	0.1s	Unrealizable	0.5s
	Total Time	1.74s		4.08s

Table 3.5: FSM Compositional Testing

- Subset 1: FSM0001v2, FSM0001v3, FSM001v1
- Subset 2: FSM002, FSM003, FSM008v1, FSM009, FSM005, FSM008v2, FSM007, FSM004, FSM006
- Subset 3: FSM011v2, FSM012, FSM010, FSM013

Properties Tested	Result(KI)	Time	Result(FP)	Time
All Properties	Unrealizable	0.2	Unrealizable	11h 22m

Table 3.6: FSM Monolithic Testing

*Timeout

3.3.5 Example: Effector Blender

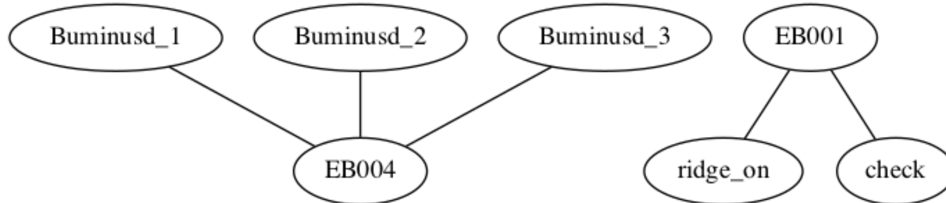


Figure 3.5: Connected Components of EB

Properties Tested	Result(KI)	Time	Result(FP)	Time
EB001	Unknown	0.4s	Unknown	0.2s
EB004	Unknown	0.1s	Unknown*	0.3s
	Total Time	0.5s		0.5s

Table 3.7: EB Compositional Testing

Unknown results for Fixpoint are due to non-linear requirements.
*AEVAL Error: Non-linear fault

Properties Tested	Result(KI)	Time	Result(FP)	Time
All Properties	Unknown	0.9s	Unknown	0.5s

Table 3.8: EB Monolithic Testing

3.3.6 Example: Triplex Signal Monitor

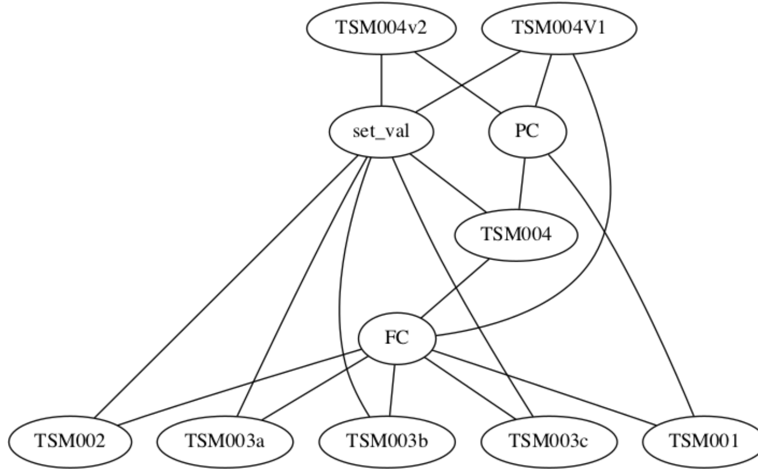


Figure 3.6: Connected Components of TSM

Testing the Triplex Signal Monitor shows an unsound result from K-Induction. Since we know FP is sound, for realizable and unrealizable results we can conclude the TSM component is realizable.

Properties Tested	Result(KI)	Time	Result(FP)	Time
All Properties	Unrealizable	0.72s	Realizable	1m 32s

Table 3.9: TSM Monolithic Testing

3.3.7 Example: Tustin Integrator

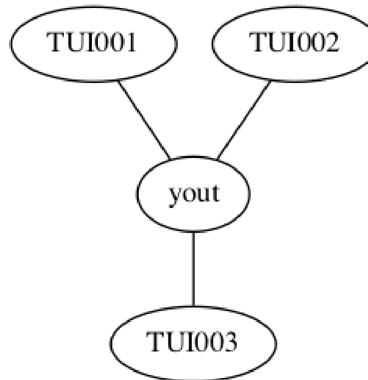


Figure 3.7: Connected Components of TUI

Properties Tested	Result(KI)	Time	Result(FP)	Time
All Properties	Unknown	0.1s	Unknown*	0.41s

Table 3.10: TUI Monolithic Testing

* AEVAL Non-Linear error

3.3.8 Example: Neural Network

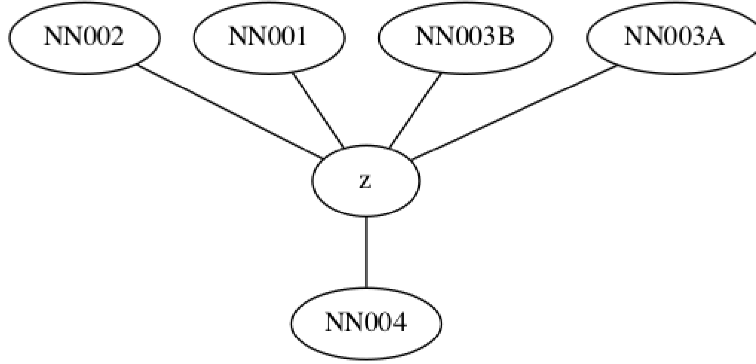


Figure 3.8: Connected Components of NN

Properties Tested	Result(KI)	Time	Result(FP)	Time
All Properties	Unknown	34.54s	Unknown	6m 37s*

Table 3.11: NN Monolithic Testing

* Timeout (Possibly Unknown due to non-constant division.)

Chapter 4

Conclusion

This report summarizes work towards supporting realizability testing through FRET and JKind. FRET was first extended to produce specifications compatible with JKind. Two realizability checking algorithms were used: Fixpoint method and a K-Induction method. A number of manual tests were performed on the Lockheed Martin Cyber Physical Challenge Problems [8]. These manual tests revealed the need to compositionally test properties.

4.1 Identified Issues

Realizability testing uncovered several notable issues.

- Inconsistent results with Neural Network (Section B.9) component revealed typing error in JKind. The pre-initialization operator in lustre was casting the values to integers instead of reals. Issue communicated with JKind researcher Andreas Katis[1] and fixed in JKind 1.6.1.
- AE-VAL solver has trouble with formulas containing both integers and real numbers. See Section B.1 of TSM results. Issue communicated with AE-VAL creator Grigory Fedyukovich[2].
- Ran into non-linear limitations for the AE-VAL solver. See Example 3.3.5 and 3.3.7 showing the Effector Blender and Tustin Integrator. Issue communicated with AE-VAL creator Grigory Fedyukovich[2].
- RollAutopilot (Section B.13) revealed a parser error in JKind. Under certain conditions the transition relation was not being provided to the SMT solver. Issue communicated with Andreas Katis[1] and fixed in JKind 1.6.1.
- Components with only inputs have their realizability formula reduced from $\forall x.\ \exists y.\ P(x, y)$ to $\forall x.\ P(x)$. Since the input is uncontrollable this always creates an unrealizable result. As a result, input-only properties are treated as assertions. See Autopilot Example 3.3.3.
- Unsound K-Induction result. See Triplex Signal Monitor Example 3.3.6.
- Realizability testing identifies unguarded FSM transitions. See Section B.2.

[1] katis001@umn.edu
[2] grigory.fedyukovich@gmail.com

4.2 Compositional Testing

Compositional testing provides several advantages due to the property

$$\forall \vec{x}.\ \exists \vec{y}.\ \mathbb{P}(\vec{x}, \vec{y}) \equiv \bigwedge_{k=1}^{|\mathbb{C}|} \forall \vec{x}.\exists \vec{y}_k.\ C_k(\vec{x}, \vec{y}_k)$$

Testing subsetts of properties ....

- ... **may be more efficient**: Monolithic testing may be more prone to solver timeouts. Consider Regulator Example 3.3.2: Compositional testing provides a result within 2 seconds while monolithic testing takes 1m 2s. A larger component may push realizability testing duration towards a timeout.
- ... **provides more information**: Consider Autopilot Example 3.3.3. Monolithic testing provides one Unknown result. Compositional testing provides three results, two realizable and one unknown. This helps narrow down problem causes.
- ... **identifies assertion-only properties**: Consider Autopilot Example 3.3.3. Properties that do not depend on outputs are identified as singletons.

Chapter 5

Appendix

Appendix A

Results

Realizabilty testing was performed over components **TSM, FSM, NN, AP, RAP, TUI** using the JKind realizability tester.

A.1 Realizability Statistics

	# Realizable	# Unrealizable	# Unknown	# Tests
FP	20	13	15	48
KI	15	11	22	48

Unknown by KI, Determined By FP
2 (TSM), 1 (TUI), 2 (AP), 4 (RAP)

A.2 Fixpoint and K-Indution Non-Linear Performance

K-Induction tests were run on the Z3 SMT solver. Z3 documentation notes that Z3 is not complete for non-linear formulas of the form $(*\ t\ s)$ where t and s are non-constant and multiplied. [9].

Fixpoint tests were run on the AE-VAL solver. AE-VAL does not support non-linear operations.

A.3 TSM

The following properties contain non-constants multiplied by a constant and solvable by both algortihms.

Fixpoint: Realizable for **TSM-003a,b,c**
K-Induction: Realizable for **TSM-003a,b,c**

TSM-003a:

TriplexSignalMonitor shall always satisfy FC = 1 $\implies$ (set_val = 0.5 * (ia +ib))

TSM-003b:

TriplexSignalMonitor shall always satisfy FC = 2 $\implies$ (set_val = 0.5 * (ia +ic))

TSM-003c:

TriplexSignalMonitor shall always satisfy FC = 2 $\implies$ (set_val = 0.5 * (ia +ic))

ia and ib are unbounded inputs and FC can be 0, 1, 2, or 4.

A.4 TUI

The following properties contain non-constants divided by non-constants. Fixpoint is able to make a conclusion, but K-Induction is not.

Fixpoint: Realizable for **TUI-003**
K-Induction: Unknown for **TUI-003**

TUI-003: Tustin_Integrator shall always satisfy

$$\text{normal} \implies (\text{yout} = \text{T}/2.0 * (\text{xin} + \text{xinpv}) + \text{ypv})$$

Where T, xin are inputs and yout is an output. xinpv depends on xin and ypv depends on yout.

A.5 NN

The following properties contain non-constants divided by non-constants.Both algorithms are unable to make a conclusion.

Fixpoint: Unknown for **NN-003a,b**
K-Induction: Unknown for **NN-003a,b**

NN-003a:

NN shall for 200 secs satisfy DeltaZDividedByXt $\leq$ 10.0 & DeltaZDividedByXt $\geq$ -35.0

NN-003b:

NN shall for 200 secs satisfy DeltaZDividedByXt $\leq$ 10.0 & DeltaZDividedByXt $\geq$ -35.0

Where

$$\text{DeltaZDividedByXt} = 0.0 \implies (z - pre_z)/(xt - pre_xt)$$
$$\text{DeltaZDividedByYt} = 0.0 \implies (z - pre_z)/(yt - pre_yt)$$

A.6 AP

Fixpoint: Unknown for **AP-004b**
K-Induction: Unknown for **AP-004b**

AP004b: When in roll_hold mode Autopilot shall always satisfy overshoot ≤ 0.1

Where

$$\text{overshoot} = (\text{roll_angle} - \text{step})/\text{step}$$

roll_angle and step are inputs.

Appendix B

List of Manual Realizability Checking Results

Each component was tested over multiple iterations to observe other combinations of inputs. These runs take the notation of [component]_0, [component]_1,..., etc.

B.1 Triplex Signal Monitor (TSM)

Testing discovered an AEval limitation: The solver has trouble with properties containing integers and real values. This can cause unknown results in Fixpoint testing since the Fixpoint algorithm is build on AEval.

Several tests were performed with integer datatypes replaced with real datatypes.

Ints and reals

Component	FP Result	FP Time	KI Result	KI Time
TSM_0	Unknown	9.6s	Unknown	17s
TSM_1	Unknown	8.7s	Unknown	12.6s
TSM_2	Realizable	0.3s	Realizable	0.1s
TSM_3	Realizable	0.3s	Realizable	0.1s
TSM_4	Unknown	3.2s	Realizable	5.1s
TSM_5	Unknown	7.4s	Unknown	14s
TSM_6	Unknown	5.2s	Realizable	6.2s
TSM_7	Unknown	6.4s	Unknown	27.1s

Ints replaced with reals

Component	FP Result	FP Time	KI Result	KI Time
TSM_0_real	Unrealizable	1m 6s	Unknown	48s
TSM_1_real	Unrealizable	1m 17s	Unknown	45s
TSM_4_real	Realizable	7s	Realizable	3s
TSM_5_real	Realizable	3s	Realizable	8s
TSM_6_real	Realizable	4s	Realizable	10s
TSM_7_real	Realizable	4s	Realizable	5s

Run Details:

- TSM_0: All properties
- TSM_1: Without [TSM-003a], [TSM-003b], and [TSM-003c]
- TSM_2: Only [TSM-003a]
- TSM_3: Only [TSM-003a], [TSM-003b], and [TSM-003c]
- TSM_4: Only [TSM-003a], [TSM-003b], [TSM-003c], and [TSM-001]
- TSM_5: Only [TSM-003a], [TSM-003b], [TSM-003c], [TSM-001], and [TSM-004]
- FSM_6: Only [TSM-003a], [TSM-003b], [TSM-003c], [TSM-001], and [TSM-004]
- TSM_7: Only [TSM-003a], [TSM-003b], [TSM-003c], [TSM-001], and [TSM-002]

B.2 Finite State Machine (FSM)

Run	FP Result	FP Time	KI Result	KI Time
FSM_0	Realizable	2s	Realizable	1.8s
FSM_0/Autopilot	Unrealizable	1.8s	Unrealizable	2s
FSM_1/Autopilot	Unrealizable	0.4s	Unrealizable	0.1s
FSM_2/Autopilot	Unrealizable	0.7s	Unrealizable	0.1s
FSM_3/Autopilot	Realizable	0.3s	Realizable	0.1s
FSM_4/Autopilot	Unrealizable	1.2s	Unrealizable	0.11s
FSM_5/Autopilot	Realizable	2.6s	Realizable	0.1s
FSM_6/Autopilot	Realizable	0.5s	Realizable	0.1s
FSM_7/Autopilot	Realizable	0.5s	Realizable	0.1s
FSM_0/Sensor	Unrealizable	0.7s	Unrealizable	0.1s
FSM_1/Sensor	Realizable	0.6s	Realizable	0.1s

Run Details:

- FSM_0: All properties
- FSM_0/Autopilot: All properties
- FSM_1/Autopilot: Only [FSM-006] and [FSM-007]
- FSM_2/Autopilot: Without [FSM-006] and [FSM-007]
- FSM_3/Autopilot: Only [FSM-004v2] and [FSM-005]
- FSM_4/Autopilot: Without [FSM-006], [FSM-007] and [FSM-008v1]
- FSM_5/Autopilot: Without [FSM-006], [FSM-007], [FSM-008v1] and [FSM-002]
- FSM_6/Autopilot: Only [FSM-004] and [FSM-005]
- FSM_7/Autopilot: Only [FSM-006], [FSM-008], and [FSM-002]
- FSM_0/Sensor: All properties
- FSM_1/Sensor: Without [FSM-011]

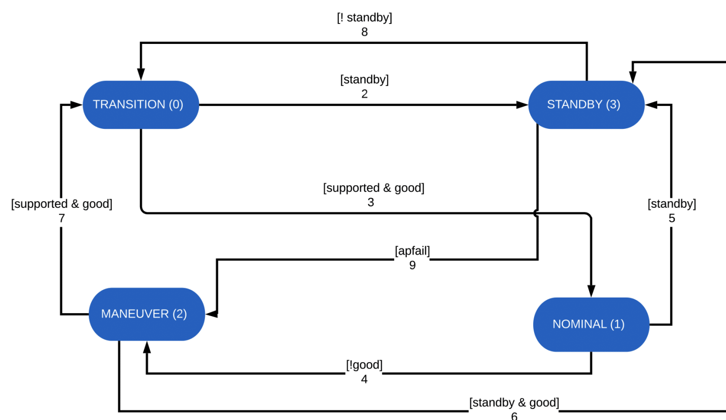


Figure B1: FSM/Autopilot

B.3 [FSM_0/Autopilot] Not Realizable

Not realizable because [**FSM-006**] and [**FSM-007**] are contradictory properties. Both antecedents can be true, but both consequents cannot. As show in Figure B1, both transitions from ap_manuever_state cannot be satisfied at once.

Jrealizability produces a counterexample showing both properties false and an invalid transition to STATE 1. The cause for the STATE = 1 is still under investigation.

Requirements

[label=][**FSM-006**]:
FSM shall always satisfy (state = ap maneuver state & standby & good) $\implies$ STATE = ap standby state
[**FSM-007**]:
FSM shall always satisfy (state = ap maneuver state & supported & good) $\implies$ STATE = ap transition state

Table B1: [**FSM_0**] Counter Example

Inputs	T = 0
apfail	False
good	True
standby	True
state	2
supported	True
State Values	
ap_transition_state	0
ap_nominal_state	1
ap_manuever_state	2
ap_standby_state	3
Properties	
FSM-006	False
FSM-007	False
Outputs	
STATE	1

B.4 [FSM_1/Autopilot] Not Realizable

Not realizable because [**FSM-006**] and [**FSM-007**] are contradictory properties. Both antecedents can be true, but both consequents cannot. As show in Figure B1, both transitions from ap_manuever_state cannot be satisfied at once.

However, the counter example returned explicitly shows the conflict between [**FSM-006**] and [**FSM-007**].

Requirements

[label=][**FSM-006**]:
FSM shall always satisfy (state = ap_maneuver_state & standby & good) $\implies$ STATE

= ap _standby_state
[FSM-007]:
FSM shall always satisfy (state = ap_maneuver_state & supported & good) $\implies$ STATE = ap_transition_state

Table B2: [**FSM_1/Autopilot**] Counter Example

Inputs	T = 0
apfail	False
good	True
standby	True
state	2
supported	True
State Values	
ap_transition_state	0
ap_nominal_state	1
ap_manuever_state	2
ap_standby_state	3
Properties	
FSM-006	False
FSM-007	True
Outputs	
STATE	0

B.5 [FSM_2/Autopilot]: Not Realizable

Not realizable because [**FSM-008v1**] and [**FSM-009**] are contradictory properties. Both antecedents can be true, but both consequents cannot. As show in Figure B1, both transitions from ap_standby_state cannot be satisfied at once.

Requirements

[label=][**FSM-008v1**]:
FSM_Autopilot shall always satisfy (state = ap_standby_state & !standby) $\implies$ STATE = ap_transition_state
[FSM-009]:
FSM_Autopilot shall always satisfy (state = ap_standby_state & apfail) $\implies$ STATE = ap_maneuver_state

Table B3: **[FSM_2]** Counter Example

Inputs	T = 0
apfail	True
good	False
standby	False
state	3
supported	False
State Values	
ap_transition_state	0
ap_nominal_state	1
ap_manuever_state	2
ap_standby_state	3
Properties	
FSM-008v1	False
FSM-009	True
Outputs	
STATE	2

B.6 [FSM_4/Autopilot]: Not Realizable

Not realizable because **[FSM-002]** and **[FSM-003]** are contradictory properties. Both antecedents can be true, but both consequents cannot. As show in Figure B1, both transitions from ap_standby_state cannot be satisfied at once.

Requirements

[label=]**[FSM-002]:**
FSM_Autopilot shall always satisfy (standby & state = ap_transition_state) $\implies$ STATE = ap_standby_state **[FSM-003]:**
FSM_Autopilot shall always satisfy (state = ap_transition_state & good & supported) $\implies$ STATE = ap_nominal_state

Table B4: [**FSM_4**] Counter Example

Inputs	T = 0
apfail	False
good	True
standby	True
state	0
supported	True
State Values	
ap_transition_state	0
ap_nominal_state	1
ap_manuever_state	2
ap_standby_state	3
Properties	
FSM-002	True
FSM-003	False
Outputs	
STATE	3

B.7 [FSM_0/Sensor]: Not Realizable

There is an ambiguous state transition from sen_nominal_state. The conflict arises between [**FSM-010**] and [**FSM-011**].

Requirements

[label=][**FSM-010**]:
FSM shall always satisfy (senstate = sen_nominal_state & limits) $\implies$ SENSTATE = sen fault state
[**FSM-011**]:
FSM shall always satisfy (senstate = sen_nominal_state & !request) $\implies$ SENSTATE = sen_transition_state

Table B5: [**FSM_0/Sensor**] Counter Example

Inputs	T = 0
MODE	False
limits	True
Request	False
senstate	0
State Values	
sen_nominal_state	0
sen_transition_state	1
sen_fault_state	2
Properties	
FSM-010	False
FSM-011	True
Outputs	
SENSTATE	1

B.8 Tustin Integrator (TUI)

Run	FP Result	FP Time	KI Result	KI Time
TUI_0	Unknown	0.4s	Unknown	0.1s
TUI_1	Realizable	0.2s	Realizable	0.1s
TUI_2	Realizable	0.5s	Unknown	0.2s
TUI_3	Unknown	0.8s	Unknown	0.1s
TUI_4	Unknown	0.3s	Unknown	0.1s

Run Details:

- TUI_0: All properties
- TUI_1: Without [TUI-003]
- TUI_2: Only [TUI-003]
- TUI_3: Only [TUI-001] and [TUI-003]
- TUI_3: Only [TUI-002] and [TUI-003]

B.9 Neural Network (NN)

Testing the Neural Network component revealed a bug in JKind realizability checker. ”Pre-initialization” operator in lustre was casting the values to integers instead of reals. This resulted in inconsistent results. The latest release (JKind 1.6.1) produced unknown results.

Run	FP Result	FP Time	KI Result	KI Time
NN_0	Unknown*	1m 40s	Unknown	35s
NN_1	Unknown*	1m 40s	Unknown	0.1s
NN_2	Unknown	46s	Unknown	0.1s
NN_3	Realizable	0.2s	Realizable	0.1s
NN_4	Unknown*	10h+	Unknown*	1m 40s

*Timeout

Run Details:

- NN_0: All properties
- NN_1: Without [NN-004]
- NN_2: Without [NN-004], [NN-003B]
- NN_3: Without [NN-004], [NN-003B], [NN-003A]
- NN_4: Only [NN-004], [NN-001], [NN-002]

B.10 Autopilot (AP)

Run	FP Result	FP Time	KI Result	KI Time
AP_0	Unknown	1m 40s	Unknown	0.2s
AP_1	Unrealizable	1.12s	Unrealizable	0.1s
AP_2	Unrealizable	0.8s	Unrealizable	0.1s
AP_3	Realizable	0.4s	Realizable	0.2s
AP_4	Realizable	3.17s	Unknown	13s
AP_5	Unknown	10m	Unknown	13s
AP_6	Realizable	11s	Unknown	1m 18s

Run Details:

- AP_0: All properties
- AP_1: Without [AP-004A] and [AP-004B]
- AP_2: Only [AP-005] and [AP-006]
- AP_3: Only [AP-000]
- AP_4: Only [AP-004A]
- AP_5: Only [AP-004B]
- AP_6: Only [AP-004A] and [AP-000]

Comments on unrealizability result of run AP_1 are skipped because other subsets of requirements cover it's issues.

B.11 [AP_2]: Not Realizable

Test run AP_2 has only inputs, not outputs. Since the inputs are uncontrollable the following properties are non-realizable.

Requirements

[label=]**[AP-005]:**
Autopilot shall always satisfy abs_roll_rate $<$ 6.6 **[AP-006]:**
Autopilot shall always satisfy abs_roll_angle $<$ 33.0

Table B6: [**AP_2**] Counter Example

Inputs	T = 0
phi	33
Local Variables	
abs_roll_rate	7.6
abs_roll_angle	33
Properties	
AP-005	False
AP-006	False
Outputs	
None	

B.12 [AP_4]: Realizable

Since the fixpoint algorithm is sound for realizable results the [AP004A] can be concluded realizable.

Requirements

[label=]**[AP-004A]:**
When in roll_hold mode, when steady_state AP shall always satisfy abs_roll_err $\leq$ 1.0

B.13 Roll Autopilot (RAP)

Run	FP Result	FP Time	KI Result	KI Time
RAP_0	Unrealizable	0.5s	Unrealizable	0.1s
RAP_1	Realizable	0.5s	Realizable	0.2s
RAP_2	Unrealizable	0.4s	Unrealizable	0.1s
RAP_3	Realizable	0.81s	Unknown	8s
RAP_4	Unrealizable	0.9s	Unrealizable	0.1
RAP_5	Unrealizable	0.3s	Unrealizable	0.1s
RAP_6	Realizable	0.3s	Unknown*	10h+
RAP_7	Realizable	0.6s	Unknown*	8s
RAP_8	Realizable	0.9	Unknown	1m 12s

Run Details:

- RAP_0: All properties
- RAP_1: Only [AP-001A]
- RAP_2: Only [AP-002A]
- RAP_3: Only [AP-002C]
- RAP_4: Only [AP-001A] and [AP-007] and [AP-008A] and [AP-00B]
- RAP_5: Only [AP-008B]
- RAP_6: Only [AP-008A]
- RAP_7: Only [AP-008A] and [AP-001]
- RAP_8: Only [AP-008A] and [AP-001] and [AP-007]

*Unknown due to crossing max_step or timeout.

Bibliography

1. Champion, A.; Gurfinkel, A.; Kahsai, T.; and Tinelli, C.: CoCoSpec: A Mode-Aware Contract Language for Reactive Systems. *Software Engineering and Formal Methods*, Springer International Publishing, 2016.

2. Halbwachs, N.; Caspi, P.; Raymond, P.; and Pilaud, D.: The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, Sept. 1991.

3. Gacek, A.; Backes, J.; Whalen, M.; Wagner, L.; and Ghassabani, E.: The JKind Model Checker. 2017.

4. 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. 2018.

5. Gacek, A.; Katis, A.; Whalen, M. W.; Backes, J.; and Cofer, D.: Towards Realizability Checking of Contracts using Theories. 2015.

6. Fedyukovich, G.; Gurnkel, A.; and Sharygina, N.: Automated Discovery of Simulation Between Programs.

7. Wahl, T.: The K-Induction Principle.

8. Lockheed Martin S5 V&V Challenges. URL `http://mys5.org/Proceedings/2016/Day_2/2016-S5-Day2_0945_Elliott.pdf`.

9. Z3 Tutorial. URL `https://rise4fun.com/z3/tutorial`.

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704–0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.
PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*
01-06-2019

2. REPORT TYPE
Technical Memorandum

3. DATES COVERED *(From - To)*

4. TITLE AND SUBTITLE
Integrating Realizability Checking in FRET

5a. CONTRACT NUMBER
NNA14AA60C

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
David Kooi, Anastasia Mavridou

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NASA Ames Research Center, Moffett Field, CA 94035
Onera, The French Aerospace Lab, FR

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–2019–220365

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

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

14. ABSTRACT
Realizability (alternatively referred to as *relative consistency* or *implementability*), i.e., checking whether a specification can be implemented by an open system, has been the subject of extensive study. Realizability can be viewed as a stronger analysis method when compared to consistency checking in that it does not only check whether the system works in *some* environment, but instead whether the system works in *all* environments. In this report, we experiment and build on the approach presented by Gacek et al. that supports checking realizability of contracts involving infinite theories using SMT solvers. In particular, we 1) extend the Formal Requirements Elicitation Tool (FRET) to support generation of Lustre contracts that can be checked for realizability with the JKind k-induction and fix point generation engines; 2) study a compositional way for checking realizability based on the notion of connected components; and 3) test our approach and the capabilities of the JKind realizability engines on a large subset of the Lockheed Martin Cyber Physical System Challenge Problems.

15. SUBJECT TERMS
realizability, requirements engineering, verification, JKind, Lustre

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