Towards Realizability Checking of Contracts using Theories

Andrew Gacek[1], Andreas Katis[2], Michael W. Whalen[2], John Backes[1], Darren Cofer[1]

[1] Rockwell Collins Advanced Technology Center
400 Collins Rd. NE, Cedar Rapids, IA, 52498, USA
{andrew.gacek,john.backes,darren.cofer}@rockwellcollins.com
[2] Department of Computer Science and Engineering,
University of Minnesota, 200 Union Street, Minneapolis, MN 55455,USA
katis001@umn.edu, whalen@cs.umn.edu

Abstract. *Virtual integration* techniques focus on building architectural models of systems that can be analyzed early in the design cycle to try to lower cost, reduce risk, and improve quality of complex embedded systems. Given appropriate architectural descriptions and compositional reasoning rules, these techniques can be used to prove important safety properties about the architecture prior to system construction. Such proofs build from "leaf-level" assume/guarantee component contracts through architectural layers towards top-level safety properties. The proofs are built upon the premise that each leaf-level component contract is *realizable*; i.e., it is possible to construct a component such that for any input allowed by the contract assumptions, there is some output value that the component can produce that satisfies the contract guarantees. Without engineering support it is all too easy to write leaf-level components that can't be realized. Realizability checking for propositional contracts has been well-studied for many years, both for component synthesis and checking correctness of temporal logic requirements. However, checking realizability for contracts involving infinite theories is still an open problem. In this paper, we describe a new approach for checking realizability of contracts involving theories and demonstrate its usefulness on several examples.

1 Introduction

In the recent years, *virtual integration* approaches have been proposed as a means to lower cost and improve quality of complex embedded systems. These approaches focus on building architectural models of systems that can be analyzed prior to construction of component implementations. The objective is to discover and resolve problems early during the design and implementation phases when cost impact is lower. Several architecture description languages such as AADL [1], SysML [2], and AUTOSAR [3] are designed to support such an engineering process, and there has been significant effort to analytically determine

system performance [4,5], fault tolerance [5], security [6], and safety [7] using these techniques.

In an ongoing effort at Rockwell Collins and The University of Minnesota, we have been pursuing virtual integration using compositional proofs of correctness. The idea is to support hierarchical design and analysis of complex system architectures and co-evolution of requirements and architectures at multiple levels of abstraction [8]. This was based on two observations about software development for commercial aircraft: first, that component-level errors are relatively rare and that most problems occur during integration [9], and second, that requirements specifications often contain significant numbers of omissions or errors [10] that are at the root of many of the integration problems. Specifically, the problem involves demonstrating *satisfaction arguments* [11], i.e., that the requirements allocated to components and the architecture connecting those components is sufficient to guarantee the system requirements. We have created the AGREE reasoning framework [12] to support compositional assume/guarantee contract reasoning over system architectural models written in AADL.

Such proof systems build from "leaf-level" assume/guarantee component contracts through architectural layers towards proofs of top-level safety properties. The soundness of the argument is built upon the premise that each leaf-level component contract is *realizable*; i.e., it is possible to construct a component such that for any input allowed by the contract assumptions, there is some output value that the component can produce that satisfies the contract guarantees.

Unfortunately, without engineering support it is all too easy to write leaf-level components that can't be realized. When applying our tools in both industrial and classroom settings, this issue has led to incorrect compositional "proofs" of systems; in fact the goal of producing a compositional proof can lead to engineers modifying component-level requirements such that they are no longer possible to implement. In order to make our virtual integration approach reasonable for practicing engineers, tool support must be provided to check whether components are *realizable*.

Realizability checking for propositional contracts has been well-studied for many years (e.g., [13,14,15,16]), both for component synthesis and checking correctness of temporal logic requirements. Checking realizability for contracts involving theories, on the other hand, is still an open problem. In this paper, we describe a new approach for checking realizability of contracts involving theories and demonstrate its usefulness on several examples. Our approach is similar to k-induction over quantified formulas. We describe two algorithms. The first is sound for both proofs and counterexamples, but computationally intractable. The second algorithm is not sound for counterexamples (i.e., it may return a 'false counterexample' to a problem that is in fact realizable), but we have found it fast and accurate in practice.

The rest of the paper is structured as follows. In Section 2 we will describe our motivation and an example to illustrate realizability, and will define realizability formally in Section 3. We next describe two algorithms for checking realizability in Section 4, our implementation in the AGREE tool suite in Section 5, and our

experience using the realizability check in Section 6. Section 7 describes related work and Section 8 concludes.

2 Motivation and Example

We have been pursuing a *proof-based virtual integration* approach for building complex systems using the architecture description language AADL [1] and the AGREE compositional reasoning system [12]. We have demonstrated the effectiveness of the approach on a variety of industrial-scale systems, including the software controller for a patient-controlled analgesia (PCA) infusion pump [17], a dual flight-guidance system [12], and several more recent models, such as a quad-redundant flight control system and a quadcopter control system. We are using this approach on the DARPA HACMS program to build secure vehicles and to demonstrate how to apply virtual integration on industrial scale systems to facilitate technology transfer.

As part of the HACMS project, we attempted a feasibility test via a classroom exercise. We used the AADL and AGREE tools in a class assignment in a graduate-level software architecture class. The students were organized into six teams of four students. Each team was asked to specify the control software for a simplified microwave oven in AADL using a virtual integration approach. The software was split into two subsystems: one for controlling the heating element and another for controlling the display panel, with several requirements for each subsystem. The goal was to formalize these component-level requirements and use them to prove three system-level safety requirements.

The results of the initial experiment were sobering. All student groups were able to prove the system-level requirements starting from formalizations of the component requirements. Unfortunately, in many cases, the proofs succeeded because the components were incorrectly specified. In fact, only one of the teams had written component-level requirements that could be implemented. The other teams had requirements which were inconsistent under certain input conditions. For example, one team produced the following informal component-level requirements:

> **Microwave-1** - While the microwave is in cooking mode, seconds_to_cook shall decrease.

> **Microwave-2** - If the display is quiescent (no buttons pressed) and the keypad is enabled, the seconds_to_cook shall not change.

and then produced the following formalized requirements[3]:

$$\textbf{guarantee}\colon \mathsf{is\_cooking}' \Rightarrow \mathsf{seconds\_to\_cook}' \leq \mathsf{seconds\_to\_cook} - 1$$

$$\textbf{guarantee}\colon (\neg \mathsf{any\_digit\_pressed} \wedge \mathsf{keypad\_enabled}) \Rightarrow \mathsf{seconds\_to\_cook}' = \mathsf{seconds\_to\_cook}$$

[3] We have translated this property and others from the higher level AGREE syntax into a two-state form that is used throughout this paper.

These formalized guarantees fail to avoid the conflict in the seconds_to_cook variable between the Microwave-1 and Microwave-2 requirements, as they cannot be both satisfied in a case where the microwave is cooking and the keypad is enabled. This error was not caught despite an analysis built into an early version of AGREE that checks contracts for *consistency*, i.e., whether the conjunction of a system's guarantees is satisfiable. We realized that consistency checking does not actually provide a trustworthy answer because it only checks whether the system works in *some* external environment, not in *all* environments. Realizability checking determines whether or not the component works in all input environments that satisfy the component assumptions.

From this experience, we decided that realizability checking was necessary for successful tech transfer of a virtual integration approach. The analysis was not only necessary for classroom settings. We also found problems with component-level requirements in two of our large-scale analysis efforts. Further, existing approaches for checking realizability do not allow predicates over infinite theories such as integers and reals, which are native to our AGREE contracts.

In the following sections, we formally define realizability over transition systems, as well as algorithms for checking realizability over infinite-state systems that are efficient and accurate in practice. A machine-checked formalization of the definitions and proofs in Coq can be found in a companion paper [18].

3 Realizability

We assume the types state and input for states and inputs. We use s for variables of type state and i for variables of type input. State represents both internal state and external outputs. A transition system is a pair (I, T) where $I : \mathsf{state} \to \mathsf{bool}$ holds on the initial states states and $T : \mathsf{state} \times \mathsf{input} \times \mathsf{state} \to \mathsf{bool}$ holds on $T(s, i, s')$ when the system can transition from state s to state s' on receipt of input i. We assume the usual notion of path with respect to a transition relation.

A contract specifies the desired behavior of a transition system. A contract is a pair (A, G) of an assumption and a guarantee. The assumption $A : \mathsf{state} \times \mathsf{input} \to \mathsf{bool}$ specifies for a given system state which inputs are valid. The guarantee G is a pair (G_I, G_T) of an initial guarantee and a transitional guarantee. The initial guarantee $G_I : \mathsf{state} \to \mathsf{bool}$ specifies which states the system may start in, that is, the possible initial internal state and external outputs. The transitional guarantee $G_T : \mathsf{state} \times \mathsf{input} \times \mathsf{state} \to \mathsf{bool}$ specifies for a given state and input what states the system may transition to.

We now define what it means for a transition system to realize a contract. This requires that the system respects the guarantee for inputs which satisfying the contract. Moreover, the system must always remain responsive with respect to inputs that satisfying the assumptions. In order to make this definition precise, we first need to define which system states are reachable given some assumptions on the system inputs.

Definition 1 (Reachable with respect to assumptions). *Let (I, T) be a transition system and let $A : \mathsf{state} \times \mathsf{input} \to \mathsf{bool}$ be an assumption. A state of (I, T) is reachable with respect to A if there exists a path starting in an initial*

state and eventually reaching s such that all transitions satisfying the assumptions. Formally, $\mathsf{Reachable}_A(s)$ *is defined inductively by*

$$\mathsf{Reachable}_A(s) = I(s) \vee \exists s_{\mathsf{prev}}, i.\ \mathsf{Reachable}_A(s_{\mathsf{prev}}) \wedge A(s_{\mathsf{prev}}, i) \wedge T(s_{\mathsf{prev}}, i, s)$$

Definition 2 (Realization). *A transition system (I, T) is a realization of the contract $(A, (G_I, G_T))$ when the following conditions hold*

1. $\forall s.\ I(s) \Rightarrow G_I(s)$
2. $\forall s, i, s'.\ \mathsf{Reachable}_A(s) \wedge A(s, i) \wedge T(s, i, s') \Rightarrow G_T(s, i, s')$
3. $\exists s.\ I(s)$
4. $\forall s, i.\ \mathsf{Reachable}_A(s) \wedge A(s, i) \Rightarrow \exists s'.\ T(s, i, s')$

The first two conditions in Definition 2 ensure that the transition system respects the guarantees. The second two conditions ensure that the system is non-trivial and responsive to all valid inputs.

Definition 3 (Realizable). *A contract is realizable if there exists a transition system which is a realization of the contract.*

Definitions 2 and 3 are useful for directly defining realizability, but not very useful for checking realizability. We now develop an equivalent notion which is more suggestive and amenable to checking. This is based on a notion called *viability*. Intuitively, a state is viable with respect to a contract if being in that state does not doom a realization to failure. We can capture this notion without reference to any specific realization, because condition 2 in the definition of realization tells us that G_T is an over-approximation of any T.

Definition 4 (Viable). *A state s is* viable *with respect to a contract $(A, (G_I, G_T))$, written* $\mathsf{Viable}(s)$*, if G_T can keep responding to valid inputs forever, starting from s. Informally, one can say that a state s is viable if it satisfies the infinite formula:*

$$\forall i_1.\ A(s, i_1) \Rightarrow \exists s_1.\ G_T(s, i_1, s_1) \wedge \forall i_2.\ A(s_1, i_2) \Rightarrow \exists s_2.\ G_T(s_1, i_2, s_2) \wedge \forall i_3.\ \cdots$$

Formally, viability is defined coinductively by the following equation

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

Theorem 1 (Alternative realizability). *A contract $(A, (G_I, G_T))$ is realizable if and only if $\exists s.\ G_I(s) \wedge \mathsf{Viable}(s)$.*

Proof. For the "only if" direction the key lemma is $\forall s.\ \mathsf{Reachable}_A(s) \Rightarrow \mathsf{Viable}(s)$. This lemma is proved by coinduction and follows directly from conditions 2 and 4 of Definition 2. Then by conditions 1 and 3 we have some state s such that $I(s)$ and $G_I(s)$. Thus $\mathsf{Reachable}_A(s)$ holds and applying the lemma we get $G_I(s) \wedge \mathsf{Viable}(s)$.

For the "if" direction, let s_0 be such that $G_I(s_0)$ and $\mathsf{Viable}(s_0)$. Define $I(s) = (s = s_0)$ and $T(s, i, s') = G_T(s, i, s') \wedge \mathsf{Viable}(s')$. Conditions 1, 2, and 3 of Definition 2 are clearly satisfied. Condition 4 follows from the observation that $\forall s.\ \mathsf{Reachable}_A(s) \Rightarrow \mathsf{Viable}(s)$ and from the definition of viability.

4 An Algorithm for Checking Realizability

In this section we develop two versions of an algorithm for automatically checking the realizability of a contract. The first version is based on Theorem 1 together with under- and over-approximations of viability. An over-approximation is useful to show that a contract is not viable, while an under-approximation is useful to show that a contract is viable. The second version of the algorithm follows from the mitigating the intractability of the first version.

We first define an over-approximation of viability called *finite viability* based on a finite unrolling of the definition of viability. Because this is an over-approximation, if a contract does not have an initial state which is finitely viable, then the contract is not viable. We formalize this when we prove the correctness of the realizability algorithm.

Definition 5 (Finite viability). *A state s is viable for n steps, written* $\mathsf{Viable}_n(s)$ *if G_T can keep responding to valid inputs for at least n steps. That is,*

$$\begin{aligned}\forall i_1.\ A(s,i_1) \Rightarrow \exists s_1.\ G_T(s,i_1,s_1) \wedge \\ \forall i_2.\ A(s_1,i_2) \Rightarrow \exists s_2.\ G_T(s_1,i_2,s_2) \wedge \cdots \wedge \\ \forall i_n.\ A(s_{n-1},i_n) \Rightarrow \exists s_n.\ G_T(s_{n-1},i_n,s_n)\end{aligned}$$

All states are viable for 0 steps.

We next define an under-approximation of viability based on *one-step extension*. This notion looks if G_T can respond to valid inputs given a finite historical trace of valid inputs and states.

Definition 6 (One-step extension). *A state s is extendable after n steps, written* $\mathsf{Extend}_n(s)$*, if any valid path of length n from s can be extended in response to any input. That is,*

$$\begin{aligned}\forall i_1, s_1, \ldots, i_n, s_n.\ \\ A(s,i_1) \wedge G_T(s,i_1,s_1) \wedge \cdots \wedge A(s_{n-1},i_n) \wedge G_T(s_{n-1},i_n,s_n) \Rightarrow \\ \forall i.\ A(s_n,i) \Rightarrow \exists s'.\ G_T(s_n,i,s')\end{aligned}$$

We now use these two notions to formally define our realizability algorithm. The core of the algorithm is based on two checks called the *base* and *extend* check.

Definition 7 (Realizability Algorithm). *Define the checks:*

$$\begin{aligned}\mathsf{BaseCheck}(n) &= \exists s.\ G_I(s) \wedge \mathsf{Viable}_n(s) \\ \mathsf{ExtendCheck}(n) &= \forall s.\ \mathsf{Extend}_n(s)\end{aligned}$$

The following algorithm checks for realizability or unrealizability of a contract.

for $n = 0$ *to* ∞ ***do***
  if not $\mathsf{BaseCheck}(n)$ ***then***
    return *"unrealizable"*
  else if $\mathsf{ExtendCheck}(n)$ ***then***
    return *"realizable"*
  end if
end for

Theorem 2 (Soundness of "unrealizable" result). *If* $\exists n.\ \neg\mathsf{BaseCheck}(n)$ *then the contract is not realizable.*

Proof. First we show $\forall s, n.\ \mathsf{Viable}(s) \Rightarrow \mathsf{Viable}_n(s)$ by induction on n. The result then follows from Theorem 1.

Theorem 3 (Soundness of "realizable" result). *If* $\exists n.\ \mathsf{BaseCheck}(n) \wedge \mathsf{ExtendCheck}(n)$ *then contract is realizable.*

Proof. First we show how $\mathsf{Extend}_n(s)$ can be used to shift $\mathsf{Viable}_n(s)$ forward. The following is proved by induction on n.

$$\forall s, n, i.\ \mathsf{Extend}_n(s) \wedge \mathsf{Viable}_n(s) \wedge A(s,i) \Rightarrow \exists s'.\ G_T(s,i,s') \wedge \mathsf{Viable}_n(s')$$

Using this lemma we can show the following by coinduction.

$$\forall s, n.\ \mathsf{Viable}_n(s) \wedge \mathsf{ExtendCheck}(n) \Rightarrow \mathsf{Viable}(s)$$

The result then follows from Theorem 1.

Corollary 1 (Soundness of Realizability Algorithm). *The Realizability Algorithm is sound.*

Due to the approximations used to define the base and extends check, the algorithm is incomplete. The following two examples show how both realizable and unrealizable contracts may send the algorithm into an infinite loop.

Example 1 (Incompleteness of "realizable" result). Suppose the type *state* is integers. Consider the contract:

$$A(s,i) = \top \qquad G_I(s) = \top \qquad G_T(s,i,s') = (s \neq 0)$$

This contract is realizable by, for example, a system that starts in state 1 and always transitions into the same state. Yet, for all n, $\mathsf{ExtendCheck}(n)$ fails since one can take a path of length n which ends at state 0. This path cannot be extended.

Example 2 (Incompleteness of "unrealizable" result). Suppose the type *state* is integers. Consider the contract:

$$A(s,i) = \top \qquad G_I(s) = (s \geq 0) \qquad G_T(s,i,s') = (s' = s - 1 \wedge s' \geq 0)$$

This contract is not realizable since in any realization the state 0 would be reachable, but the contract does not allow a transition from state 0. However, $\mathsf{BaseCheck}(n)$ holds for all n by starting in state $s = n$.

Implementing this algorithm requires a way of automatically checking the formulas $\mathsf{BaseCheck}(n)$ and $\mathsf{ExtendCheck}(n)$ for validity. This can be done in an SMT-solver that supports quantifiers over the language the contract is expressed in. Checking $\mathsf{ExtendCheck}(n)$ is rather nice in this setting since it has only a single quantifier alternation. Moreover, using an incremental SMT-solver one can reuse much of the work done to check $\mathsf{ExtendCheck}(n)$ to also check $\mathsf{ExtendCheck}(n+1)$. However, $\mathsf{BaseCheck}(n)$ is problematic. First, it has $2n$ quantifier alternations which puts even small cases outside the reach of modern SMT-solvers. Second, the quantifiers make it impractical to reuse the results of $\mathsf{BaseCheck}(n)$ in checking $\mathsf{BaseCheck}(n+1)$. Finally, due to the quantifiers, a counterexample to $\mathsf{BaseCheck}(n)$ would be difficult to relay back to the user. Thus we need a simplification of $\mathsf{BaseCheck}(n)$ in order to make our algorithm practical.

Definition 8 (Simplified base check). *Define a simplified base check which checks that any path of length n from an initial state can be extended one step.*

$$\mathsf{BaseCheck}'(n) = \forall s.\ G_I(s) \Rightarrow \mathsf{Extend}_n(s)$$

First, note that this check has a single quantifier alternation. Second, this check can leverage the incremental features in an SMT-solver to use the results of $\mathsf{BaseCheck}'(n)$ in checking $\mathsf{BaseCheck}'(n+1)$. Finally, when this check fails it can return a counterexample which is a trace of a system realizing the contract for n steps, but then becoming stuck. This provides very concrete and useful feedback to system developers. The correctness of this check is captured by the following theorem.

Theorem 4 (One-way soundness of simplified base check).

$$(\exists s.\ G_I(s)) \Rightarrow \forall n.\ (\forall k \leq n.\ \mathsf{BaseCheck}'(k)) \Rightarrow \mathsf{BaseCheck}(n)$$

Proof. We first prove the following by induction on n:

$$\forall s, n.\ \mathsf{Extend}_n(s) \wedge \mathsf{Viable}_n(s) \Rightarrow \mathsf{Viable}_{n+1}(s)$$

The final result follows using this and induction on n.

Thus replacing $\mathsf{BaseCheck}(n)$ in the realizability algorithm with $\mathsf{BaseCheck}'(n)$ preserves soundness of the "realizability" result. However, because the implication in Theorem 4 is only in one direction, the algorithm is no longer sound for the "unrealizable" result. That is, it may return a counterexample showing n steps of a realization of the contract that gets into a stuck state. The following example makes this point explicit.

Example 3. Consider again Example 1 where the type *state* is integers and the contract is:

$$A(s,i) = \top \qquad G_I(s) = \top \qquad G_T(s,i,s') = (s \neq 0)$$

As before, this contract is easily realizable. However, $\mathsf{BaseCheck}'(n)$ fails for all n since it will consider a path starting at state n and transitioning n steps to state 0 where no more transitions are possible.

The benefits of this second version of the algorithm outweigh its costs. The cases where a contract is realizable, yet fails the modified base check seems unlikely in practice. We have encountered none in our case studies. Moreover, when a contract does spuriously fail the simplified base check, it can almost always be rewritten into a form which would pass.

5 Implementation

We have built an implementation of the realizability algorithm as an extension to JKind [19], a re-implementation of the KIND model checker [20] in Java. Our tool is called JRealizability and is packaged with the latest release of JKind. The model's behavior is described in the Lustre language, which is the native input language of JKind and is used as an intermediate language for the AGREE tool suite.

We unroll the transition relation defined by the Lustre model into SMT problems (one for the base check and another for the extend check) which can be solved in parallel. We use the SMT-LIB Version 2 format which most modern SMT solvers support. The most significant issue for SMT solvers involves quantifier support, so we use the Z3 SMT solver [21] which has good support for reasoning over quantifiers and incremental search. The tool is often able to provide an answer for models containing integer and real-valued variables very quickly (in less than a second). Because of the use of quantifiers over a range of theories, it is possible that for one of the checks, Z3 returns `unknown`; in this case, we discontinue analysis. In addition, because our realizability check is incomplete, the tool terminates analysis when either a timeout or a user-specified max unrolling depth (default: 200) is reached. In this case we are able to report how far the base check reached which may provide some confidence in the realizability of the system.

6 Case Studies

As a part of testing the algorithm in actual components, we examined three different cases: a quad-redundant flight control system, a medical infusion pump, and a simple microwave controller. In this section, we provide a brief description of each case study and summarize the results in Table 1 at the end of the section.

6.1 Quad-Redundant Flight Control System

We ran our realizability analysis on a Quad-Redundant Flight Control System (QFCS) for NASA's Transport Class Model (TCM) aircraft simulation. We were provided with a set of English language requirements for the QFCS components and a description of the architecture. We modeled the architecture in AADL and the component requirements as assume/guarantee contracts in AGREE. As the name suggests, the QFCS consists of four redundant Flight Control Computers (FCCs). Each FCC contains components for handling faults and computing actuator signal values. One of these components is the Output Signal Analysis and Selection component (OSAS). The OSAS component is responsible for determining the output gain for signals coming from the control laws and going

to the actuators. The output signal gain is determined based on the number of other faulty FCCs or based on failures within the FCC containing the OSAS component. The OSAS component contains 17 English language requirements including the following:

OSAS-S-170 – If the local Cross Channel Data Link (CCDL) has failed, OSAS shall set the local actuator command gain to 1 (one).

OSAS-S-240 – If OSAS has been declared failed by CCDL, OSAS shall set the actuator command gain to 0 (zero).

We formalized these requirements using the following guarantees:

$$\textbf{guarantee}\colon \mathsf{ccdl\_failed} \Rightarrow (\mathsf{fcc\_gain}' = 1)$$
$$\textbf{guarantee}\colon \mathsf{osas\_failed} \Rightarrow (\mathsf{fcc\_gain}' = 0)$$

These guarantees are contradictory in the case when the local CCDL has failed and the local CCDL reports to the OSAS that the OSAS has failed. This error eluded the engineers who originally drafted the requirements as well as the engineers who formalized them. In this case, there should be an assumption that if the CCDL has failed then it will not report to the OSAS that the OSAS has failed. This was not part of the original requirements. However, AGREE's realizability analysis was able to identify the error and provide a counterexample.

6.2 Medical Device Example

Our realizability tool was also used to verify the realizability of the components in the Generic Patient Controlled Analgesia infusion pump system that was described in [22]. The controller consists of six subcomponents that were given as input for the tool to verify the requirements described inside. While five of the models were proven to be realizable, a subtly incorrect requirement definition was found in the contract for the controller's infusion manager.

GPCA-1 - The mode range of the controller shall be one of nine different modes. If the controller is in one of the first two modes the commanded flow rate shall be zero.

$$\begin{aligned}
&\textbf{guarantee}\colon \\
&\quad (\mathsf{IM_OUT.Current_System_Mode}' \geq 0) \ \wedge \\
&\quad (\mathsf{IM_OUT.Current_System_Mode}' \leq 8) \ \wedge \\
&\quad (\mathsf{IM_OUT.Current_System_Mode}' = 0 \Rightarrow \\
&\qquad \mathsf{IM_OUT.Commanded_Flow_Rate}' = 0) \wedge \\
&\quad (\mathsf{IM_OUT.Current_System_Mode}' = 1 \Rightarrow \\
&\qquad \mathsf{IM_OUT.Commanded_Flow_Rate}' = 0)
\end{aligned} \tag{1}$$

GPCA-2 - Whenever the alarm subsystem has detected a high severity hazard, then Infusion Manager shall never infuse drug at a rate more than the specified Keep Vein Open rate.

$$\begin{aligned}&\textbf{guarantee}:\\&\quad(\mathsf{TLM\_MODE\_IN.System\_On}' \wedge\\&\qquad \mathsf{ALARM\_IN.Highest\_Level\_Alarm}' = 3) \Rightarrow\\&\quad(\mathsf{IM\_OUT.Commanded\_Flow\_Rate}' = \mathsf{CONFIG\_IN.Flow\_Rate\_KVO}')\end{aligned} \tag{2}$$

The erroneously defined guarantee (2) tries to assert that the IM_OUT.Commanded_Flow_Rate to some (potentially non-zero) Flow_Rate_KVO if the alarm input is 3; however, this may occur when the IM_OUT.Current_System_Mode is computed to be zero or one, in which case the flow rate is commanded to be 0. While discovering and fixing the problem was not difficult, the error was not discovered by the regular consistency check in AGREE.

6.3 Microwave Assignment

The realizability tool was used to check the contracts for the microwave models produced by the graduate student teams described in Section 2 that provided the initial motivation for this work. The microwave consists of two subsystems that manage the cooking element and display panel of the device. Table 1 shows the corresponding results for each team, named as MT1, MT2, etc. While every team but one managed to provide an implementable set of requirements for the microwave's mode controller, there were several interesting cases involving the display control component. For space reasons, we highlight only one here.

Microwave-1 - While the microwave is in cooking mode, seconds_to_cook shall decrease.

Microwave-3 - When the keypad is initially enabled, if no digits are pressed, the value shall be zero.

Team 6 formalized these requirements as

$$\textbf{guarantee}: (\mathsf{cooking\_mode}' = 2) \Rightarrow (\mathsf{seconds\_to\_cook}' = \mathsf{seconds\_to\_cook} - 1)$$

$$\begin{aligned}&\textbf{guarantee}: (\neg\mathsf{keypad\_enabled} \wedge \mathsf{keypad\_enabled}' \wedge \neg\mathsf{any\_digit\_pressed}') \Rightarrow\\&\quad(\mathsf{seconds\_to\_cook}' = 0)\end{aligned}$$

In the counterexample provided, the state where the microwave is cooking (cooking_mode = 2) and no digit is pressed creates a conflict regarding which value is assigned to the seconds_to_cook variable: should it decrease by one, or be assigned to zero? This counterexample is interesting because it indicates a missing assumption on the environment: the keypad is not enabled when the

cooking mode is 2 (cooking). Without this assumption about the inputs, the guarantees are not realizable.

Case study	Model	Result	Time elapsed (seconds)	Base check depth (# of steps)
QFCS	FCS	realizable	1.762	0
QFCS	FCC	unrealizable	0.981	1
GPCA	Infusion Manager	unrealizable	0.2	1
GPCA	Alarm	realizable	0.316	0
GPCA	Config	realizable	0.102	0
GPCA	OutputBus	realizable	0.201	0
GPCA	System_Status	realizable	0.203	0
GPCA	Top_Level	realizable	0.103	0
MT 1	Mode Control	realizable	0.229	0
MT 1	Display Control	unrealizable	0.207	1
MT 2	Mode Control	realizable	0.202	0
MT 2	Display Control	unknown	1000 (tool timeout)	1
MT 3	Mode Control	realizable	0.203	0
MT 3	Display Control	unrealizable	0.202	1
MT 4	Mode Control	realizable	0.202	0
MT 4	Display Control	unrealizable	0.521	1
MT 5	Mode Control	unrealizable	0.1	1
MT 5	Display Control	unrealizable	0.222	1
MT 6	Mode Control	realizable	0.201	0
MT 6	Display Control	unknown	1000 (tool timeout)	1

Table 1. Realizability checking results for case studies

Table 1 contains the exact results that were obtained during the three case studies. Every "realizable" result was determined to be correct since an implementation was produced for each of the components analyzed, ensuring the accuracy of the tool. Every contract that was identified as "unrealizable" was manually confirmed to be unrealizable, i.e., there were no spurious results. Additionally, the number of steps that the base check required to provide a final answer was not more than one, with the unknown results being particularly interesting, as the tool timed out before the solver was able to provide a concrete answer. This shows that there is still work to be done in terms of the algorithm's scalability, as well as an efficient way to eliminate quantifiers, making the solving process easier for Z3.

7 Related Work

The idea of *realizability* has been the subject of intensive study. Gunter et al. refer to it using the term *relative consistency* in [23], while Pnuelli and Rosner use the term *implementability* in [13] to refer to the problem of synthesis for propositional LTL. Additionally, the authors in [13] proved that the lower-bound time complexity of the problem is doubly exponential, in the worst case. In the following years, several techniques were introduced to deal with the synthesis problem in a more efficient way for subsets of propositional LTL [24], simple

LTL formulas ([14], [25]), as well as in a component-based approach [16] and specifications based on other temporal logics ([26], [15]), such as SIS [27]. Finally, an interesting and relevant work has been done regarding the solution to the controllability problem using in [28] [29] and [30], which involves the decision on the existence a strategy that assigns certain values to a set of controllable activities, with respect to a set of uncontrollable ones.

Recent work in solving infinite game problems [31,32] can be specialized to the problem of realizability. In this work, the authors describe a framework for analyzing arbitrary two-player games. To provide proofs within the framework, *template formulas* must be provided by the user that describe the shape of a Skolem function that is used to explicitly define an inductive invariant that demonstrates the realizability of a model. Although this work is more general than ours, the applicability of the approach requires user-provided templates that are problem specific, so is not entirely automated.

The main contribution of our work is that it automatically checks the realizability of infinite domain systems. The problem is, in general, undecidable. Still, the application of bounded model checking can still offer an approximate answer to the realizability problem as we experienced by the fact that Z3 managed to solve the majority of our test models.

8 Conclusions and Future Work

In this paper, we have presented a new approach for determining realizability of contracts involving infinite theories using SMT solvers. This approach allows analysis of a class of contracts that were previously not solvable using automated analysis. The approach is both incomplete and conservative, i.e., it may return "false positive" results, declaring that a contract is not realizable when it could be realized. However, it has been shown to be both fast and effective in practice on a variety of models.

The results of this paper provide a good foundation towards further research in realizability. In much the same way that many properties are not *inductive*, some contracts cannot be proven realizable using one step extensions. We are examining alternate algorithms, similar to approaches such as IC3 [33], which support property-directed invariant generation, to improve the approach presented here. However, this requires generalizing the IC3 approach to solve quantified formulas (as well as to generalize counterexamples over quantified formulas). We hope to demonstrate an approach involving a IC3-like algorithm in the near future.

In addition, for realizable systems, it is likely that we want to consider the *synthesis* problem, which we have not explicitly considered in this paper. Synthesis aims to construct a concrete implementation of the contract, rather than determine its existence. It is known for propositional systems that the synthesis problem is equivalent in complexity to the realizability problem [13], but it is not known (to us) whether this equivalence is true in the infinite-state case.

Acknowledgments. This work was funded by DARPA and AFRL under contract FA8750-12-9-0179 (Secure Mathematically-Assured Composition of Con-

trol Models), and by NASA under contract NNA13AA21C (Compositional Verification of Flight Critical Systems), and by NSF under grant CNS-1035715 (Assuring the safety, security, and reliability of medical device cyber physical systems).

References

1. SAE-AS5506: Architecture Analysis and Design Language. SAE (2004)
2. Friedenthal, S., Moore, A., Steiner, R.: A Practical Guide to SysML: Systems Modeling Language. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2008)
3. Consortium, A.: Automotive Open System Architecture (AUTOSAR) Revision 4.2.1. AUTOSAR (2014)
4. Varona-Gomez, R., Villar, E.: Aadl simulation and performance analysis in systemc. In: Engineering of Complex Computer Systems, 2009 14th IEEE International Conference on. (2009) 323–328
5. Bozzano, M., Cimatti, A., Katoen, J.P., Nguyen, V.Y., Noll, T., Roveri, M.: Safety, dependability and performance analysis of extended aadl models. Comput. J. **54** (2011) 754–775
6. Apvrille, L., Roudier, Y.: SysML-Sec: A model-driven environment for developing secure embedded systems. In: SAR-SSI 2013, 8ème Conférence sur la Sécurité des Architectures Réseaux et des Systèmes d'Information, 16-18 Septembre 2013, Mont-de-Marsan, France, Mont-de-Marsan, FRANCE (2013)
7. Bozzano, M., Cimatti, A., Katoen, J.P., Katsaros, P., Mokos, K., Nguyen, V.Y., Noll, T., Postma, B., Roveri, M.: Spacecraft early design validation using formal methods. Reliability Engineering and System Safety **132** (2014)
8. Whalen, M.W., Gacek, A., Cofer, D., Murugesan, A., Heimdahl, M.P., Rayadurgam, S.: Your what is my how: Iteration and hierarchy in system design. Software, IEEE **30** (2013) 54–60
9. Rushby, J.: New challenges in certification for aircraft software. In: Proceedings of the ninth ACM Int'l Conf. on Embedded software, ACM (2011) 211–218
10. Miller, S.P., Tribble, A.C., Whalen, M.W., Heimdahl, M.P.E.: Proving the shalls: Early validation of requirements through formal methods. Int. J. Softw. Tools Technol. Transf. **8** (2006) 303–319
11. Hammond, J., Rawlings, R., Hall, A.: Will it work? [requirements engineering]. In: Requirements Engineering, 2001. Proceedings. Fifth IEEE Int'l Symposium on. (2001) 102 –109
12. Cofer, D.D., Gacek, A., Miller, S.P., Whalen, M.W., LaValley, B., Sha, L.: Compositional verification of architectural models. In Goodloe, A.E., Person, S., eds.: Proceedings of the 4th NASA Formal Methods Symposium (NFM 2012). Volume 7226., Berlin, Heidelberg, Springer-Verlag (2012) 126–140
13. Pnueli, A., Rosner, R.: On the Synthesis of a Reactive Module. Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL'89) (1989) 179–190
14. Bohy, A., Bruyre, V., Filiot, E., Jin, N., Raskin, J.F.: Acacia+, a tool for LTL Synthesis. Proceedings of the 24th International Conference on Computer Aided Verification (CAV'12) (2012) 652–657
15. Hamza, J., Jobstmann, B., Kuncak, V.: Synthesis for Regular Specifications over Unbounded Domains. Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design (2010) 101–109

16. Chatterjee, K., Henzinger, T.A.: Assume-Guarantee Synthesis. Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07) (2007) 261–275
17. Murugesan, A., Whalen, M.W., Rayadurgam, S., Heimdahl, M.P.: Compositional verification of a medical device system. In: ACM Int'l Conf. on High Integrity Language Technology (HILT) 2013, ACM (2013)
18. Katis, A., Gacek, A., Whalen, M.W.: Machine-checked proofs for realizability checking algorithms (2015) Submitted `http://arxiv.org/abs/1502.01292`.
19. Gacek, A.: JKind - a Java implementation of the KIND model checker. `https://github.com/agacek/jkind` (2014)
20. Hagen, G.: Verifying safety properties of Lustre programs: an SMT-based approach. PhD thesis, University of Iowa (2008)
21. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems. Springer (2008) 337–340
22. Murugesan, A., Sokolsky, O., Rayadurgam, S., Whalen, M., Heimdahl, M., Lee, I.: Linking Abstract Analysis to Concrete Design: A Hierarchical Approach to Verify Medical CPS Safety. Proceedings of ICCPS'14 (2014)
23. Gunter, C.A., Gunter, E.L., Jackson, M., Zave, P.: A Reference model for Requirements and Specifications. IEEE Software **17** (2000) 37–43
24. Klein, U., Pnueli, A.: Revisiting Synthesis of GR(1) Specifications. Proceedings of the 6th International Conference on Hardware and Software: Verification and Testing (HVC'10) (2010) 161–181
25. Tini, S., Maggiolo-Schettini, A.: Compositional Synthesis of Generalized Mealy Machines. Fundamenta Informaticae **60** (2003) 367–382
26. Bene, N., ern, I., tefak, F.: Factorization for Component-Interaction Automata. Proceedings of the 38th International Conference on Current Trends in Theory and Practice of Computer Science (2012) 554–565
27. Aziz, A., Balarin, F., Braton, R., Sangiovanni-Vincentelli, A.: Sequential Synthesis using SIS. Proceedings of the 1995 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'95) (1995) 612–617
28. Cimatti, A., Micheli, A., Roveri, M.: Solving temporal problems using SMT: Weak controllability. In: AAAI. (2012) 448–454
29. Cimatti, A., Micheli, A., Roveri, M.: Solving temporal problems using SMT: Strong controllability. In: CP. (2012) 248–264
30. Cimatti, A., Micheli, A., Roveri, M.: Solving strong controllability of temporal problems with uncertainty using SMT. Constraints (2014)
31. Beyene, T., Chaudhuri, S., Popeea, C., Rybalchenko, A.: A constraint-based approach to solving games on infinite graphs. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '14, New York, NY, USA, ACM (2014) 221–233
32. Beyene, T.A., Popeea, C., Rybalchenko, A.: Solving existentially quantified Horn clauses. In: Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings. (2013) 869–882
33. Bradley, A.: SAT-based model checking without unrolling. VMCAI (2011)