

Creating Better System Models: A Method for Using Compositional Reasoning to Validate Architectures with Assumption/Guarantee Contracts

Isaac Amundson
Collins Aerospace
Minneapolis, Minnesota, U.S.
Isaac.Amundson@collins.com

Josh Kahn
The MathWorks Inc.
Novi, Michigan, U.S.
joshkahn@mathworks.com

Vidya Srinivasan
The MathWorks Inc.
Natick, Massachusetts, U.S.
vsriniva@mathworks.com

Gopal Narayan Rai
Collins Aerospace
Hyderabad, Telangana, India
Gopal.Narayan.Rai@collins.com

Janet Liu
Collins Aerospace
Cedar Rapids, Iowa, U.S.
Janet.Liu@collins.com

Abstract. Formal methods have proved to be a valuable tool for identifying defects early in the development of safety-critical systems. Despite that, several factors have impeded their adoption within the systems engineering community. Some of these include lack of commercially available solutions, poor integration of analysis functionality in existing model-based systems engineering (MBSE) tools, and difficulty interpreting the results of the formal analyses. One such analysis that is popular among pockets within the aerospace community is the Assume Guarantee Reasoning Environment (AGREE), which analyzes Architecture Analysis and Design Language (AADL) models. AGREE is an open-source property-proving model checker that uses compositional reasoning to prove the system composition is valid based on assumptions and guarantees associated with the system components. The goals of this work are to develop a method for using AGREE in a more widely adopted commercially available tool and to take advantage of MBSE formalisms to better convey the analysis results, especially counterexamples. The hope is that this will increase the use of formal methods by high-assurance systems developers.

Keywords. Compositional Reasoning, Architecture Modeling, System Composer, AADL, AGREE, Assume-Guarantee Reasoning, Architecture Analysis

Introduction

Aerospace and defense systems rely on complex collections of distributed real-time embedded software. This software is critical to system safety and presents many challenges for the organizations that develop it. Model-based design tools are commonly used to implement high-assurance software functions, and formal analysis of behavioral models and executable code is gaining traction in some industry sectors. Unfortunately, system-level design tools for specifying and verifying a system architecture, including interactions of distributed components, resource allocation decisions, and communication mechanisms, are less mature. This has made it challenging to apply formal analyses effectively at the system level.

There are several other factors that compound matters and impede the uptake of formal assurance technologies in general. These include unintuitive user interfaces, scalability limitations, and few options for commercially licensed tools and vendor support (Davis, et al., 2013). Another reason for the slow adoption can be inferred from research, such as McMillan et al. (2024), in which activity and state machine behaviors were used for requirements modeling on a Systems Modeling Language (SysML) (Object Management Group, 2024) model, organized into tables, exported in Comma Separated Value (CSV) format, converted to another semantic language, input into another tool for analysis, and the results then stored in yet another tool. This highlights the lack of support in existing Model-Based Systems Engineering (MBSE) tools for natively handling analyses with one notable quote being, “The first issue is that SysML tools are not well suited for accepting evidence about analysis. While Cameo and other MBSE tools typically allow enough customizations to make this work, it is not a minor undertaking.” This underscores the need for more tightly coupled analysis capabilities in modern MBSE tools.

We aim to promote a formal system modeling methodology that incorporates existing practices and artifacts compatible with tools and processes used in industry, thereby facilitating greater adoption. The work in this paper directly addresses this goal by using a popular modeling framework that accurately captures a system architecture and supports formal analysis. We deeply embed formal verification into the design process to enable correct-by-construction system development that works correctly the first time. Using an architecture modeling language with a well-defined execution semantics, design patterns that provide formally guaranteed properties, and components with formally specified contracts ensures that the system design will meet its requirements even before implementation.

Specifically, we focus on verifying system architecture composition by annotating components with formal assume-guarantee contracts, where “assumptions” describe a component’s expectations of the environment and “guarantees” describe bounds on the behavior of the component when the assumptions are valid. We then apply a model checker to prove the correctness of (1) component interfaces (i.e., the output guarantees of each component must be strong enough to satisfy the input assumptions of downstream components), and (2) component implementations, (i.e., the input assumptions of a system and the output guarantees of its subcomponents must be strong enough to satisfy its output guarantees). Compositional reasoning enables scalable formal verification by splitting a complex system analysis into a collection of verification tasks corresponding to the architecture’s hierarchical structure. By decomposing the verification effort into proofs about each subsystem within the architecture, the analysis can be scaled to very large system designs (Murugesan, Whalen, Rayadurgam, & Heimdahl, 2013).

Previous efforts at developing an MBSE compositional reasoning tool resulted in the Assume Guarantee Reasoning Environment (AGREE) (Cofer, et al., 2012), developed as an annex to the Architecture Analysis and Design Language (AADL) (SAE International, 2022). However, due to the reasons described above, AADL and AGREE have yet to be widely adopted in real-world product development efforts despite having

seen moderate use by the formal methods research community. By bringing compositional reasoning to a more widely used end-to-end model-based design framework, we hope to increase access to this powerful analysis method and improve the dependability of the high-assurance products that have become increasingly essential in our daily lives.

The contributions of this work are as follows:

- We apply compositional reasoning, as inspired by AGREE, in a widely used MBSE framework, Systems Composer™ (MathWorks, 2024), and demonstrate how to annotate system model components with formal behavioral contracts that are traced to system requirements.
- We develop a MATLAB® Toolbox (MathWorks, 2024) for System Composer that enables assume-guarantee style compositional reasoning.
- We present our approach for generating explainable counterexamples corresponding to the formal analysis results.
- We provide case studies demonstrating compositional reasoning in System Composer and compare our results with semantically equivalent AADL/AGREE models.

Background and Related Work

Before the introduction of standardized system modeling languages, and still common today, system definitions existed in various forms such as textual documents, Microsoft® PowerPoint® (Microsoft, 2024) presentations, Microsoft Visio® (Microsoft, 2024) diagrams, or even as objects in relational databases such as IBM DOORS (IBM, 2024). Standardized modeling languages were introduced to facilitate integration, analysis, and system architecture reuse. One such language is AADL, a specialized extensible system modeling language for describing real-time embedded safety-critical systems. The introduction of AADL was shortly followed by a common standard known as the Systems Modeling Language (SysML) (Object Management Group, 2024). SysML was first and foremost a graphical language for describing system models, providing a standardized format derived from the popular software-oriented Unified Modeling Language (UML) (Object Management Group, 2017). The goal was to align systems engineers on an unambiguous design representation (da Silva, Linhares, Padilha, Roqueiro, & de Oliveira, 2006). SysML has seen broader adoption across multiple industries compared to AADL despite being published a few years later. In lieu of that, AADL was ahead of its time, with its elements having actual semantic meaning in the execution domain and support for formal analyses not found in more generic languages. This allowed for a more complete specification of hardware and software architectures for embedded systems.

The emergence of formal systems modeling languages has led to the development of numerous authoring tools that are fundamentally model based. Among the popular tools are Cameo Systems Modeler (Dassault Systemes, 2024), recognized for its support of SysML and integration capabilities; Enterprise Architect (Sparx Systems, 2024), offering modeling solutions for UML, SysML, and Business Process Model and Notation (BPMN) (Object Management Group, 2014); Safety-Critical Application Development Environment (SCADE) Suite (Ansys, 2024), frequently used in the aerospace and automotive industries; Rhapsody (IBM, 2024), which provides an environment for collaborative systems engineering and software development; and System Composer (MathWorks, 2024), a system and software architecture modeling tool natively integrated with Matrix Laboratory (MATLAB) (MathWorks, 2024) and Simulink® (MathWorks, 2024).

Various types of qualitative and quantitative architecture analyses that can be performed within these tool environments typically involve creating and testing a scenario against the model (Dobrica & Niemela,

2002). The most common analysis patterns are quantitative in nature, such as roll-up or network analyses. In roll-up analyses, the top-level system parameter values are calculated as the sum of the values of the individual parts of which it is composed. The method of conducting the analysis varies by tool [e.g., *Rollup Pattern simulation* (No Magic, Inc., n.d.), *Simple Roll-Up analysis using robot system with properties* (MathWorks, n.d.)] but have the same end result. For example, the cost of a car can be calculated as the sum of the cost of its wheels, seats, bolts, chassis, engine, transmission, etc. This analysis may be used as part of a larger trade study to aid in make/buy decisions or optimal component selection to minimize the total cost of the system. Network analyses are usually found in software-oriented architecture tools. They are used for various purposes such as optimizing dataflow through a system to minimize latency or, as part of a safety and reliability study, to find a system’s resilience to the loss of data paths (McCabe, 2007).

Formal methods can be used for more rigorous analyses to *prove* that the model satisfies its requirements. Simulink Design Verifier™ (MathWorks, 2024) is a formal methods-based tool that includes a powerful property proving engine (Schürenberg, 2012) for detecting design errors in Simulink and Stateflow® (MathWorks, 2024) behavioral models. The novel approach of this research is to apply Simulink Design Verifier to prove compositional properties of a System Composer architecture model that does not contain linked behavioral models. This enables analysis of an architecture model in the system design phase when the cost of fixing defects is much lower than later in the development lifecycle (Dawson, Burrell, Rahim, & Brewster, 2010).

Assume-Guarantee Compositional Reasoning

We would like to know whether our system is composed correctly when modeling a system architecture during product design. Specifically, we would like to know if any two components are connected that have no business being connected, or whether the system’s subcomponents adequately meet its requirements. Early approaches to verifying system architectures involved flattening the architecture hierarchy prior to formal analysis (Jahier, Halbwachs, Raymond, Nicollin, & Lesens, 2007), but they did not scale well to real-world system models. Compositional reasoning partitions the formal analysis of a (potentially complex) system architecture into smaller verification tasks aligning with the natural architecture decomposition. Components are annotated with formal assume-guarantee contracts (McMillan K. L., 1999), where assumptions represent assertions about the environment or invariants on the component’s inputs and guarantees represent the component’s requirements.

The Assume Guarantee Reasoning Environment (AGREE) (Cofer, et al., 2012) is a compositional reasoning tool for Architectural Analysis and Design Language (AADL) models. The most common tool for authoring and analyzing AADL models is the Open Source AADL Tool Environment (OSATE) (Carnegie Mellon University, 2024), a collection of Eclipse (The Eclipse Foundation, 2024) plugins published and maintained by the Software Engineering Institute at Carnegie-Melon University. AGREE is implemented in OSATE as an AADL *annex*, a mechanism for providing domain-specific extensions to the AADL base language. Under the hood, AGREE translates the model and annotated behavioral contracts into Lustre (Halbwachs, Caspi, Raymond, & Pilaud, 1991), which is then run on a backend model checker, Kind2 (Champion, Mebsout, Stickse, & Tinelli, 2016) or JKind (Gacek, Backes, Whalen, Wagner, & Ghassabani, 2018), to either prove the contracts or generate a counterexample. The analysis is performed compositionally, following the architecture hierarchy, so analyses at higher levels are based on the verification of components at the next level down.

For example, Figure 1 illustrates a simple system (*top_level*) from the *Integer_Toy* model included with the AGREE distribution (loonwerks, 2024). The system contains three subcomponents, *A_sub*, *B_sub*, and *C_sub*, all of which have associated assumptions and/or guarantees. Component *B_sub* has an assumption

about its input that the value will always be less than 20. Is this a valid assumption? Is there an upstream component (in this case, A_sub) that guarantees that data sent to B_sub will always satisfy that assumption? Examining the contracts on A_sub, we see that the output is guaranteed always to be less than twice the input, assuming the input is less than 20. In other words, taken in isolation, the output of A_sub must always be less than 38. Without any other context, this invalidates the assumption on B_sub since an A_sub input value of 12, for example, would produce an output value in the range of 20-23. In this case, however the input to A_sub is constrained by an assumption placed on the top_level system that its input will be less than 10. Considering this constraint, the output of A_sub will always be less than 18, which satisfies the assumption on B_sub.

AGREE also checks that a component's assumptions and subcomponent guarantees satisfy its guarantees. In the same example, the top_level system's output is always guaranteed to be less than 50, assuming the input is less than 10. Do the subcomponent contracts support this guarantee? They do; however, if the guarantee had stated that the top_level system output would always be less than 5, for example, the guarantee would be violated and AGREE would generate a counterexample specifying the values of the component inputs and outputs (and other state variables) that caused the violation.

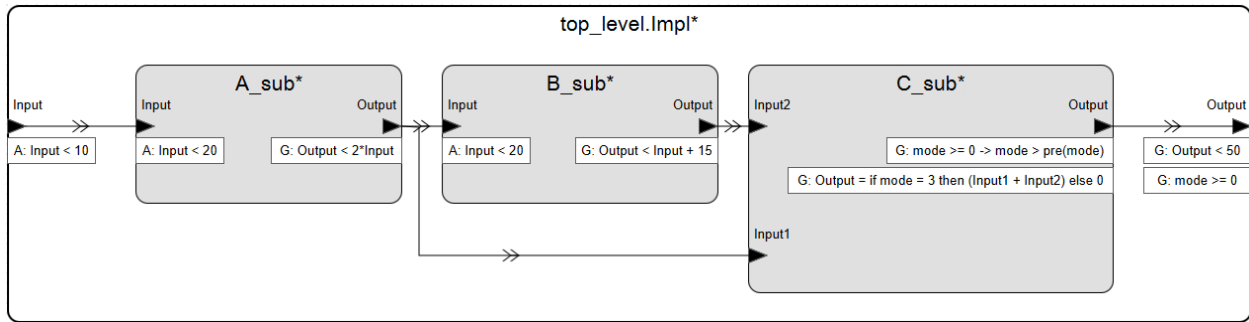


Figure 1. Integer_Toy Example in AADL/AGREE

Compositional Reasoning using System Composer

The novel method outlined herein adapts the AGREE-style analysis to a more generic architecture model, leveraging MATLAB, a popular analysis and modeling tool in the engineering community. The approach is divided into the following tasks:

- Modeling the system.
- Defining assume/guarantee contracts and associating them with the system elements.
- Transforming the system and its contracts into a solvable formal methods problem.
- Generating actionable results.

The following subsections describe how each of these pieces was addressed to perform the analysis.

Modeling the System

System Composer was selected to model the system architecture because of its native MATLAB integration, which is used as the analysis engine, and the availability of APIs for accessing the architecture model.

This brings an additional advantage over AADL since the models can be specified using a modern graphical editor. Although the AADL standard also defines a graphical language, the OSATE graphical editor is far from mature, and most users prefer to rely on the textual editor instead. The system model, shown below in Figure 2, is semantically equivalent to the AADL *Integer_Toy* model described in the previous section. It consists of hierarchically nested components with connected ports. Interfaces, which are used to define port data types in System Composer, were not used in this model in order to more closely align with its AADL counterpart. Parameters define internal state data, such as the system *mode*, which is not communicated through the ports.

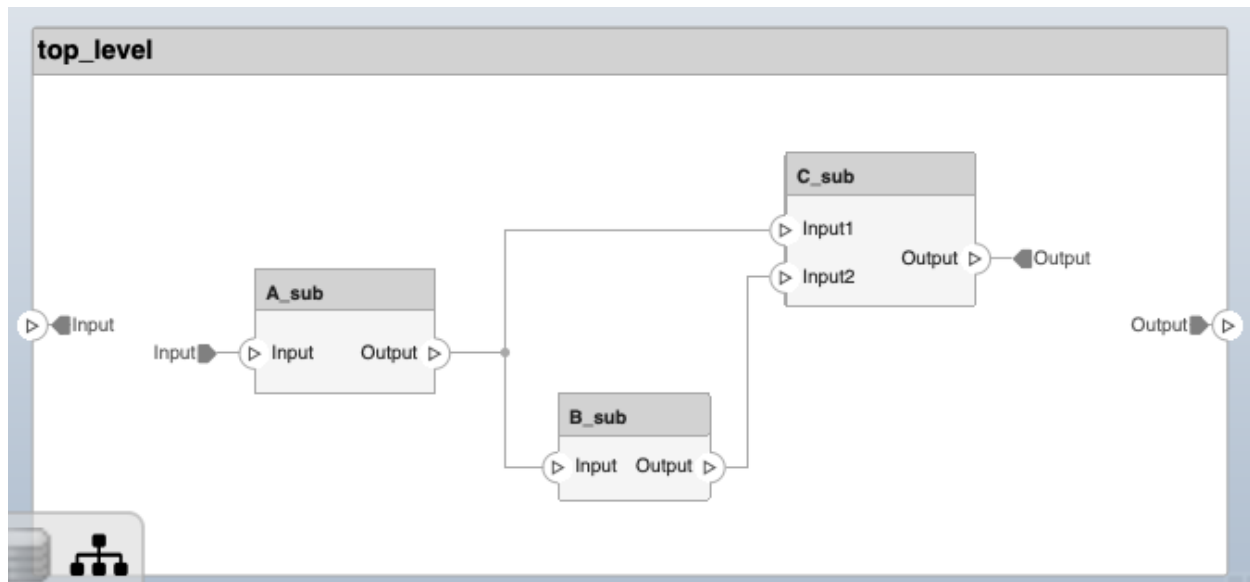


Figure 2. Integer_Toy Example in System Composer

Defining Assume/Guarantee Contracts and Associating them with System Elements

Once the system model was defined, a method for specifying contracts and associating them with model elements was needed. Profiling is a common mechanism for extending a modeling language and is found in languages such as UML and SysML. With profiling, custom profiles, which are collections of “stereotypes,” are created and linked to models. Stereotypes are generic type definitions for model elements that can be given type-specific values when applied to elements. They provide a convenient method for adding additional metadata and property fields to model elements to extend the ontology.

Therefore, we created an AGREE-based profile with a stereotype whose properties could be used to express the assume-guarantee contracts as text strings. The result is shown below in Figure 3.

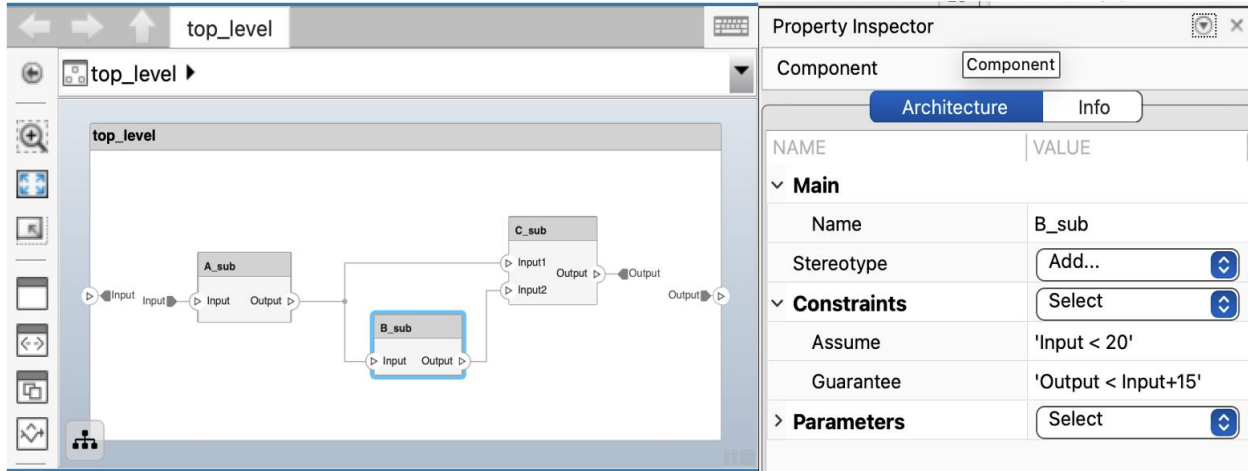


Figure 3. Integer_Toy Example with an Assume-Guarantee Contract Stereotype

Several issues were encountered with this method:

1. Scalability: A stereotype can only be instantiated on a component once, so additional stereotypes would need to be created to associate more contracts with a component, which is not scalable.
2. Reuse: The contracts are more like requirements than component attributes, that is, independent of the final implementation. Defining the contract directly on the component prevented the contracts from being reused without copy/paste and loss of synchronization.
3. Property Inspector: Stereotype property values can be set in System Composer through direct entry in the Property Inspector pane or through the API. However, there is a very limited area in the Property Inspector to enter text, as shown on the right-hand side of Figure 3. Additionally, stereotype properties are not designed to support code-like complex expressions for values. This makes it difficult to author anything beyond trivial contracts.
4. Syntax Highlighting and Validation: The benefits of using MATLAB include its rich semantic language and wide selection of toolboxes. However, it is difficult to author and debug complex code (in any language) without the features provided by a modern integrated development environment (IDE). By entering code as plain text in the Property Inspector, we lost that capability.
5. Traceability and Verification: There was no way to link a requirement to the stereotype for traceability and verification.

To address these issues, the stereotype was switched from being applied directly to the component to being applied to a requirement, as shown in Figure 4. We then linked the requirement to the desired component to associate the contract(s).

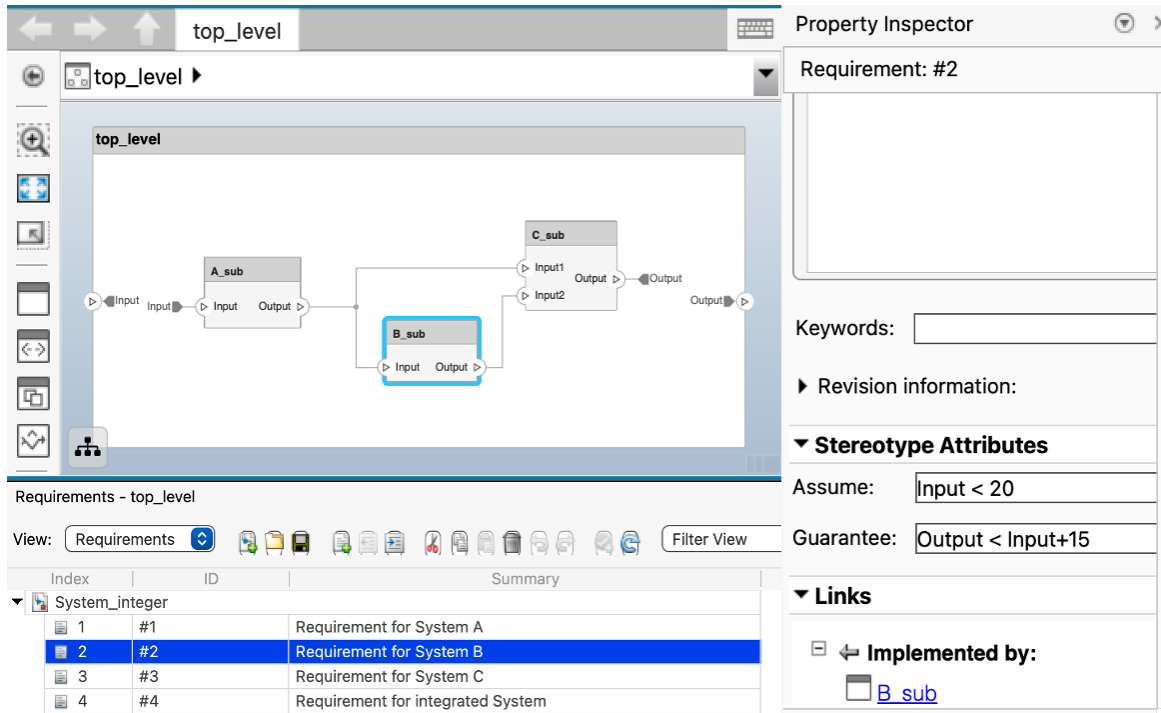


Figure 4. Stereotyped Requirement Contract

While this solves the scalability and reuse issues, it still has the same problems related to debugging and editability as when the contract stereotype was applied to the component. The requirements table block from the Requirements Toolbox™ (MathWorks, 2024) was also considered, but ultimately MATLAB Classes (MathWorks, 2024) were used to address these problems. A MATLAB Class defines an object that encapsulates data, and the operations performed on that data. Using MATLAB Classes allowed us to specify encapsulated constraints in an object-oriented fashion in the MATLAB editor, which provides syntax highlighting and code checking, and enables associating contracts with multiple components. An abstract class was then developed to provide a template for creating individual assume-guarantee contracts. The stereotype was then modified to contain a single string property for specifying the name of the contract class, as shown below in Figure 5.

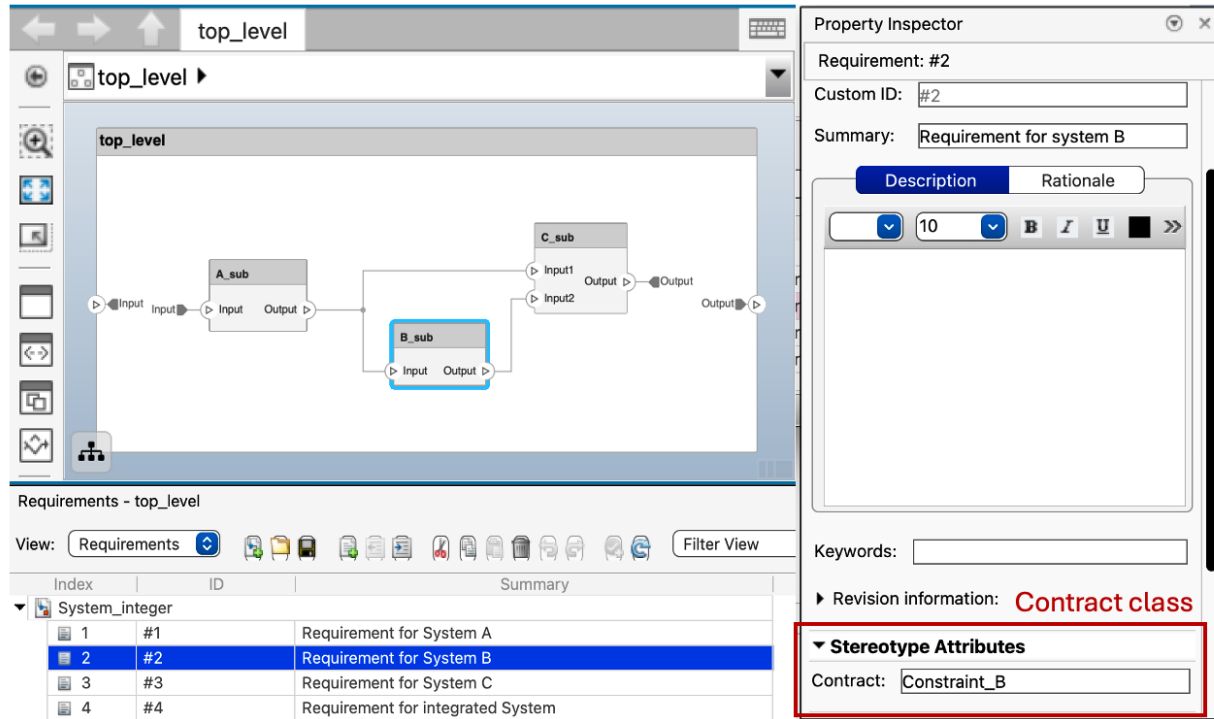


Figure 5. Requirement with the Contract Class Name Specified as the Stereotype Value

Transforming the System and Contracts to a Solvable Formal Methods Problem

With the system defined, we must now perform the following:

- Validate that the linked component satisfies the associated contract in terms of input and output symbols.
- Prove that the contracts are satisfied in the system.
- Identify assumptions and guarantees that are not satisfied.
- Provide information on the path(s) of failure, if a counterexample is generated.

The tooling developed in System Composer leverages the information specified in the architecture model and the linked AGREE contracts authored in MATLAB to validate the input and output symbols. In the contract for System B, for example, the inputs and outputs specified in the function signature are in terms of the expected symbols, as shown in Figure 6 (left). This is validated with the port and/or parameter names on the component, as shown in Figure 6 (right), and the user is alerted if there is a mismatch between them.

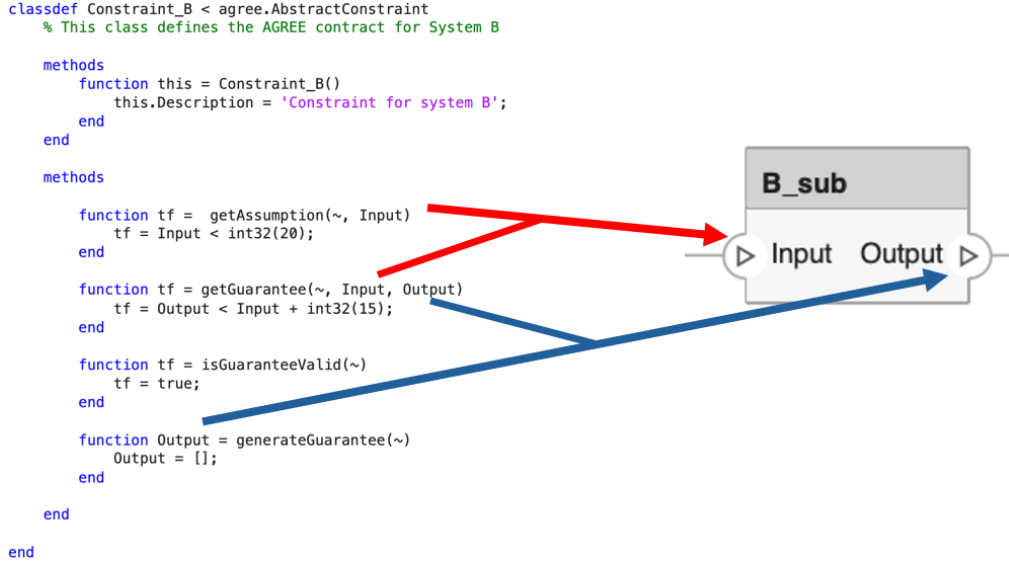


Figure 6. Contract for Component B_{sub} – Function Signature Symbols Correspond to the Component Port Names

We then leverage the analysis framework in System Composer to iterate over the system’s components based on connectivity to prove that the system as composed satisfies the assume-guarantee contracts. The iteration output is a Simulink proof model, aggregating all the contracts and their connections in a MATLAB Function Block that can be used as an input to Simulink Design Verifier for property proving. Simulink Design Verifier then either determines the system composition to be valid or provides one or more counterexamples that were found to invalidate the contracts.

Generating Actionable Results

One pain point when conducting an AGREE analysis in OSATE is that it is difficult to interpret and debug the generated counterexamples. This makes determining which model elements contribute to the contract violation non-trivial. In this work, we alleviate that pain by creating a System Composer sequence diagram depicting the order of events and property values from the counterexample as provided by Simulink Design Verifier. Additionally, we created a filtered Architecture View containing an architecture model slice, only including the components and specific ports implicated in the violation.

To demonstrate this, we changed the guarantee on the *Integer_Toy* model so that the system output is always less than 5. Figure 7 shows the generated sequence diagram when this is false.

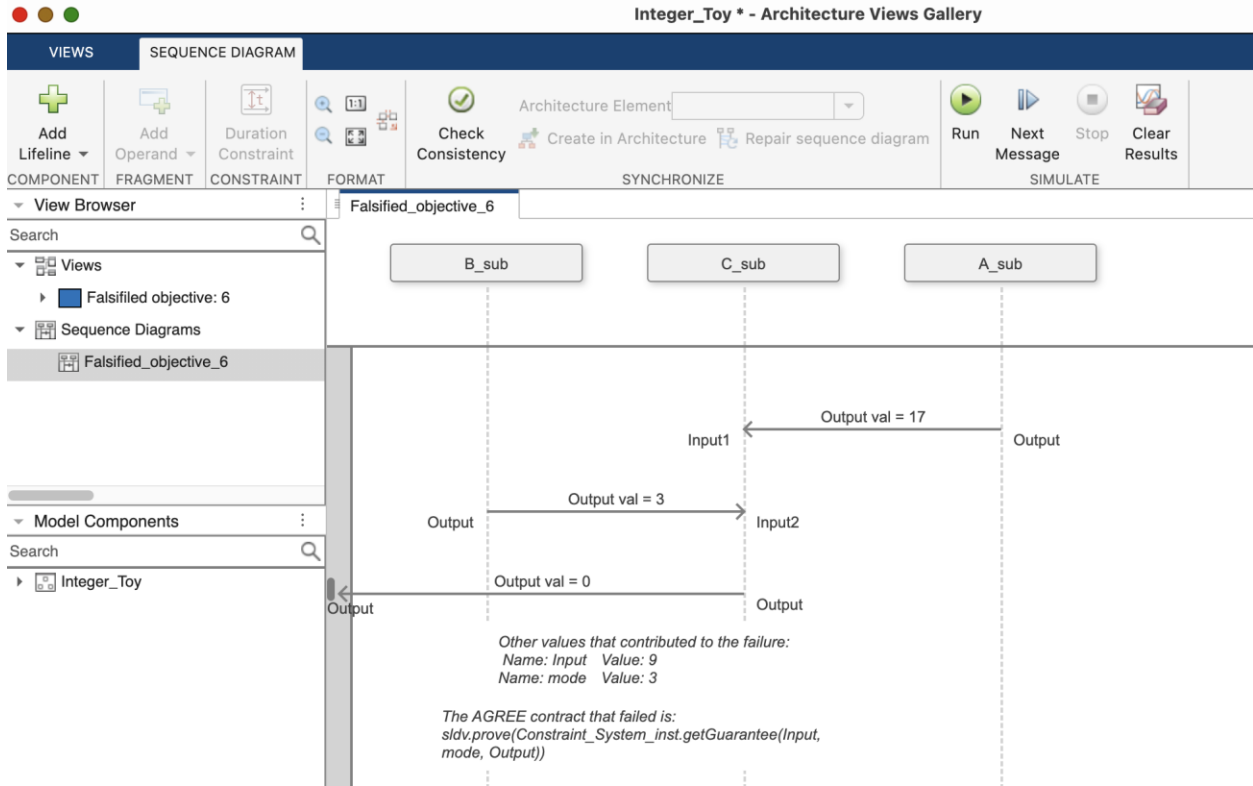


Figure 7. Sequence Diagram Counterexample for the Integer_Toy Model

Application–Automotive Control System

In this section, we demonstrate our approach on a more complex *Car* model, which is also included in the AGREE distribution. The top-level system represented in AADL is depicted in Figure 8. We replicate the system architecture in System Composer (see Figure 9), specify assume-guarantee contracts, formally verify the automotive control system, and compare with the AGREE results on the AADL model.

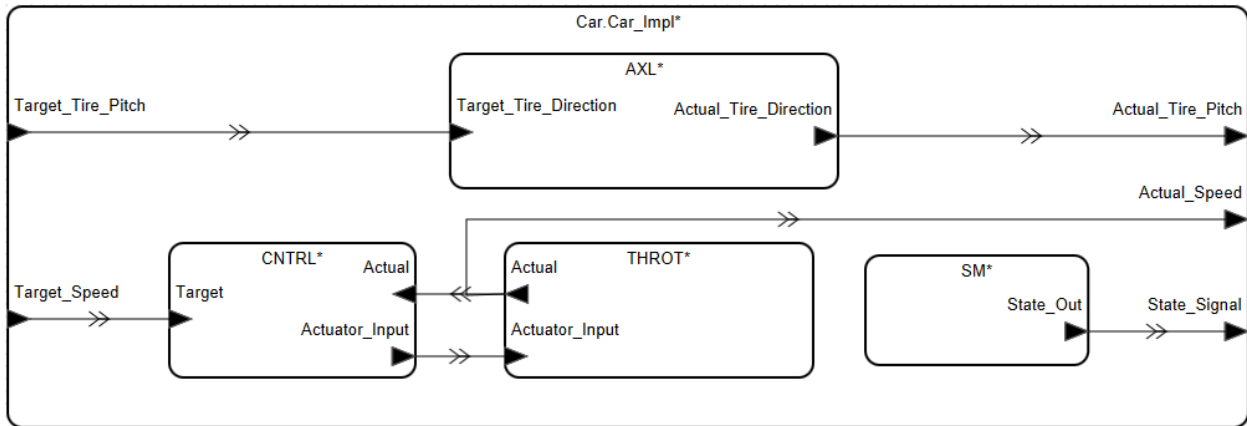


Figure 8. Car System-Level Diagram in OSATE

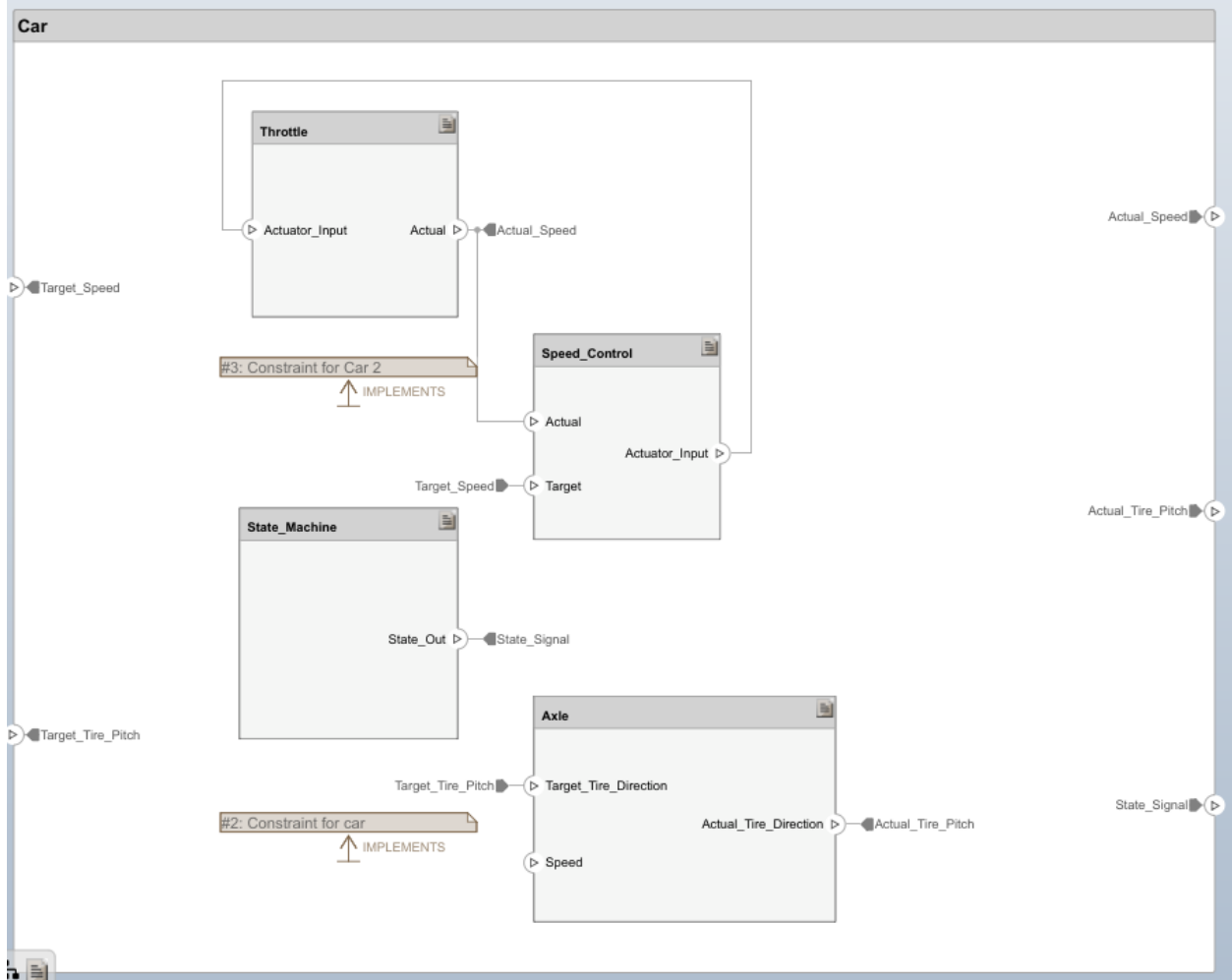


Figure 9. Car System-Level Diagram in System Composer

The top-level system specifies the component *assumptions* and *guarantees* based on safety-critical requirements as follows:

Assumptions:

- The target speed remains positive and under 150 mph

Guarantees:

- The actual speed does not exceed the target speed
- Acceleration is limited to 40.0 mph/s

The Car system AADL textual model is shown in Figure 10.

```

system Car
  features
    Target_Speed: in data port Types::speed.speed_impl;
    Actual_Speed: out data port Types::speed.speed_impl;
    Target_Tire_Pitch: in data port Types::pitch.pitch_impl;
    Actual_Tire_Pitch: out data port Types::pitch.pitch_impl;
    State_Signal: out data port Types::state_sig.impl;

  annex agree {**
    const max_accel : real = 40.0;
    assume A1 "target speed is positive" : Target_Speed.val >= 0.0;
    assume A2 "reasonable target speed" : Target_Speed.val < 150.0;
    property const_tar_speed =
      true -> Target_Speed.val = pre(Target_Speed.val);
    guarantee G_car_1 "actual speed is less than constant target speed" :
      const_tar_speed => (Actual_Speed.val <= Target_Speed.val);
    guarantee G_car_2 "acceleration is limited" :
      Agree_Nodes::abs(Actual_Speed.val - prev(Actual_Speed.val, 0.0)) < max_accel;
  **};

end Car;

system implementation Car.Car_Impl
  subcomponents
    THROT: system Transmission::Throttle.Throttle_Impl;
    CNTRL: system Transmission::Speed_Control.Speed_Control_Impl;
    AXL: system Steering::Axle.Axle_Impl;
    SM: system Transmission::State_Machine.State_Machine_Impl;

  connections
    SpeedToThrot: port CNTRL.Actuator_Input -> THROT.Actuator_Input {
      Communication_Properties::Timing => immediate;};
    AcSpeedToTop: port THROT.Actual -> Actual_Speed {Communication_Properties::Timing => immediate;};
    AcSpeedToCntrl: port THROT.Actual -> CNTRL.Actual {Communication_Properties::Timing => immediate;};
    TgSpeedToCntrl: port Target_Speed -> CNTRL.Target {Communication_Properties::Timing => immediate;};
    TgPtichToAxl: port Target_Tire_Pitch -> AXL.Target_Tire_Direction {
      Communication_Properties::Timing => immediate;};
    AcPtichToCar: port AXL.Actual_Tire_Direction -> Actual_Tire_Pitch {
      Communication_Properties::Timing => immediate;};
    SStoSM: port SM.State_Out -> State_Signal {Communication_Properties::Timing => immediate;};

end Car.Car_Impl;

```

Figure 10. AADL Representation of the Car System and AGREE Contracts

The key subsystems and their respective contracts are as follows:

Steering Module: This subsystem ensures safe tire pitch control, especially at higher speeds. The following AGREE contracts illustrate its operation:

- Assumption: None
- Guarantee: Tire pitch does not exceed 0.20 radians when the speed surpasses 45 mph

Transmission Module: This subsystem includes two critical components:

- *Speed Control:* Implements a Proportional-Derivative (PD) controller to minimize the error between target and actual speeds. The following AGREE contracts illustrate its operation:
 - Assumption: None
 - Guarantee: The PD controller will ensure that the actuator input minimizes the error and achieves the target speed within acceptable tolerance

- *Throttle*: This subsystem simulates acceleration proportional to actuator input, reflecting real-time speed adjustments. The following AGREE contracts illustrate its operation:
 - Assumption: None
 - Guarantee: The actual speed will be adjusted proportionally to the actuator input without exceeding physical constraints

State Machine: This subsystem maintains vehicle state consistency, ensuring transitions are traceable and output state signals align with historical system data. The following AGREE contracts illustrate its operation:

- Assumption: None
- Guarantee: The state signal output will always reflect the correct current state

The AGREE analysis conducted in OSATE produced a counterexample violating the guarantee that the output speed should always remain under 150 mph under specific input conditions, as shown in Figure 11. Figure 12 shows the full counterexample.
















Property	Result
▼  Verification for Car.Car_Impl	1 Invalid, 29 Valid
▼  Contract Guarantees	1 Invalid, 2 Valid
 Subcomponent Assumptions	Valid (0s)
 [G_car_1] actual speed is less than constant target speed	Invalid (0s)
 [G_car_2] acceleration is limited	Valid (0s)
>  This component consistent	1 Valid
>  THROT consistent	1 Valid
>  CNTRL consistent	1 Valid
>  AXL consistent	1 Valid
>  SM consistent	1 Valid
>  Component composition consistent	1 Valid
>  Verification for THROT	4 Valid
>  Verification for CNTRL	4 Valid
>  Verification for AXL	4 Valid
>  Verification for SM	9 Valid

Figure 11. Summary of AGREE Verification Generated in OSATE

Counterexample			

Variables for the selected component implementation			

Variable Name	0	1	2

Inputs:			
{Target Speed.val}	1499/10	0	0
{Target Tire Pitch.val}	-1/10	3/10	-1/10
State:			
{[G_car 1] actual speed is less than constant target speed}	true	true	false
{_TOP.AXL..ASSUME.HIST}	true	true	true
{_TOP.CNTRL..ASSUME.HIST}	true	true	true
{_TOP.SM..ASSUME.HIST}	true	true	true
{_TOP.THROT..ASSUME.HIST}	true	true	true
{const tar speed}	true	false	true
Outputs:			
{Actual Speed.val}	1499/110	1499/121	14990/1331
{Actual Tire Pitch.val}	-1/10	1/5	-1/10
{State Signal.val}	0	0	0

Variables for AXL			

Variable Name	0	1	2

Inputs:			
{AXL.Speed.val}	0	451/10	0
{AXL.Target Tire Direction.val}	-1/10	3/10	-1/10
State:			
{AXL..ASSUME.HIST}	true	true	true
Outputs:			
{AXL.Actual Tire Direction.val}	-1/10	1/5	-1/10

Variables for CNTRL			

Variable Name	0	1	2

Inputs:			
{CNTRL.Actual.val}	1499/110	1499/121	14990/1331
{CNTRL.Target.val}	1499/10	0	0
State:			
{CNTRL..ASSUME.HIST}	true	true	true
{CNTRL.e}	1499/11	-1499/121	-14990/1331
{CNTRL.e dot}	-1499/11	17988/121	-1499/1331
{CNTRL.e int}	1499/11	14990/121	-31479/1331
{CNTRL.u}	1499/11	-1499/121	-14990/1331
Outputs:			
{CNTRL.Actuator Input}	1499/11	-1499/121	-14990/1331

Variables for SM			

Variable Name	0	1	2

Inputs:			
State:			
{SM..ASSUME.HIST}	true	true	true
Outputs:			
{SM.State Out.val}	0	0	0

Variables for THROT			

Variable Name	0	1	2

Inputs:			
{THROT.Actuator Input}	1499/11	-1499/121	-14990/1331
State:			
{THROT..ASSUME.HIST}	true	true	true
Outputs:			
{THROT.Actual.val}	1499/110	1499/121	14990/1331

Figure 12. AGREE Counterexample Generated in OSATE

We then replicated the analysis using our MATLAB Toolbox, transforming the system model and contracts into a System Composer model with MATLAB Classes representing the contracts linked to the model through requirements. The toolbox-generated counterexample, shown in Figure 13, matches that from AGREE/AADL, demonstrating parity between the tools.

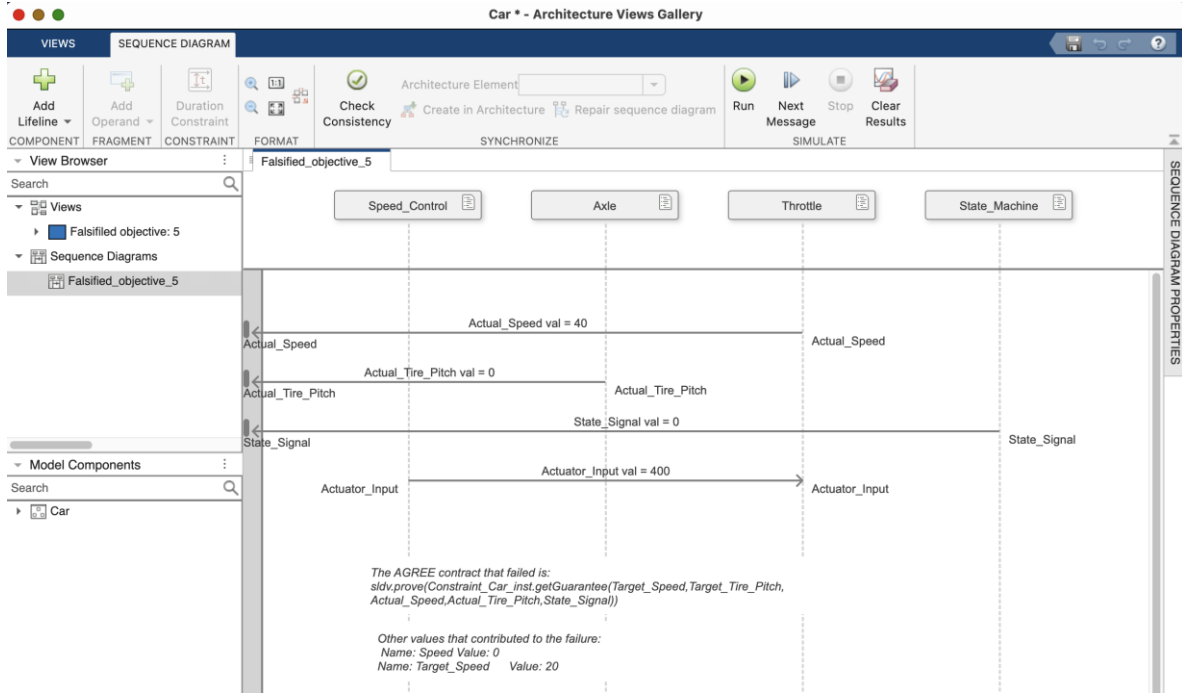


Figure 13. Counterexample Sequence Diagram in System Composer

Conclusion

While formal methods are useful for early model defect detection, they have seen a slow adoption within the systems engineering community for numerous reasons. This paper demonstrates a method for conducting one type of formal analysis, compositional reasoning, in System Composer, a commercially available MBSE tool. Using System Composer, we create an architecture model, apply assume-guarantee contracts to the components within, and then use the property-proving engine of Simulink Design Verifier to perform the formal analysis. The analysis results were validated using equivalence testing against an AADL model built in OSATE and analyzed with AGREE. Some customizations were required, such as creating a custom MATLAB Class to capture the assume-guarantee contracts, a function to convert the architecture model into an analyzable format, and one to convert the analysis results into a sequence diagram. The System Composer analysis results matched those in OSATE. However, the generated sequence diagrams in System Composer made them easier to understand and troubleshoot. Integrating visual tools like annotated sequence diagrams and architecture views simplified the analysis process for complex systems, facilitating better understanding and resolution of failures in large-scale models. As models get larger and more complex, parsing through tabular data to understand the root of the problem becomes significantly more time consuming and difficult. Sequence diagrams that can be annotated and simulated provide a richer environment for understanding and debugging issues as they are encountered.

This work opens the door to future improvements, four of which are listed below:

1. *Modularity*: System Composer supports the concept of *model referencing*, where large models can be partitioned into encapsulated subsystems. Extending this work for architecture models that include model referencing would help address scalability and reuse by dividing the large integrated model into modular subcomponents.
2. *Implementation*: Since System Composer is a Simulink-based tool, it should be possible to enhance this work by refining components and contracts with actual Simulink implementations. This would enable the analysis to be run on a mixed contract/implemented system.
3. *Verification*: Since sequence diagrams were used to capture the counterexample from the analysis (and they are executable in System Composer), using them as a verification tool after linking implementation models may be possible to ensure that each individual Simulink behavior, or even the entire implemented architecture, adheres to the contract constraints.
4. *SysML v2*: The SysML v2 specification includes a formalism for specifying requirements as constraints. MathWorks has stated publicly that System Composer will be a SysML v2-compliant tool. There is also an effort in the Object Management Group's Systems Modeling Community to bring AADL to SysML v2 natively. Therefore, using our toolbox to perform compositional reasoning with System Composer and Simulink Design Verifier on architecture models from other MBSE tools may be possible.

The primary goal of this work was to make MBSE-based formal analysis more accessible to the systems engineering community. Accordingly, these customizations have been implemented as a MATLAB Toolbox that is available upon request from the authors of this paper.

References

- Ansys. (2024). *Ansys SCADE Suite*. Retrieved 11 18, 2024, from Ansys:
<https://www.ansys.com/products/embedded-software/ansys-scade-suite>
- Carnegie Mellon University. (2024). Retrieved 11 18, 2024, from Open Source AADL Tool Environment (OSATE): <https://osate.org/>
- Champion, A., Mebsout, A., Stickse, C., & Tinelli, C. (2016). The Kind 2 Model Checker. In S. Chaudhuri, & A. Farzan (Ed.), *Computer Aided Verification. CAV 2016. Lecture Notes in Computer Science*. 9780, pp. 510-517. Springer, Cham. doi:10.1007/978-3-319-41540-6_29
- Cofer, D., Gacek, A., Miller, S., Whalen, M. W., LaValley, B., & Sha, L. (2012). Compositional Verification of Architectural Models. *Goodloe, A.E., Person, S. (eds) NASA Formal Methods. NFM 2012. Lecture Notes in Computer Science*. 7226, pp. 126-140. Norfolk, Virginia: Springer, Berlin, Heidelberg. doi:10.1007/978-3-642-28891-3_13
- da Silva, A. J., Linhares, M. V., Padilha, R., Roqueiro, N., & de Oliveira, R. S. (2006). An Empirical Study of SysML in the Modeling of Embedded Systems. *2006 IEEE International Conference on Systems* (pp. 4569-4574). Taipei, Taiwan: IEEE. doi:10.1109/ICSMC.2006.384866
- Dassault Systemes. (2024). *Cameo Systems Modeler*. Retrieved 11 18, 2024, from Dassault Systemes:
<https://www.3ds.com/products/catia/no-magic/cameo-systems-modeler>
- Davis, J. A., Clark, M., Cofer, D., Fifarek, A., Hinchman, J., Hoffman, J., . . . Wagner, L. (2013). Study on the Barriers to the Industrial Adoption of Formal Methods. *FMICS 2013. Lecture Notes in Computer Science*. 8187, pp. 63-67. Springer, Berlin, Heidelberg. doi:10.1007/978-3-642-41010-9_5
- Dawson, M., Burrell, D. N., Rahim, E., & Brewster, S. (2010). Integrating Software Assurance into the Software Development Life Cycle (SDLC). *Journal of Information Systems Technology and Planning*, 3, 49-53.

- Dobrica, L., & Niemela, E. (2002). A survey on software architecture analysis methods. *IEEE Transactions on Software Engineering*, 28(7), 638-653. doi:10.1109/TSE.2002.1019479
- Gacek, A., Backes, J., Whalen, M., Wagner, L., & Ghassabani, E. (2018). The JKind Model Checker. In H. Chockler, & G. Weissenbacher (Ed.), *Computer Aided Verification. CAV 2018. Lecture Notes in Computer Science. 10982*, pp. 20-27. Springer, Cham. doi:10.1007/978-3-319-96142-2_3
- Halbwachs, N., Caspi, P., Raymond, P., & Pilaud, D. (1991). The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*. 79(9), pp. 1305-1320. IEEE. doi:10.1109/5.97300
- IBM. (2024). *IBM Engineering Requirements Management*. Retrieved 11 18, 2024, from IBM: <https://www.ibm.com/products/requirements-management>
- IBM. (2024). *IBM Engineering Systems Design Rhapsody*. Retrieved 11 18, 2024, from IBM: <https://www.ibm.com/products/systems-design-rhapsody>
- Jahier, E., Halbwachs, N., Raymond, P., Nicollin, X., & Lesens, D. (2007). Virtual execution of AADL models via a translation into synchronous programs. *Proceedings of the 7th ACM & IEEE International Conference on Embedded Software* (pp. 134-143). Salzburg, Austria: Association for Computing Machinery. doi:10.1145/1289927.1289951
- loonwerks. (2024). *loonwerk/AGREE*. Retrieved 11 18, 2024, from Github: <https://github.com/loonwerks/AGREE/releases>
- MathWorks. (2024). *Classes*. Retrieved 11 18, 2024, from MathWorks: <https://www.mathworks.com/help/matlab/object-oriented-programming.html>
- MathWorks. (2024). *Create and Share Toolboxes*. Retrieved 11 18, 2024, from MathWorks: https://www.mathworks.com/help/matlab/matlab_prog/create-and-share-custom-matlab-toolboxes.html
- MathWorks. (2024). *MATLAB*. Retrieved 11 18, 2024, from MathWorks: <https://www.mathworks.com/products/matlab.html>
- MathWorks. (2024). *Requirements Toolbox*. Retrieved 11 18, 2024, from MathWorks: <https://www.mathworks.com/products/requirements-toolbox.html>
- MathWorks. (2024). *Simulink*. Retrieved 11 18, 2024, from MathWorks: <https://www.mathworks.com/products/simulink.html>
- MathWorks. (2024). *Simulink Design Verifier*. Retrieved 11 18, 2024, from MathWorks: <https://www.mathworks.com/products/simulink-design-verifier.html>
- MathWorks. (2024). *Stateflow*. Retrieved 11 18, 2024, from MathWorks: <https://www.mathworks.com/products/stateflow.html>
- MathWorks. (2024, 11 04). *System Composer*. Retrieved from MathWorks: <https://www.mathworks.com/products/system-composer.html>
- MathWorks. (n.d.). *Simple Roll-Up analysis using robot system with properties*. Retrieved 11 18, 2024, from MathWorks: <https://www.mathworks.com/help/systemcomposer/ug/simple-roll-up-analysis.html>
- McCabe, J. D. (2007). *Network Analysis, Architecture, and Design (3rd ed.)*. Elsevier.
- McMillan, C., Lee, L., Russell, L., Prince, D., Hasanovic, N., Durling, M., . . . Kleven, E. (2024). Verification and Validation of Model-Based Systems Requirements and Design Leveraging Formal Methods to Increase Development Assurance. *AeroTech Conference & Exhibition*. SAE International. doi:10.4271/2024-01-1917
- McMillan, K. L. (1999). Circular Compositional Reasoning about Liveness. In L. Pierre, & T. Kropf (Ed.), *Correct Hardware Design and Verification Methods. CHARME 1999. Lecture Notes in Computer Science. 1703*, pp. 342-346. Berkeley, CA, USA: Springer, Berlin, Heidelberg. doi:10.1007/3-540-48153-2_30
- Microsoft. (2024). *Microsoft PowerPoint*. Retrieved 11 18, 2024, from Microsoft: <https://www.microsoft.com/en-us/microsoft-365/powerpoint>
- Microsoft. (2024). *Visio*. Retrieved 11 18, 2024, from Microsoft: <https://www.microsoft.com/en-us/microsoft-365/visio/flowchart-software>

- Murugesan, A., Whalen, M. W., Rayadurgam, S., & Heimdahl, M. P. (2013). Compositional verification of a medical device system. *HILT '13: Proceedings of the 2013 ACM SIGAda annual conference on High integrity language technology* (pp. 51-64). Pittsburgh, Pennsylvania, USA: Association for Computing Machinery. doi:10.1145/2527269.2527272
- No Magic, Inc. (n.d.). *Rollup Pattern simulation*. Retrieved 11 18, 2024, from No Magic: <https://docs.nomagic.com/display/CST2021x/Rollup+Pattern+simulation>
- Object Management Group. (2014). *Business Process Model and Notation (BPMN)*. Retrieved 11 18, 2024, from Object Management Group: <https://www.omg.org/spec/BPMN>
- Object Management Group. (2017). *About the Unified Modeling Language (UML)*. Retrieved 11 18, 2024, from OMG Standards Development Organization: <https://www.omg.org/spec/UML/>
- Object Management Group. (2024). *OMG System Modeling Language (SysML)*. Retrieved 11 18, 2024, from Object Management Group: <https://www.omg.org/spec/SysML/2.0/Beta2/About-SysML>
- SAE International. (2022). *Architecture Analysis & Design Language (AADL) AS5506D*. SAE International. doi:10.4271/AS5506D
- Schürenberg, M. (2012). Scalability Analysis of the Simulink Design Verifier on an Avionic System. *Bachelor Thesis*. Hamburg, Germany: Hamburg University of Technology. Retrieved 11 18, 2024, from <https://www.ifis.uni-luebeck.de/~moeller/publist-sts-pw-and-m/source/papers/2012/schuer12.pdf>
- Sparx Systems. (2024). *Enterprise Architect*. Retrieved 11 18, 2024, from Sparx Systems: <https://sparxsystems.com/products/ea/>
- The Eclipse Foundation. (2024). Retrieved 11 18, 2024, from Eclipse Foundation: <https://www.eclipse.org/>

Biography



Isaac Amundson. Isaac Amundson is a Technical Fellow at Collins Aerospace. He has over 20 years of academic and industry experience in safety-critical cyber-physical systems research and development and has first-hand knowledge of the challenges, roadblocks, and pitfalls involved in certifying products in heavily regulated environments. He has M.S. degrees in Mechanical Engineering and Computer Science, and a Ph.D. in Computer Science from Vanderbilt University. In his current role, he leads multiple government-sponsored (DARPA, NASA) research programs in which he is exploring practical methods for generating and conveying assurance through the use of new and existing formal analysis tools.



Josh Kahn. Josh Kahn is an MBSE Solutions Specialist at MathWorks, collaborating with business partners to solve complex pains related to architecture modeling, systems integration, and simulation. Josh is an active participant in the INCOSE systems engineering community and the OMG standards development organization and has published multiple papers. Josh has been with MathWorks since 2020, bringing ten years of industry experience from GE and Collins, contributing to and leading systems development in aerospace and defense industries. He has an M.Eng. in Space Systems Engineering from the University of Michigan and B.S. in Mechanical Engineering from Florida Atlantic University.



Vidya Srinivasan. Vidya Srinivasan is a Principal Software Engineer at MathWorks, specializing in the System Composer product. She focuses on delivering robust architecture modeling tools to meet the needs of the MBSE community. Vidya has been with MathWorks since 2003, contributing to various products and capabilities. She has extensive experience in designing and implementing scalable, robust, and high-performing software to meet customer needs. Vidya holds an M.S. in Electrical Engineering from Northern Illinois University.



Dr. Gopal Narayan Rai. Dr. Narayan Rai is a Principal Investigator and Aerospace Engineer with expertise in Formal Methods. He holds a Ph.D. in Computer Science and works in Research & Development at Collins Aerospace. His research focuses on AI-driven automation, formal methods, and safety-critical systems. His work aims to enhance reliability and automation in aerospace systems through advanced computational techniques.



Dr. Janet Liu. Dr. Liu is a Senior Research Manager at Collins Aerospace. She earned her Ph.D. in Computer Science from Iowa State University. She has been with Collins Aerospace (previously Rockwell Collins) since 2008, with experience in Commercial Systems Flight Controls and Applied Research & Technology. In her current position, she has been promoting the application of formal methods and model-based analysis for safe & secure system design within Collins Aerospace.