AFRL-RQ-WP-TR-2016-0172

MODEL-BASED COMPOSITIONAL REASONING FOR COMPLEX SYSTEMS OF SYSTEMS (SoS)

M. Anthony Aiello, Benjamin D. Rodes, Ashlie B. Hocking, Jonathan C. Rowanhill, and John C. Knight

Dependable Computing LLC

Alec J.D. Bateman and Kevin Ehlmann

Barron Associates Inc.

NOVEMBER 2016
Final Report

THIS IS A SMALL BUSINESS INNOVATION RESEARCH (SBIR) PHASE II REPORT.

AIR FORCE RESEARCH LABORATORY
AEROSPACE SYSTEMS DIRECTORATE
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7541
AIR FORCE MATERIEL COMMAND
UNITED STATES AIR FORCE

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the USAF 88th Air Base Wing (88 ABW) Public Affairs Office (PAO) and is available to the general public, including foreign nationals.

Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RQ-WP-TR-2016-0172 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

*//Signature//
SEAN J. REGISFORD
Program Manager
Autonomous Control Branch
Power and Control Division

//Signature//
MATTHEW A. CLARK, Chief
Autonomous Control Branch
Power and Control Division
Aerospace Systems Directorate

//Signature//
BRYAN J. CANNON, Principal Scientist
Power and Control Division
Aerospace Systems Directorate

This report is published in the interest of scientific and technical information exchange and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

*Disseminated copies will show "//Signature//" stamped or typed above the signature blocks.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
November 2016	Final	15 August 2014 – 14 October 2016

4. TITLE AND SUBTITLE
MODEL-BASED COMPOSITIONAL REASONING FOR COMPLEX SYSTEMS OF SYSTEMS (SoS)

5a. CONTRACT NUMBER
FA8650-14-C-2528

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
65502F

6. AUTHOR(S)
M. Anthony Aiello, Benjamin D. Rodes, Ashlie B. Hocking, Jonathan C. Rowanhill, and John C. Knight (Dependable Computing LLC)
Alec J.D. Bateman and Kevin Ehlmann (Barron Associates Inc.)

5d. PROJECT NUMBER
3005

5e. TASK NUMBER

5f. WORK UNIT NUMBER
Q1DC

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Dependable Computing LLC
2120 North Pantops Drive
Charlottesville, VA 22911

Barron Associates Inc.
1410 Sachem Place, Suite 202
Charlottesville, VA 22901

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Research Laboratory
Aerospace Systems Directorate
Wright-Patterson Air Force Base, OH 45433-7541
Air Force Materiel Command
United States Air Force

10. SPONSORING/MONITORING AGENCY ACRONYM(S)
AFRL/RQQA

11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S)
AFRL-RQ-WP-TR-2016-0172

12. DISTRIBUTION/AVAILABILITY STATEMENT
DISTRIBUTION STATEMENT A: Approved for public release. Distribution is unlimited.

13. SUPPLEMENTARY NOTES
This is a Small Business Innovation Research (SBIR) Phase II report. The contractor has waived its SBIR data rights protections.

PA Case Number: 88ABW-2017-0371; Clearance Date: 31 Jan 2017.

14. ABSTRACT

This report was developed under a SBIR contract. A system interface abstraction technology, a novel theory and framework that enables system of systems analysis, was developed in this effort. SoS analysis is a major challenge area due to the complexity of behavioral interactions possible in an SoS. Testing of these systems cannot provide adequate coverage or assurance of correct behavior. Compositional analysis, which reasons about system behaviors from component abstractions, offers a compelling alternative, but requires that: 1) components provide the guarantees claimed under stated assumptions and 2) assumptions stated are comprehensive. System interface abstraction technology provides necessary support through: 1) formal analysis and argument-based reasoning of component context, assumptions, and guarantees and 2) formal analysis and argument-based reasoning of compositional properties based on components. System interface abstraction technology is comprised of four key elements: 1) a novel theory of SoS engineering; 2) a novel assurance-case technology for argument composition; 3) a novel theory of enhanced formal contracts; and 4) a novel compositional analysis framework. The technology is demonstrated by application to examples: 1) a novel, argument-based response to a hypothetical request for proposals for a simple system of systems and 2) a hypothetical small unmanned aerial system (UAS).

15. SUBJECT TERMS SBIR Report, formal methods, assurance cases, compositional reasoning, compositional argumentation, compositional certification, verification, validation, certification

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT:	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON (Monitor)
a. REPORT	b. ABSTRACT	c. THIS PAGE			Sean J. Regisford
Unclassified	Unclassified	Unclassified	SAR	210	19b. TELEPHONE NUMBER *(Include Area Code)* (937) 713-7021

January 23, 2017

Dr. Sean J. Regisford
AFRL/RQQA
2210 Eighth Street
Bldg 146, Room 300
Wright-Patterson AFB, OH 45433

Subject: Contract Number FA8650-14-C-2528, Phase II SBIR

Dear Dr. Regisford

Dependable Computing hereby waives its SBIR Data Rights to all contents of the final report for subject contract. The Government is granted an unlimited nonexclusive license to use, modify, reproduce, release, perform, and display or disclose this report and the data contained herein.

We affirm that we are aware that the report may be released to other contractors to the Government and approve potential release to other contractors.

Sincerely,

John C. Knight

President

Dependable Computing LLC
2120 North Pantops Drive, Charlottesville, VA 22911-8648
Phone: 434-563-3183
Web: www.dependablecomputing.com

Table of Contents

Section **Page**

List of Figures

Figure **Page**

List of Tables

Table **Page**

1 SUMMARY

Systems of systems (SoS) — systems for which the supporting components are regarded as individual systems — exhibit significant complexity. This complexity, which arises from the richness of behavioral interactions and from the inherent complexity of the components, poses a significant challenge to traditional verification, validation and certification approaches.

Traditionally, verification and validation of systems of systems has been attempted through testing. Unfortunately, testing cannot provide complete coverage, even at the unit level. The complexity of behavioral interactions that arise in a system of systems makes them essentially untestable.

Formal methods, which use mathematical proofs to establish critical properties, have been successfully applied at the unit level. At the system level, however, these techniques often suffer from a state-space explosion problem similar to that of testing.

Compositional reasoning addresses these limitation by enabling reasoning about system of systems behaviors at the architectural level using abstractions of component behaviors. Compositional reasoning depends on two critical assumptions:

1. the components provide the guarantees they claim, under the assumptions they state; and
2. the assumptions stated are comprehensive of all of the required context under which the guarantees are provided.

Stated assumptions are often insufficiently complete to support compositional analysis. Additional support is required to strengthen compositional reasoning. Specifically, support is needed to:

1. enable more complete reasoning about components by more fully identifying the context under which guarantees can be established; and
2. enable more complete compositional reasoning by accounting for the complete context.

This Phase II effort builds upon our successful Phase I effort to develop system interface abstraction technology: a novel theory and a framework that:

1. supports formal analysis and argument-based reasoning of component context, assumptions and guarantees; and
2. supports formal analysis and argument-based reasoning of compositional properties based on components.

System interface abstraction technology has four critical components:

1. a novel theory of system-of-systems engineering including a reference model, a reference development process, and reference mechanics;
2. novel assurance-case technology for systems of systems supporting argument composition;
3. a novel theory of enhanced formal contracts for systems of systems; and
4. a novel compositional analysis framework for systems of systems.

The technology is demonstrated by application to two examples:

1. an application to a hypothetical small unmanned aicract system (UAS) based on the Ultra Stick platform; and
2. a novel, argument-based response to hypothetical request for proposals for development of a hypothetical system based on the cooling-tanks problem.

2 INTRODUCTION

2.1 Problem Description

As processing power continues to increase, the amount of software deployed for modern systems increases. Whereas space, weight and power considerations limit the scope and complexity of features realized in hardware for new systems, the space, weight and power requirements of software change relatively little based on software size and complexity. As such, software of virtually any size and complexity can be included on virtually any system.

While the scope and complexity of software — including safety-critical software — has increased substantially, there has not been a concomitant increase in the efficacy and capability of tools and methods for the verification and validation of this software. This lack is particularly striking for safety-critical software applications for modern aerospace systems, especially as these systems incorporate more and more significant autonomy. Moreover, the composition of these systems into systems of systems presents further challenges, as behaviors emerge from the unexpected ways in which autonomous systems interact to produce new and unanticipated failure modes.

2.1.1 System of Systems Complexity

A system of systems is a system for which its supporting components are regarded as individual systems that may operate and be managed independently from each other [1] [2]. Each component system may be similarly composed of subcomponents that can be further decomposed recursively, forming a hierarchical decomposition.

The focus of this effort is on systems of systems for which there is a centralized managing authority that has coercive power on component systems, and regulates, manages and certifies the system of systems. The two applicable categories of systems of systems are therefore those in which either (1) component systems are developed specifically for use in a given system of systems (a *directed* system of systems) or (2) component systems retain independent ownership, objectives, funding and development, etc. (an *acknowledged* system of systems) [1].

The behavioral complexity of systems of systems, coupled with the variability and dynamic nature of system-of-systems components, increases the difficultly and costs for the managing authority. The behavioral complexity of systems of systems arises from the complex behaviors of the component systems. Frequently, component behaviors are the result of complex software that operates in the context of the component-system hardware and the component-system environment. The system-of-systems problem is thus heavily dependent on understanding how physical and software-defined behaviors will compose within the novel environment of the system of systems.

2.1.2 Limitations in Testing, Formal Methods and System Modeling

Testing, commonly used to provide assurance for cyber-physical systems, cannot provide complete coverage — even at the unit level [3]. This lack of coverage is even more pronounced in system integration and system of systems integration. The state-space explosion inherent to the composition of complex systems into systems of systems is essentially untestable, leading to concerns of apparent nondeterminism, emergence, and interoperability.

The application of formal methods to software systems has enabled many of the limitations inherent to software testing to be redressed. Rather than sampling the input space of a piece of

software to provide some assurance of correct operation, formal methods allow formal proofs of correctness over all inputs, providing complete assurance of correct operation under the assumptions of the formal analysis.

Formal methods have been most successfully applied at the unit level, where software function is relatively constrained, inputs are clearly identified, and desired outputs are well understood. This success, combined with the success of standards like DO–178B/C, has led some researchers to conclude that, in essence, the problem has been solved at the unit level [4] [5]. While these claims are incomplete and optimistic — and largely inapplicable outside the domain of commercial avionics software — they do correctly point to the more pressing issues: requirements engineering and system specification, architecture design and modeling, especially with respect to the composition of systems within a system of systems.

In an effort to cope with the complexity of modern systems and systems of systems, systems engineers have developed and adopted system and architectural modeling languages. These languages provide a more rigorous framework for developing and presenting requirements, use cases, behaviors, and architectures. Some of the developed modeling languages even provide formal semantics, upon which certain analytic capabilities have been developed.

System and architectural modeling languages facilitate decomposition of complex systems and systems of systems into simpler components, by providing explicit representations of the modularity employed. Using these languages, component interfaces are clearly described and component contracts are illustrated through the connections between components. Tools such as AGREE [6] enable partial analysis of these contracts using assume-guarantee reasoning.

Compositional reasoning of this form depends on two critical assumptions:

1. the components provide the guarantees they claim, under the assumptions they state; and
2. the assumptions stated are comprehensive of all of the required context under which the guarantees are provided.

Commonly, explicitly stated assumptions are not comprehensive, but include many additional, implicit assumptions about the system context — especially the environment in which the system will operate. Dependence on these implicit assumptions threatens the validity of any analysis that does not explicitly include them, reducing justifiable assurance in correct operation of the component and thus in the correctness of the composition.

Additional support is, therefore, required to strengthen compositional reasoning. Specifically, support is needed to:

1. enable more complete reasoning about components by more fully identifying the context under which guarantees can be established; and
2. enable more complete compositional reasoning by accounting for the complete context.

2.2 Solution Approach

In this Phase II effort, we built upon our successful Phase I effort, furthering the development of the *system-interface abstraction technology* (SIAT). System-interface abstraction technology provides a theory and a framework that:

1. supports formal analysis and argument-based reasoning of component context, assumptions and guarantees; and

2. supports formal analysis and argument-based reasoning of compositional properties based on components.

System-interface abstraction technology is a system-of-systems enabling technology that provides a comprehensive infrastructure to support compositional reasoning and assessment of complex systems of systems.

The system-of-systems enabling technology rests on four pillars (Figure 1):

1. a novel theory of system-of-systems engineering including a reference model, a reference development process, and reference mechanics;
2. novel assurance-case technology for systems of systems supporting argument composition;
3. a novel theory of enhanced formal contracts for systems of systems; and
4. a novel compositional analysis framework for systems of systems.

Underlying the pillars are two demonstrations:

1. an application to a hypothetical small UAS based on the Ultra Stick platform; and
2. a novel, argument-based response to hypothetical request for proposals for development of a hypothetical system based on the cooling-tanks problem.

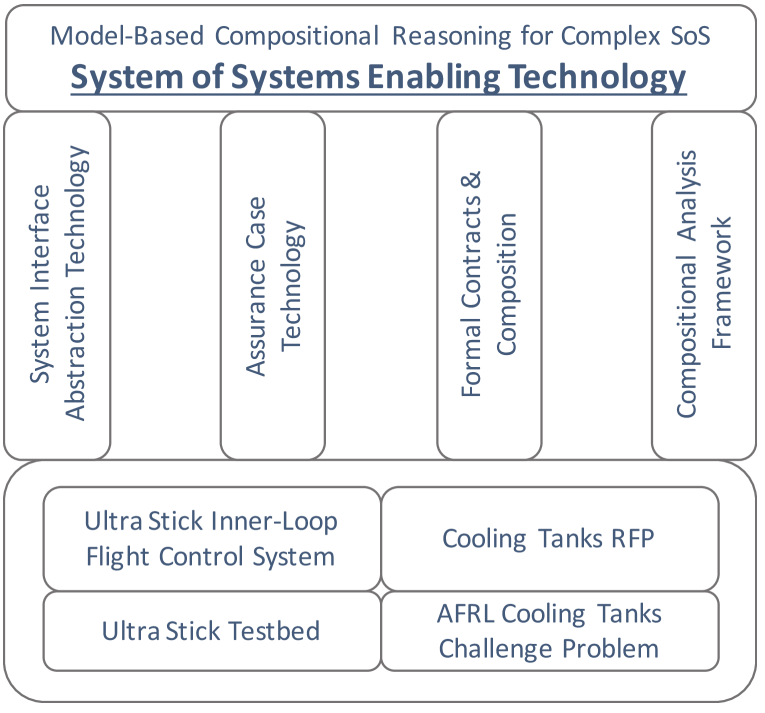


Figure 1: System of Systems Enabling Technology

3 METHODS, ASSUMPTIONS AND PROCEDURES

This section presents the development of system-interface abstraction technology, a system-of-systems enabling technology that provides a comprehensive infrastructure to support compositional reasoning and assessment of complex systems of systems. System-interface abstraction technology is comprised of a reference model, reference processes, and reference mechanics.

3.1 System Interface Abstraction Technology

Certification of complex systems and systems of systems is a significant challenge. Their scope and complexity makes reasoning about critical properties challenging and makes assessing the system for regulatory acceptance challenging. The only viable approach to such systems is to reason about them and certify them compositionally.

Compositional reasoning for complex systems and systems of systems requires two critical steps:

1. Showing that the demands of the architecture on its components satisfy system requirements; and
2. Showing that selected components satisfy the demands of the architecture.

These two reasoning steps provide compositional assurance that the system successfully satisfies its design goals.

Compositional certification for complex systems and systems of systems similarly requires two critical steps:

1. Arguing successful development by showing that architectural demands satisfy success goals; and
2. Arguing the compatibility of selected components with architectural demands and the compatibility of component context with system context.

These two argument steps, which naturally align with the reasoning steps, provide compositional certification.

These four steps require careful attention to several critical artifacts of systems engineering:

- requirements,
- context,
- architecture (or specification), and
- interfaces.

System interface abstraction technology provides support for compositional reasoning for certification of complex systems and systems of systems. The technology is composed of three parts:

1. **The *SIAT reference model***: describes the essential systems-engineering *artifacts* that are associated with successful development and the general relationships between these artifacts.
2. **The *SIAT reference development process***: describes general engineering *activities* that are undertaken to produce the reference model artifacts.
3. **The *SIAT reference mechanics***: describes example instantiations of the engineering artifacts and development activities.

The rationale of this division is to separate the SIAT theory from SIAT application. Generally, a reference model is a framework codifying goals/concepts and their interrelationships. A reference model does not specify a particular instantiation of these concepts but instead provides general organization and structure.

The reference development process provides more detail about how reference model concepts are used in a development effort; however, the process is still generally described to minimize coupling with specific engineering paradigms or tools.

The reference mechanics add further detail to the reference development process, describing specific development tools and techniques. Some of the mechanics are based on technologies developed as part of this Phase II effort, specifically, mechanics are based on an assurance case technology, theory of component contracts, and composition analysis tools, as shown in Figure 2.

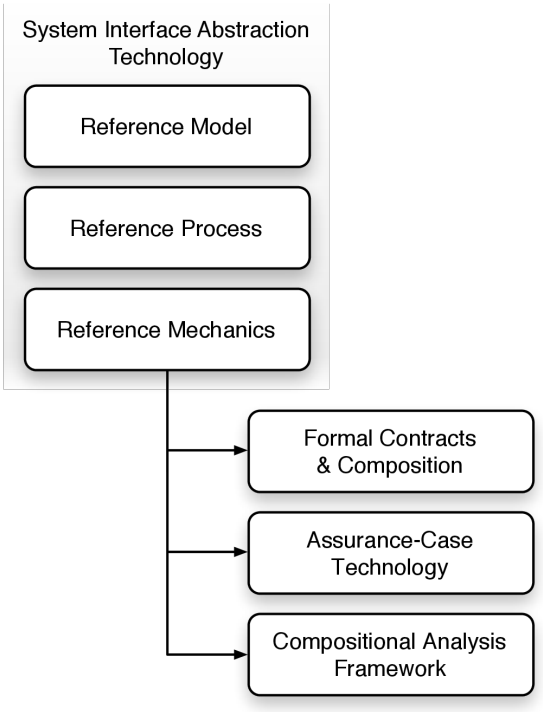


Figure 2: System Interface Abstraction Technology Overview

The *fundamental goal* of the three components of SIAT is to provide system developers with a practical framework for establishing justifiable assurance that the development effort is *successful*.

To aid the presentation of the technology, we start by considering a motivating example.

3.1.1 Motivating Example

System-interface abstraction technology provides the developer with clear identification of requirements, context, architectural demands, and interfaces. These are essential artifacts that support compositional reasoning and compositional certification of complex systems and systems of systems.

To illustrate system interface abstraction technology, we present and discuss a motivating example of a simple pitch-attitude monitoring capability for an unmanned aircraft.

3.1.1.1 Understanding the Problem

The customer has been flying a remotely piloted aircraft for a while and is concerned because sometimes the pitch attitude becomes excessive without warning. The customer might say to the developer, simply, "my pitch attitude becomes excessive without my realizing it," but, more likely, the customer will have already decided that he or she would like to have a monitoring system. The customer might therefore state his or her need like this:

A system is needed to provide pitch-attitude monitoring for a small, remotely piloted aircraft. The system should continuously monitor pitch attitude and alert the pilot when pitch attitude exceeds ± 45°.

We call this need a *problem*, and cast the development effort in terms of solving the customer's problem. A *successful development effort* solves the customer's problem completely — or sufficiently completely that the customer will be satisfied. Integral to solving the customer's problem are business-related concerns — such as managing the effort so as to complete the solution on budget and on time — and regulatory concerns — such as adhering to any regulations that might inhibit the customer's use of the solution.

Often contrary to initial statements, which focus entirely on functionality, assurance is integral to the problem. The first step in solving a problem is therefore understanding the problem as completely as possible. This means that the statement of functionality must be refined, the overall system context must be defined, and assurance must be defined. The developer, upon receiving the problem statement, might ask questions like:

- Is there additional functionality desired that has not been expressed?

Or:

- Is the aircraft fitted with a pitch sensor?
- Does the aircraft system have an alerting capability?
- Does the aircraft have a computer system that can be used to conducting monitoring and activate the alert?

Or:

- What level of assurance is required?
- Does the assurance arise as a result of regulations?

Upon receiving answers to these questions, the refined problem statement might be written:

A system is needed to provide pitch-attitude monitoring for a small, remotely piloted aircraft. The system should continuously monitor pitch attitude and alert the pilot when pitch attitude exceeds ± 45°. The system is comprised of a pitch sensor, a computer, and an alarm. The pitch sensor provides input to the computer through memory-mapped input/output (I/O). The alarm is activated by the computer through memory-mapped I/O. The system must provide at least five nines of availability (99.999% availability).

3.1.1.2 Defining the Solution

The problem statement is equivalent to a set of very high-level requirements: it captures the essence of what the customer wants. But requirements are best thought of as being about the solution, describing what functionality/behavior the system must provide, in terms of changes to the environment, to solve the problem. We therefore separate the problem statements from the requirements. Additionally, there is typically more than one set of requirements for a given problem statement, just as there is more than one possible implementation that satisfy a given set of requirements.

The second step in solving a problem is identification of the context in which the problem exists and the solution must operate. This identification actually takes place concurrently with understanding the problem, extends into the identification of solution requirements and continues through design and implementation. The context consists of the physical environment and potentially other factors, such as regulatory constraints.

The third step in solving a problem is to work with the customer to identify the requirements that will constrain the solution to the problem. This task is among the most difficult in system and software engineering, as it is fundamentally informal and fraught with opportunities for misunderstanding and miscommunication. This task is also among the most important in system and software engineering, as errors introduced in requirements are exponentially more expensive to fix than errors introduced later in the development cycle. Working from a clear understanding of the problem helps to ensure that requirements are accurately and completely identified, but does not guarantee it.

The requirements for the pitch-attitude monitoring system might include the following:

The system shall raise the alarm when pitch attitude is greater than 45° or less than –45°.

The concurrency of solution requirements and solution context identification is essential because the requirements are phrased in terms of context. This is clear from the example above:

- *raise the alarm* refers to the alarm — a part of the identified context, as stipulated in the refined problem statement
- *pitch attitude* refers to a state of the aircraft — a part of the environment from the point of view of the monitoring system, because it is beyond the system boundary
- the thresholds also refer to the state of the aircraft

Requirements describe how the environment should change to solve the problem. Additionally, elements of the solution context may contribute requirements that are not explicit in the problem statement. The regulatory environment, in particular, is a frequent source of additional requirements.

3.1.1.3 Developing the Solution

The fourth step in solving a problem is, broadly speaking, building the solution. This step includes specification, design and implementation.

The specification is a high-level description of how the system will satisfy its requirements. Ideally, the specification should be sufficiently abstract to support multiple paths of implementation. In this respect the specification can be thought of as “what” the system must do to satisfy its requirements, and further design specifies “how” the specification is satisfied. The

distinction between “how” and “what” is often unclear and dependent on point of view [4]. The specification for the pitch-attitude monitoring system might include the following:

The system shall issue the alarm command to the transmitter when the pitch input provided by the pitch sensor is greater than 45° or less than 45°.

3.1.1.4 Identifying Solution and Problem Discrepancies

Careful examination of the example from the specification and the example from the requirements reveals a problem. The requirements are stated in terms of the environment, but the specification is stated in terms of the system. There is an implicit assumption encoded in the specification that “the pitch input provided by the pitch sensor” is the same as the “pitch attitude” described in the requirements.

This assumption is fundamentally flawed. While it is *desirable* for the pitch input from the sensor to be the same as pitch attitude, they are in reality distinct phenomena. The input received from a properly designed, properly installed, and properly functioning sensor should closely correspond to the phenomenon it observes, but it cannot be relied upon to perfectly represent that phenomenon at all times. Instead, the input from the sensor approximates the phenomenon, but is subject to latency, inaccuracy, imprecision, range limits, etc.

Without identifying the flaw in this assumption, the validation effort might conclude that the specification satisfies the requirements simply because it includes the same thresholds. The result would be a system that, once fielded, sometimes satisfies the requirements but sometimes does not. If the customer considers ±45° pitch to be a hard limit, the system will not satisfy the customer and development will be unsuccessful.

3.1.1.5 Validating the Solution

Validating that the specification satisfies the requirements therefore requires identifying the correspondence between elements of the specification and elements of the environment. In this example, the pitch input from the pitch sensor corresponds to the pitch attitude of the aircraft. The identification of the correspondence includes identifying what is lost in the approximation of the pitch attitude by the pitch sensor, to include accuracy, precision and latency.

If the pitch sensor promises ±1° accuracy, then the system may not warn the pilot until the pitch attitude has exceeded ±46°. If the pitch sensor furthermore promises ±2° precision, the warning may not take place until the pitch attitude has exceeded ±48°. Moreover, if the pitch sensor promises 0.5 seconds latency, the response may be delayed even further. Exactly when the warning will be issued will depend on the dynamics of the aircraft and how quickly pitch attitude can change in 0.5 seconds.

Having identified the approximation of pitch attitude by the pitch sensor in the correspondence, it is now clear that validation cannot succeed. In identifying the requirements, we have unwittingly stipulated a requirement that cannot possibly be satisfied. We cannot show that the specification will satisfy the requirement because, most of the time, it will not.

There are two solutions to this problem:

1. We modify the requirement so that it is satisfiable;
2. We change our understanding of validation to include the necessary approximation made by the sensor.

The first approach is much better than the second approach. While the second approach has the advantage of being simpler and more expedient, it prevents the issue from being clearly documented and explained. If instead we modify the requirement, we will clearly document the problem of approximation of pitch attitude by the sensor. Doing this will force us to consider other requirements related to pitch attitude — we are thus likely to identify and correct all of the related requirements issues at once. A risk in modifying the requirement, however, is that the new requirement is overly tailored to the details of the solution. We must be careful to express the revised requirement so that the new requirement admits the necessity of approximation but does not depend upon or assume a specific approximation.

3.1.1.6 Refining the Problem and Solution

There are a number of ways in which the revised requirement might be stated. The exact phrasing will depend upon the needs of the customer — in particular, how precise the customer requires the pitch-attitude monitoring to be. Simple, straightforward phrasing of the requirement is no longer possible: acknowledgement of the approximation requires consideration of false positive as well as false negatives for the alarm — even when all parts of the system are operating correctly.

While we might be tempted to say that the rate of false positives and false negatives is a design detail, the discussion above shows that it is not. The customer may not be used to thinking of these kinds of details, but they are fundamentally part of the requirements that define the solution.

In this example, the customer may not be overly concerned with precision, and may accept requirements that say:

The system shall raise the alarm when pitch attitude is greater than 45° or less than –45°. The system shall not raise the alarm when pitch attitude is less than 40° or greater than –40°.

These requirements state the limit of acceptability for false positives and false negatives without resorting to probabilities — probabilities that, mostly likely, the customer does not know. Between these stated limits, the behavior of the system is not constrained. The alarm may be raised or not and the requirements will still be satisfied.

With a sufficiently accurate and precise sensor, with sufficiently low latency given the dynamics of the aircraft, we can ensure that an alarm is never raised when it should not be and is always raised when it should be — provided that the system is working correctly.

3.1.1.7 Summary and Conclusions

The steps described above correspond to typical systems engineering activities. The artifacts and processes described map to those identified by system interface abstraction technology. With this example in mind, we present the system interface abstraction technology rigorously, in the following sections.

3.1.2 Reference Model

The prior motivating example speaks to the underlying difficulties in development as largely an issue of understanding the problem and its environment, and assessing the solution with respect to the identified problem and environment. The SIAT reference model is largely based upon

these observations and upon prior related work on problem-oriented development approaches [7] [8] [9].

Generally, a problem oriented reference model consists of four abstract components:

1. the Problem;
2. the Environment;
3. the Solution; and
4. the Argument.

The Argument tells us that the Solution in its intended Environment solves the Problem.

The SIAT reference model extends the basic problem-oriented concepts to include the following artifacts:

1. the Problem;
2. the Requirements;
3. the Context;
4. the Correspondence;
5. the Specification;
6. the Implementation; and
7. the Argument.

The extensions of the SIAT reference model are as follows:

- **Separation of the requirements from the problem**: Traditionally, requirements define a problem to solve; however, elicited requirements might not define the right problem. Often development begins with an abstract problem description that is further refined into requirements. The SIAT model separates the abstract problem from the requirements to better align with how systems are developed and to explicitly address the risks associated with abstract problem identification and requirements elicitation separately.
- **Replacement of the environment with context**: SIAT defines a more general notion of context that subsumes the physical environment, providing additional and important information for development beyond what is defined in terms of physical phenomena. Regulation, for example, is part of a system's context but not necessarily its physical environment.
- **Addition of correspondence**: Correspondence is an explicit relationship between phenomena of the real-world and phenomena specific to the solution. Correspondence is helpful in justifying that an implemented solution actually solves the problem by showing how phenomena are related.
- **Refinement of the solution into the specification and implementation**: The "solution" is essentially the implemented system; however, solutions are not engineered directly from the problem in practice. Refining the concept of "solution" into the specification and implementation is more aligned with how systems are developed. A specification is developed to satisfy requirements, and an implementation is developed to satisfy the specification. Risks associated with development of the specification and the implementation are therefore crucial in justifying the solution system solves the problem.

The result of this separation is the ability to make a more compelling and comprehensive argument. Specifically, the argument tells us that:

1. the Problem, Context and Requirements are adequately defined and

2. the Implementation of the Specification in Correspondence with its intended Context satisfies the Requirements of the solution to the Problem.

We refer to the assurance goal implied by this kind of argument as **successful development**.

The remainder of this section further discusses the details of each SIAT reference model artifact. Questions concerning the production of these artifacts during development are further addressed in Section 3.1.3.

3.1.2.1 Terminology

Before describing each of these components of the reference model in detail, we first introduce some definitions for recurring concepts.

Phenomenon A phenomenon is an observable entity. Examples of phenomena include events, values and relationships.
Variable A variable describes a phenomenon that has values.
Type A type describes the set of possible values for variable.
Instance An instance is a particular value of a given type.
Domain (or Knowledge Domain) A community of like-mindedness and shared mental space. A domain refers to the knowledge ecology of an expertise or field. Although a domain can be partially captured in artifacts documenting regulations, protocols, operational definitions, relevant phenomena, etc., domains are abstract concepts, defined by a collective of mental models and social convention/agreement of experts.
Optative Optative expresses realizable intention or desire.
Indicative Indicative expresses a statement of fact.
Successful Development System development is considered successful if (1) the problem is adequately defined and (2) the problem is solved. More specifically, the problem is solved if the system implementation of the system specification in correspondence with its intended context satisfy the requirements of the solution to the problem.

3.1.2.2 The Context

The context is a comprehensive and indicative description of constraining aspects of the world that the system under development will operate in. Context typically cannot be fully documented due to scope. Instead, context is defined largely based on domain specific reckoning of context. Context is therefore defined operationally; however common entities of the context include the physical environment, the system domain, staff/support infrastructure, maintenance infrastructure, regulations, etc.

The concept of context is an extension of environment or world (the physical environment) within a problem-oriented approach. Since all systems operate within the a physical setting, the physical environment is a mandatory component of the SIAT context. The rationale for extending the physical environment into the notion of context is the observation that often other indicative factors that are not easily expressed as physical phenomena constrain development. For example, the domain in which the system is developed and regulations.

The environment is a highlighted component of the context within the SIAT reference model, not only because the environment is a mandatory component, but the environment the perhaps the most pervasive element of context throughout the system development. Regulatory context, for example, influences requirements and constraint design and implementation, but environment

influences all phases of system development. Further, a system is typically alters some phenomena of the physical environment in some manner to solve the problem, whereas other components of the context are not altered by the system. We therefore provide a more rigorous definition of the environment as follows:

The **Environment** includes a set of related *Indicative Phenomena* that are usually treated as a unit in problem analysis. The *Types* associated with the *Phenomena* of the **Environment** exist in the world outside of the *System*. These *Environment Types* are sometimes also called *Real-World Types*.

3.1.2.3 The Problem

In SIAT, the problem captures the essence of the customer's need as simply and succinctly as possible with respect to the context. Hence, the problem in SIAT is an abstract problem description and/or a set of abstract requirements, not a completed requirements document. The rationale is to first focus the abstract problem and then refine the problem into requirements in subsequent development activities.

We think of the problem as being the first component of the SIAT reference model as it drives development, but it cannot be defined without reference to its context, and, in particular, its physical environment. This is natural, as the problem emerges from the context and is typically defined to alter some set of phenomena of the environment. Rigorously,

A **Problem** describes an alternate optative **Environment** in which some *Phenomena* differ from those in the actual, indicative **Environment** as defined in the *Context*.

3.1.2.4 The Requirements

The requirements refine the identified problem, constraining the solution space of the problem further by identifying elements of the solution that are of particular importance to the customer. Rigorously,

Requirements express the solution to the **Problem** in terms of *Variables* or *Values* of the *Phenomena* shared between the **Problem** and the **Environment**. Because **Requirements**, through **Problem** they solve, express a possible, future **Environment**, they are an *Optative* description.

While the problem speaks about the future environment in simple, succinct terms, the requirements speak about the future environment in detail. Because the requirements are restricted to environmental phenomena, engineers are limited to describing *what* the solution will accomplish. Without reference to the phenomena of the system, engineers are precluded from saying *how* the system will solve the problem.

3.1.2.5 The Specification

The specification is the beginning of the realization of the solution requirements. Rigorously,

The **Specification** describes an *Optative* set of *Phenomena* that is the *System*. The *Types* associated with the *Phenomena* of the **Specification** exist in the *System*. These *System Types* are sometimes also called *Machine-World Types*.

While the requirements speak about the future environment in detail, the specification speaks about the future system in detail. The specification is restricted to system phenomena, limiting engineers to describing the system that will solve the problem.

3.1.2.6 The Correspondence

The requirements express the solution in terms of environmental phenomena. The specification expresses the solution in terms of system phenomena. In order to show that the specification indeed satisfies the requirements, and thus solves the problem, the relationship between system phenomena referenced in the specification and environmental phenomena referenced in the requirements must be described. The correspondence expresses this relationship. Rigorously,

The *Correspondence* between *System Types* and *Environment Types* is made through an *Indicative* description of the **Environment**. This description provides a *Correspondence Model* of critical relationships amongst *Phenomena* that are shared between the **Environment** and the **Specification**.

3.1.2.7 The Implementation

The implementation is the realization of the specification and by extension the solution requirements. Rigorously,

The **Implementation** expresses the realization of the **Specification** in terms of the *System Types* introduced in the **Specification**. Additionally, the **Implementation** may rely on hidden *Phenomena* of the *System* — that is, *System Phenomena* that were not shared between the **Specification** and the **Environment** and are not described in the **Correspondence**.

The specification can be conceptualized as a high-level system design. During implementation, the specification is further decomposed into a detailed design until the level of granularity is sufficient to build the actual system.

3.1.2.8 The Argument

In SIAT, the reference model artifacts are tied together by the last reference model artifact, the argument. The argument provides the rationale for belief that the problem has been identified and the implemented system successfully solves the identified problem. Rigorously,

The **Argument** is an explicit and comprehensive logical structure, supported by a body of evidence, justifying that the developed/implemented system is *Successful*. Successful development requires not only a justification that the developed system solves the identified **Problem** within the given **Context** (e.g., Environment and regulatory considerations), but also requires justification that the **Problem** and **Context** are correctly, completely and appropriately identified.

Where possible, the argument can be based on deductive/formal logic; however, requirements are fundamentally informal and doubts exist about the fidelity of formal models. Consequently, the argument is primarily informal based on inductive logic.

3.1.3 Reference Process

To better situate the reference model within a development process, this section describes the SIAT reference process. The reference process defines a set of high-level SIAT development activities. The purpose and rationale of the reference processes is to organize general engineering

activities associated with the application of SIAT and to relate these activities to the reference model artifacts defined in the previous section. The reference process grounds the reference model in abstract development activities, providing more structure/guidance for the application and further discussion of the SIAT concepts. The reference process also introduces basic concepts for the composition of components in system development, to be the focus of later sections. The reference process does not provide detailed descriptions of development activities, but rather describes the general activities that surround the reference model artifacts. In general, reference processes produce reference model artifacts as outputs.

SIAT, as a problem-oriented approach, can be conceptualized in terms of three primary development activities (shown in Figure 3):

1. understanding the problem,
2. developing the solution, and
3. assessing the solution.

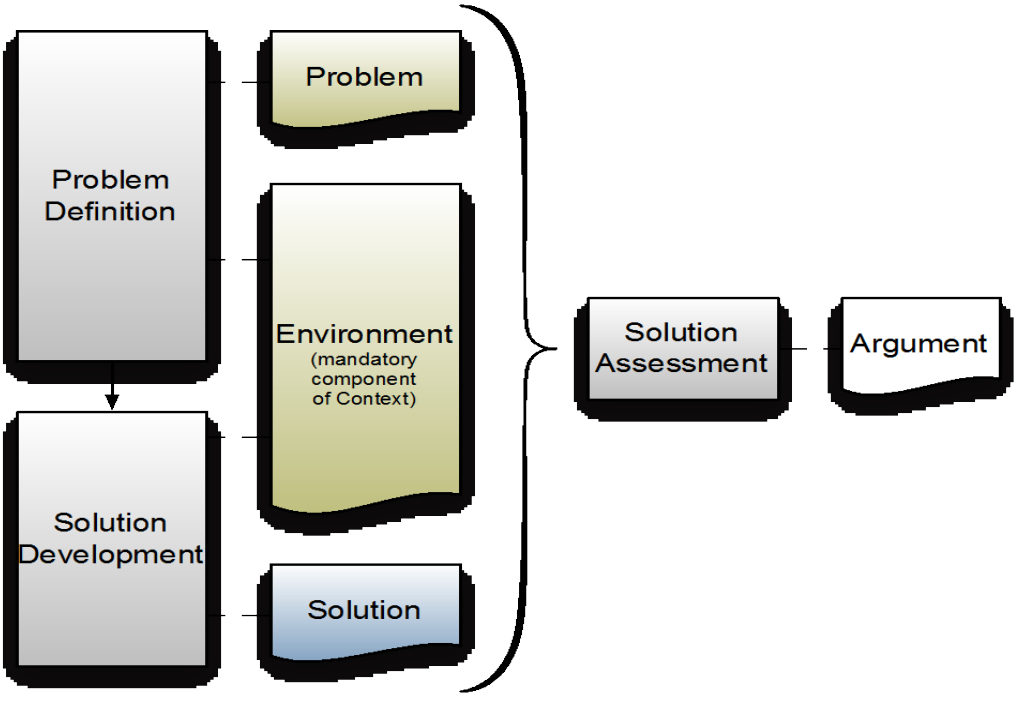


Figure 3: High-Level Reference Process

While these activities speak to the rationale underpinning a general problem-oriented approach, they are too abstract and do not provide sufficient granularity to align with all of the SIAT reference model artifacts. The SIAT reference process is therefore refined into five development activities to expose the relationship between reference model artifacts, and further, to expose the mechanics of composition (shown in Figure 4):

1. Problem Definition;
2. Solution Definition;
3. Solution Specification;
4. Solution Development;
5. Solution Assessment.

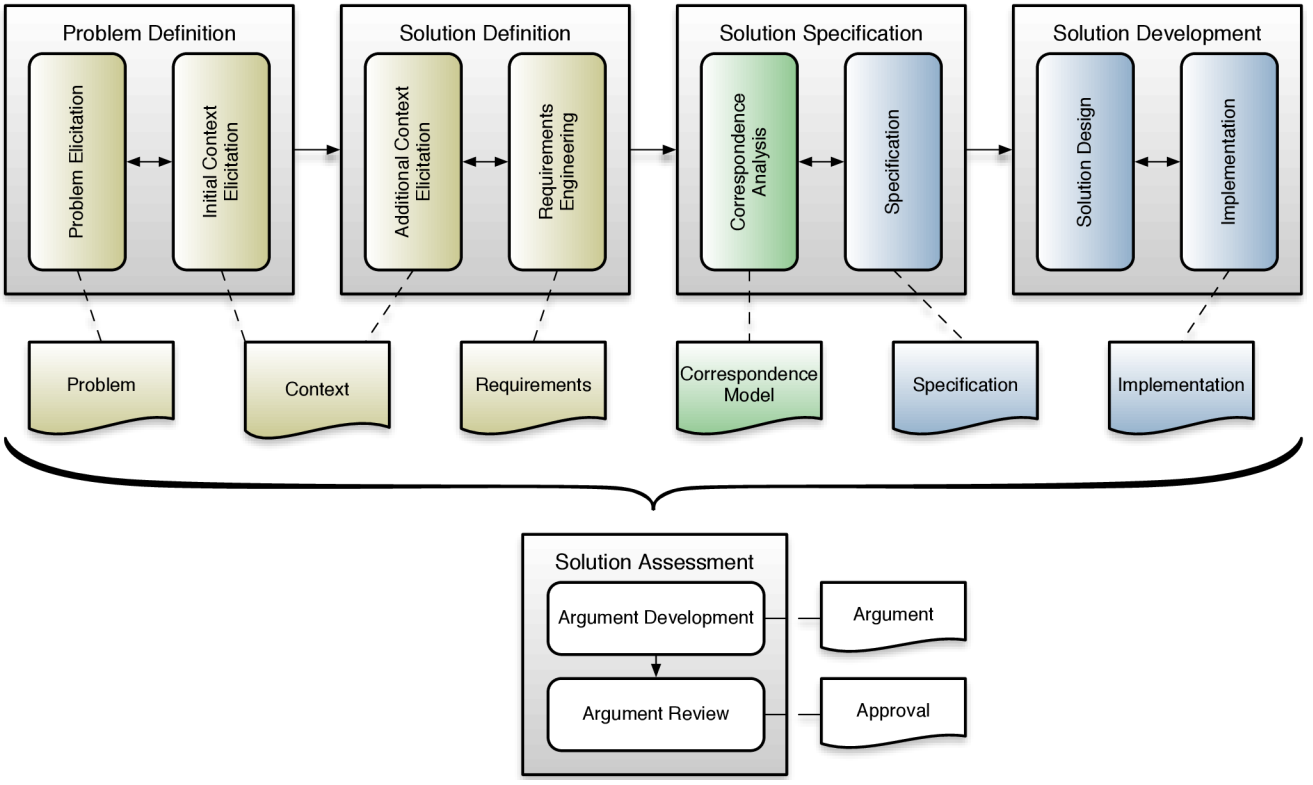


Figure 4: Detailed Reference Process

The remainder of this section further describes the detailed processes and relationships shown in Figure 4. To simplify the discussion, and to separate out the mechanics of the adoption of these processes, the model is described as a linear progression of activities (a waterfall model). Feedback loops between any set of processes, either for development or to accomodate future system change and maintenance, are neither precluded nor mandated by the reference process.

3.1.3.1 Problem Definition

The problem, as described in the reference model, is the fundamental driver of system development in SIAT. The first process component of the reference process is therefore the activity of defining the problem. The inputs to this activity are any initial customer ideas about the problem and knowledge about the problem's context. In some cases, the context and/or problem may be previously or partially defined. The output artifacts are the problem (the abstract problem description) and an an initial description of the context, which are conceptually developed in two subprocesses:

1. problem elicitation and
2. initial context elicitation.

A specific order in which the outputs are produced is not assumed. The problem and context will likely be elicited concurrently. The produced context should be complete with respect to the identified problem, but, as is discussed in more detail below, the context is considered "initial" since further refinement of the problem in subsequent development activities often reveals deficiencies in the originally developed context.

3.1.3.2 Solution Definition

The second process component is the definition of the solution. The primary purpose of solution definition is to engineer system requirements by refining the previously identified problem; however, in the process of developing system requirements, engineers often reveal more information about the problem necessitating refinements to the context. For example, a new requirement may reference an environmental phenomenon that was never defined, or reference functionality necessitating further explication of relevant regulations. Solution definition therefore consists of two subprocesses:

1. continued context elicitation and
2. requirements engineering.

The problem and the initial context are inputs from the problem definition activity. The solution requirements and the context are outputs.

The formality by which requirements are documented is not specified in SIAT. Requirements can sometimes be refined into formal descriptions, but not all requirements are amenable to formalization (e.g., usability requirements). Whenever possible, formalizing requirements is advisable to avoid issues associated with the ambiguity of natural language.

3.1.3.3 Solution Specification

The third process component of the reference process is the specification of the solution behavior. For this process component, the requirements and the context are inputs from the solution definition activity. The specification and the correspondence are outputs, produced by means of two subprocesses:

1. correspondence analysis and
2. specification.

The specification development activity produces the specification artifact that stipulates the how the solution system will be developed to satisfy the system requirements. Correspondence analysis examines those phenomena in the environment that must be accessed by the system under development and relates those environmental phenomena to system phenomena for use by the specification. Correspondence analysis can be performed after the specification is produced, but could also be performed iteratively or concurrently with the specification activity as system phenomena are identified within the specification artifact.

3.1.3.4 Solution Development

The fourth process component is the development of the solution. In this process component, the specification is refined through a detailed system design activity, resulting in detailed design demands. Design demands are traditionally specified to a level of granularity sufficient to allow system engineers to directly implement a solution to satisfy the demands. Conceptually, solution development consists of two subprocesses:

1. solution design and
2. implementation.

SIAT, however, also supports satisfying design demands by integrating other modular components. During design, developers assess the possible benefits in integrating other modular components into the implemented system rather than implementing the system in house. For

example, developers may find cost and organizational benefits by using existing and reusable components, or by further decomposing the solution into subproblems to be compartmentalized and developed in parallel development efforts.

The use of components to satisfy design demands implies a third solution development subprocess, component integration, see Figure 5. The implemented system is produced by some combination of local (in house) implementation and component integration. The exact distribution of local implementation and component integration is based on system-specific design decisions. Design demands earmarked to be satisfied by components are said to be *delegated* or *allocated* to components.

The component integration activity requires the selection or development of components. Components may or may not be previously developed, developed by third parties or developed under the concepts of SIAT. Regardless of the origin and development methodology of components, justification is required to demonstrate that the composed components are compatible and that the design demands delegated to components are satisfied by the provided behavior of referenced components.

If components are developed under SIAT, the relationship between development processes is shown in Figure 5. General compatibility is assessed through assessment and comparison of contexts. Delegated demands are satisfied by contractural agreement between the delegated demand and component requirements. This approach keeps to the SIAT problem-oriented approach, hiding component detailed design from high-level components, promoting flexibility in component development. Justification of general compatibility and contractual satisfaction becomes part of the assurance argument developed in the solution assessment process. Component systems may themselves be implemented using components recursively.

Component integration is both a development and maintenance activity. Component integration is performed post system deployment in response to changes to components.

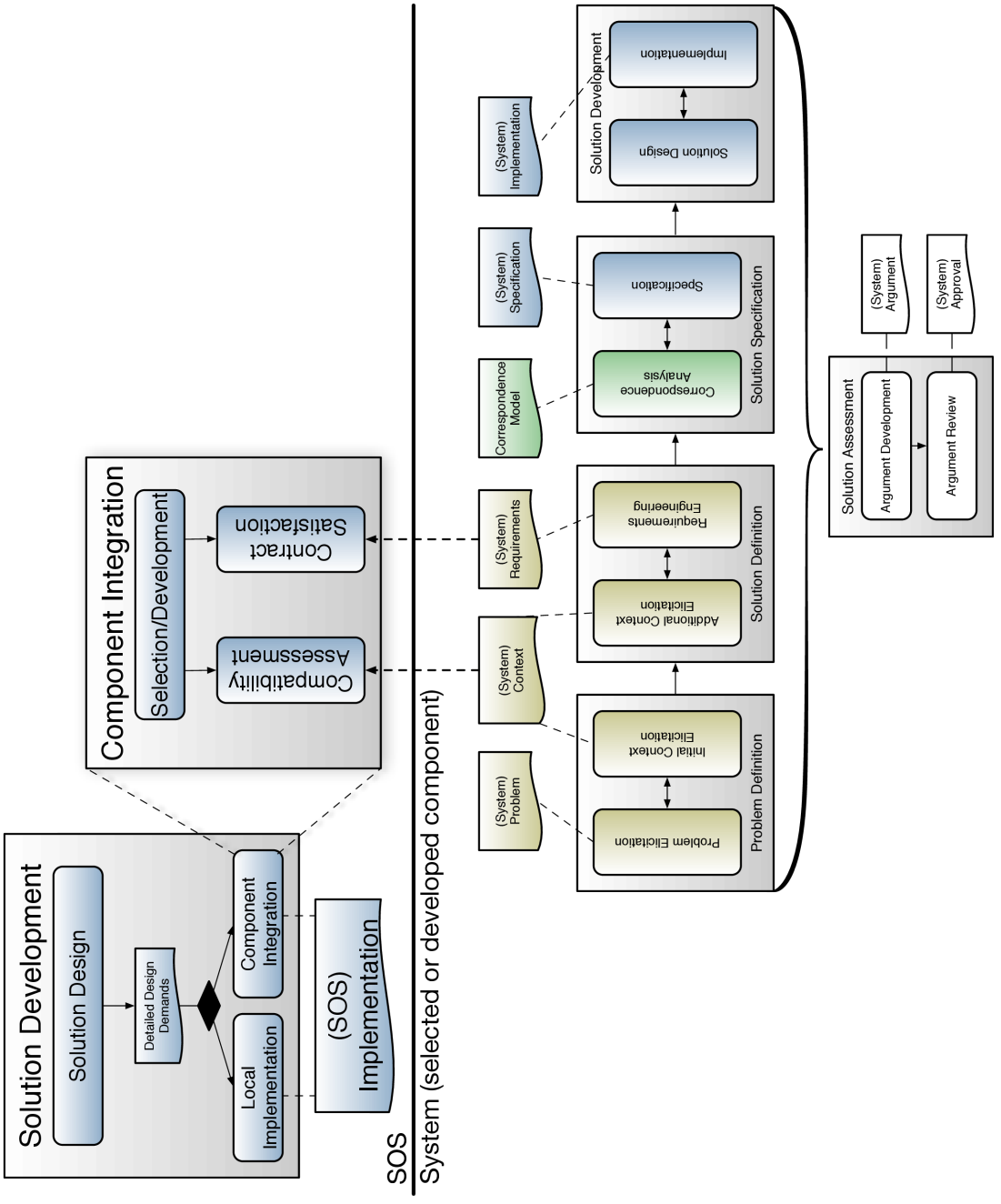


Figure 5: Recursive Component Development and Integration

3.1.3.5 Solution Assessment

The final process component is assessment of the solution. The goal of this process component is to verify and validate that the implemented system is a solution to the problem. The inputs to this process are all of the reference process outputs: problem, context, requirements, correspondence, specification, and implementation. Solution assessment also uses fine-grained details about each development process and development artifact as necessary. For example, if components are integrated into the solution, solution assessment also uses details of the compatibility assessment and contract satisfaction. The outputs of solution assessment are an assurance argument and

documented approval of the implemented solution based on a careful review of the argument. Solution assessment therefore consists of two subprocesses:

1. argument development and
2. argument review.

The argument is a comprehensive justification that the development process was *successful*: the solution system (the implementation) solves the problem, and the problem (the problem description, context, and requirements) has been adequately identified. The argument can be iteratively and incrementally developed in parallel with all other reference model processes [10]

Because of the scope and purpose of the argument, argument development can serve as an initial and incremental assessment of the implemented system: assessments are required at each level of argument development to justify claims. The argument, however, is an artifact documenting belief that the development is successful. Final approval of the solution system is still necessary prior to system deployment.

A domain-specific authority is designated to review and confirm the solution system is indeed successful. Since the argument captures the comprehensive belief that the system is successful, review of the system is primarily a review of the argument (argument claims and supporting evidence). The reviewing authority can be the customer, domain-specific experts and/or other 3rd-party stakeholders (e.g., certification authorities). At the end of the review activity, the solution is either approved for use, or deficiencies are identified necessitating refinements in earlier reference processes.

3.1.4 Reference Mechanics

The reference mechanics gives specific instantiations of process components and specific forms to model components. This section describes several possible reference mechanics. Some of the presented mechanics are further investigated as part of this Phase II effort, and discussed in further detail in subsequent sections.

3.1.4.1 Problem Frames

The basis underlying SIAT's "problem-oriented" theory and philosophy originates from the concept of Problem Frames [8]. Problem frames are a theory and set of mechanics for structring and analyzing problems, based on the clear separation of problem, environment and solution.

While the problem frame concept specifies particular mechanics and notation for analyzing problems, SIAT borrows more from the underlying motivation and theory than from these mechanics. In particular, SIAT is motivated by the observation that system failures often occur as a result of invalid requirements that originate from an improper understanding of the problem and its environment.

As Jackson notes, practically all engineers and practitioners agree that focusing on the problem and *what* the system must do, not *how*, is of the utmost importance in early system development [8]. He also notes that this is not a useful moto. There are difficulties in distinguishing between the problem and its solution. The problem is located in the real world, and it is often difficult to focus on the problem. Engineers instead focus on where the solution to the problem is located: the system to be developed. The substance of Jackson's problem frame approach centers around clearly identifying and separating the problem, the environment and the system.

Problem frames describe a problem and its solution using a set of canonical frames, which represent patterns commonly seen in developing software. The most basic form of the problem frame is shown in Figure 6.

Figure 6: The Basic Problem Frame

A key concept adopted for SIAT from problem frames is the *frame concern*. Jackson defines the frame concern as "*the central concern for problems of a class defined by a problem frame*" [8]. More generally, the frame concern summarizes the argument that is associated with a specific problem frame. At the highest level, the frame concern can be read as: "the system and its environment satisfy the requirements". To *address* or *satisfy* the frame concern is to justify that the problem is adequately defined and solved by the developed system.

The frame concern makes explicit the distinction amongst three fundamentally different descriptions:

1. the specification – the optative description of what the *machine* (the system under development, further defined in subsequent sections) does to solve the problem;
2. the domain description — the indicative description of the causal relationships in the domain upon which the machine relies to solve the problem; and
3. the requirement(s) — the optative description of what is required to solve the problem.

The argument implied by the frame concern is that:

1. the specified system behavior (M)
2. combined with the given environment/context (W) produces
3. the required behavior (R).

More formally, the argument stipulates that the system, as developed within its operating context, *entails* the requirements ($M \wedge W \vdash R$). This argument concept, combined with similar extensions of problem frames in a related reference model [9] provides the basis by which arguments are structured in SIAT.

SIAT separates Jackson's theory from the mechanics of problem frames. SIAT primarily makes use of the theory underlying problem frames for problem analyses, and uses these concepts to govern the form of an assurance argument and the mechanics for developing and assessing the argument. The mechanics associated with problem frames and context diagrams (the specific diagram structures) can be used when analyzing problems and context, but it is not a requirement of our approach.

3.1.4.2 Assurance Cases

Arguments in SIAT are documented using *assurance cases* noted using Goal Structuring Notation (GSN). An assurance case is a reasoned and compelling argument, supported by a body

of evidence, that a system, service or organization will operate as intended for a defined application in a defined environment [22]. A further discussion and background on assurance cases and GSN is described in Appendix A.

3.1.4.3 The Toulmin Model

Assurance cases documented in GSN are based on the Toulmin model of argumentation [11].

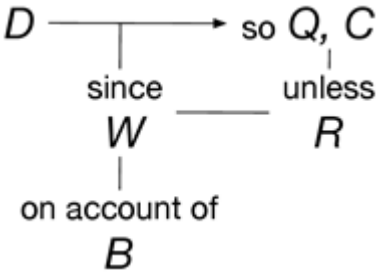


Figure 7: The Toulmin Model

The Toulmin model, illustrated in Figure 7, is comprised of the following components:

- **Claim** (*C*): The position, claim or conclusion "*whose merits we seek to establish*". Claims are represented as goals in GSN notation.
- **Qualification** (*Q*): Modal qualifiers (e.g., 'most', 'usually', 'presumably', or 'always') that constrain the scope of the claim.
- **Data** (*D*): The data or grounds used as evidence in support of a claim. Data is represented by solution elements within GSN notation.
- **Warrant** (*W*): Practical standards or cannon of argument providing the rationale by which the data provides support of a claim. Warrants are typically represented as strategy elements in GSN notation between two goals; however, in traditional GSN notation, warrants are not expressed between a goal and solution element.
- **Backing** (*B*): The support, justification or authority that supports the validity of the warrant. Often backing is not explicitly expressed within an argument, and is instead resident in the implicit knowledge endogenous to the domain in which the argument exists.
- **Rebuttal** (*R*): Scenarios and exceptions that undermine the validity of the claim and authority of the warrant. There is no standard by which rebuttals are documented with GSN arguments; however, the concept of a confidence argument [12], provides a mechanisms for separating out arguments justifying the mitigation of rebuttal scenarios.

The Toulmin model concepts provide foundation for comprehensive inductive reasoning, and further provide a foundation for novel modularity support (e.g., composition schemes, further discussed in Section 3.3).

3.1.4.4 Modular Design

Modularity is often proposed as a mechanism to combat the growing complexity of system and software design. A large, complex problem is recursively broken up into smaller, simpler problems, until either the level of complexity has reached manageable levels or the problem can

no longer be easily decomposed. Additionally, modularity can facilitate reuse: when components are suitably decoupled from one other, they often can be used to construct new, different systems.

Critical principals of modularity include [13] [14]:

- low coupling,
- high cohesion, and
- information hiding.

Modularity is a key mechanism in SIAT. SIAT proposes that components be developed with carefully identified and described interfaces that are based on assume-guarantee reasoning (see Assume-Guarantee Contracts, below). Additionally, SIAT proposes that the arguments that the frame concern has been satisfied be structured to facilitate later use (see Section 3.3). When composition is used in system development SIAT provides reference processes to support compositional reasoning and to support compositional argumentation that component composition is correct.

3.1.4.5 Assume-Guarantee Contracts

Assume-guarantee reasoning is a common approach for compositional reasoning. Typical uses of assume-guarantee reasoning occur in component-based software engineering and design by contract engineering paradigms. In these paradigms, software modules (such as an application, object, or function) stipulate pre-conditions and post-conditions. Software modules provide properties/behaviors as specified by explicit post-conditions (i.e., guarantees). Modules may also stipulate pre-conditions (i.e., assumptions) associated with each guarantee. These assumptions must be valid in order to provide the corresponding guarantees. Thus, traditionally, valid assumptions imply the provided guarantees ($A \Rightarrow G$).

Assume-guarantee pairs for a module are often referred to as assume-guarantee *contracts*, resulting in some confusion in terminology. With respect to a more natural and common use of the term, contracts are better thought of as an *agreement* involving at least two modules (or in the legal sense, parties). Typical uses of the term "contract" within assume-guarantee reasoning only consider one side of an agreement.

In SIAT, the term "contract" refers to a mapping between the demands of one module to the provisions (guarantees) of another. The individual demands and provisions of a module together with their associated provided and assumed contextualization are referred to as *interfaces*. All interfaces specify some concept of an assumption and a guarantee (discussed in more detail below). A *contract* is therefore defined as an agreement between module interfaces.

A contract is considered valid if all involved module interfaces are satisfied. Satisfaction of an interface is achieved by meeting the assumptions of the interface with the guarantees of another interface; hence, the contractual relationship is bidirectional (see Figure 8).

Ideally, the interfaces in an assume-guarantee contract will specify *all* assumptions and *all* guarantees, thus completely describing the context in which the component exists. In practice, this is difficult to do completely and even more difficult to do formally. Argumentation provides a compelling mechanism to address this challenge, and is discussed in Section 3.3.

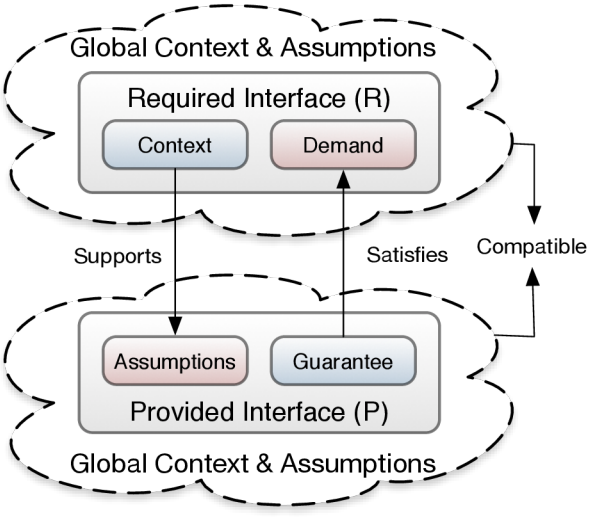


Figure 8: Bidirectional Interface Support

In a hierarchical decomposition of modules, contracts are formed between two modules at different levels in the hierarchy (one higher than the other — See Figure 8). This hierarchical relationship imposes certain roles on module interfaces depending on which side of the contract a module is found:

- The module that is higher in the hierarchy serves a *demanding*, *promisee*, or *consuming* role, specifying a "*required interface*": certain demands are *assumed* to be met by other components where the context of the demand is *guaranteed*.
- The module that is lower in the hierarchy serves a *providing* or *promiser* role, specifying a "*provided interface*": the module specifies *provisions* or *guarantees* if certain contextualizing *assumptions* are valid.

Both interfaces have a concept of an assumption and a guarantee. The key difference between the required and provided interfaces is the syllogism between each interface's assumptions and guarantees. In the required interface, guarantees (the system contextualization) imply assumptions (assumed satisfaction of demands) ($G \Rightarrow A$). Conversely, in the provided interface, assumptions (the assumed context) imply guaranteed behavior or properties ($A \Rightarrow G$). Guarantees of one interface serve to validate the assumptions of the other, see Figure 8.

Interfaces document formally the syntax and the semantics of what the component assumes or requires of the system and its environment and what the component guarantees or provides to the system and its environment.

Syntax is concerned with machine-world representations and is easily written down and checked. For example, a 12-element vector of 64-bit floating-point numbers is a statement of syntax. It is very easy for a traditional type checker to ensure syntactic compatibility of components.

Semantics are real-world concerns and are harder to write down and harder to check. Semantics are described in terms of real-world phenomena. For example, pitch attitude or airspeed are real-world phenomena. Usually, semantics are tied to syntax by implicit convention, such as the use of a name or non-rigorous documentation associated with system development.

SIAT proposes an explicit, rigorous model of the correspondence between syntax (machine-world representations) and semantics (real-world phenomena). Additionally, real-world phenomena can be formalized using real-world types, allowing formal analysis of much of the semantic content of interfaces. This analysis provides greater assurance of correct composition.

Real-world types, correspondence models, and their analysis are discussed further in Section 3.4.

3.1.4.6 CLASS: Comprehensive Lifecycle Assurance for System Safety

CLASS, or the Comprehensive Lifecycle for Assured System Safety, is a combined methodology and toolset for developing and maintaining safety critical software systems, illustrated in Figure 9 [15]. CLASS represents an example mechanic for managing the development lifecycle and capturing domain knowledge. CLASS can support SIAT in three ways:

1. CLASS provides tools and processes that ensure that the system and its assurance case are synchronized. This synchronization is important in all phases of the system lifecycle, from development through retirement. CLASS mechanisms can interact with SIAT to support argument and system modularity by interfacing CLASS processes with SIAT argument processes.
2. CLASS provides a repository of artifacts that support system development. SIAT-specific artifacts, such as the success argument patterns, the requirements satisfaction patterns, and the argument contract patterns can be included in CLASS repositories to facilitate applying SIAT to new development efforts.
3. CLASS provides tools and processes that faciliate the capture, structuring, and maintenance of domain knowledge. Domain knowledge is critical to SIAT, as it represents a significant component of context that must be considered for correct argument composition, validation and assessment.

The foundation of CLASS is system safety assurance–arguing why a system is safe. Assurance requires teams to think comprehensively about safety and be able to demonstrate this thinking in a rigorous argument. Argument as a basis for safety assurance is an increasingly common regulatory practice in Europe and the United States.

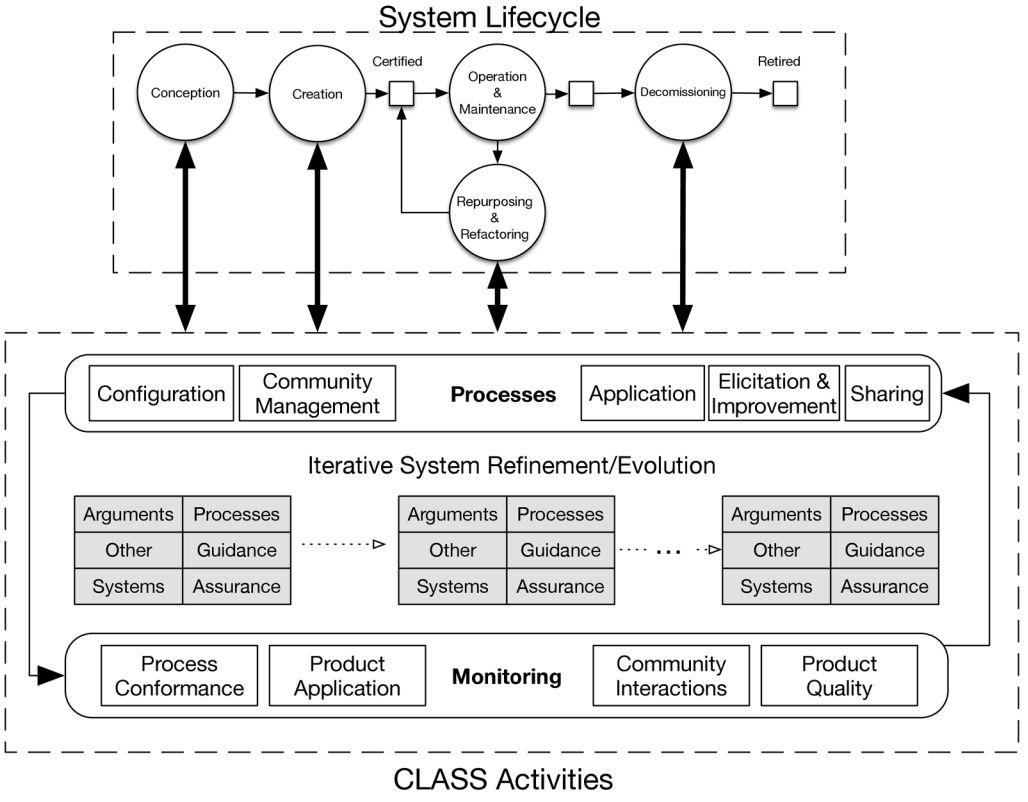


Figure 9: High-Level CLASS Infrastructure

Using CLASS, teams can build, maintain, and retire safety critical software systems with comprehensive and rigorous safety arguments. Unlike other assurance methodologies, CLASS focuses on directly extracting, applying, and testing the assurance rationales of a system's experts. CLASS methodology is based on several principles:

- Domain Arguments: System experts have expert arguments for why their practices and designs are sufficient. These are called *domain arguments* [16]. CLASS encourages writing system safety arguments directly from domain arguments, which in turn encourages living, continuously improving, and representative safety assurance.
- Community of Practice: CLASS replaces the role of the dedicated safety case author with the system's community of experts. CLASS tools focus on team collaboration to build, review, improve, and maintain arguments [15].
- Procedural Support: Assurance of complex systems means having the right people do the right things at the right times. CLASS supports modeling and executing workflow through BPMN2 (Business Process and Modeling Notation) in order to enforce best practices and team norms [15] [17].
- Knowledge Sharing: CLASS provides resource packages modeled on the open source software paradigm. Packages contain guidance documents, argument patterns, and BPMN2 processes representing best practices. Beginners can get started by downloading these packages and applying them to their CLASS-managed systems. Intermediate users can tailor them. Experts can publish their resources as new packages to share with a wider community [18].

- Assured System Modules: In CLASS, a system and its assurance arguments are a resource package. When building a system of systems in CLASS, one downloads component systems as CLASS packages. Therefore, the system of systems receives both the component systems and their assurance artifacts in support of compositional safety. Importantly, CLASS design includes support for notifying dependent systems of changes to component assurance [15].
- Rationale Certification: CLASS promotes the rationalization of the certification process through argument-driven assurance. When standards contain a clear rationale, certification becomes an assurance activity amenable to the CLASS management [19] [20].
- Active Monitoring: CLASS provides tools to actively monitor the continued assurance of a system. This includes monitoring the active run-time state of the system software as well as the activity of executed workflow surrounding the system. This monitoring collects run-time evidence required for assurance, as well as supporting detection of argument assumption violations [17] [21].

Together, these properties support a methodology and toolset in which assurance is integrated into all stages of the system lifecycle. The integration is active, with ownership of the process by the system's experts being the key driver of assurance quality.

CLASS tools are built from open source applications familiar to IT and software development teams [17]. Components include software project management (Maven), resource repositories (Nexus), version control (Git), workflow automation (Camunda), and a programmable wiki environment (XWiki).

3.2 Arguing Successful Development

A fundamental concept of the SIAT theory is the explicit justification that the development effort is successful (see Section 3.1). Successful development of a complex system or system of systems includes successful communication of the rationale for justifiable assurance of success. System interface abstraction technology therefore incorporates the assurance case to document and communicate this rationale.

An assurance case is a reasoned and compelling argument, supported by a body of evidence, that a system, service or organization will operate as intended for a defined application in a defined environment [22]. In SIAT, the argument is used to justify successful system development. System interface abstraction technology documents arguments using the Goal Structuring Notation (GSN) [23]. Detailed background on assurance cases and GSN is provided in Appendix A.

In principle, the organization of arguments supporting successful development is subject to interpretation and therefore may differ depending on the argument engineers and stakeholders. The reference success argument facilitates development of arguments that follow the principals of SIAT. Furthermore, the success argument pattern enables engineers to iteratively refine, alter, and record concepts of successful development so that they can be subjected to further scrutiny.

This section presents reference argument patterns for arguing successful development using GSN. For further background on the notation of arguments and the use and instantiation of argument patterns, readers are referred to Appendix A and the GSN community standard [23].

3.2.1 Practical Argument Patterns: Pattern Flexibility

Argument patterns as suggested in SIAT are intentionally less rigid and prescriptive than is often suggested by more traditional argument patterns. Patterns in SIAT are used more to guide discussion and provide a basis for argument development. SIAT argument patterns should be considered maleable and not strictly prescriptive.

The rationale for this choice is that fine-grained argument structures and phrasing within the arguments can often be adequately expressed in more than one way. Furthermore, each domain will likely require subtle variations and refinement to argument structure and phrasing to meet the expectations of relevant stakeholders and argument reviewers. Defining patterns to express all possible domain arguments (see Section 4.3 and Section 3.1.4) is not practical.

Rather than attempt to provide one definitive argument structure that will likely be the subject of controversy by domain experts, argument patterns presented here take a more practical approach. Specifically, patterns are used to convey a conceptual organization and flow of the argument that can be further refined as necessary.

We therefore take a position that the argument concepts described within the presented patterns are of primary importance, more so than the fine-grained pattern structures, organization, or phrasing itself. Patterns should not be used to force a manner of communicating the argument that is considered unacceptable or undesirable within a given domain. Variations to the patterns as the argument is instantiated are allowed to better meet domain-specific needs, but variations are anticipated to convey the same general principles.

3.2.2 High-Level Argument Structure: The Success Argument

The success argument is the top-level argument structure in SIAT, and is used as the top-level argument for every modular component of development (further discussed in Section 3.3). The success argument argues that the development of the system component was successful by justifying that the application of the reference model was successful: succesful problem definition, context definition, solution definition, and solution assessment (discussed in Section 3.1.2). The highest level of the success argument pattern is illustrated in Figure 10. Branches of the argument are explicated in subsequent sections.

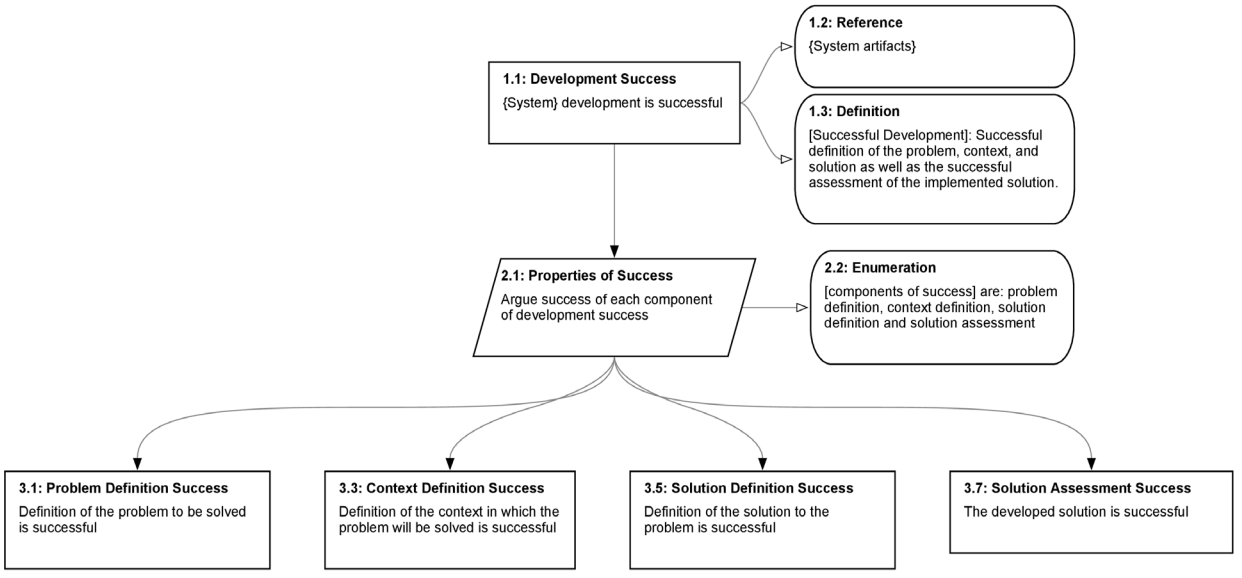


Figure 10: Success Argument Pattern

3.2.2.1 Defining Success, Adequacy, Mitigation, Etc.

The form and content of the success argument pattern is based upon concepts from the reference model (Section 3.1.2) and common practice. The argument is used to demonstrate that the development of the given system is "successful", where the term "success" indicates an assessment of a larger class of concepts within SIAT. "Success" is used instead of any specific metrics because concepts of success are largely stakeholder-defined, domain-specific, and highly detailed and complex, as is defined by the complex argument structure underneath a success claim. The term "success" is used instead of terms like "adequate" since success seems to imply more strongly a threshold by which termination of development is acceptable.

In SIAT, successful development is justified by components of success. These components can be classified in two general categories:

1. successful definition of key SIAT development artifacts, i.e., the problem, context, and solution, i.e., requirements (Goals 3.1, 3.3 and 3.5 in Figure 10), and
2. successful assessment of the implemented system with respect to the defined problem, context, and solution (Goal 3.7 in Figure 10).

If these goals are justified, then under SIAT, the argument justifies successful development.

Other non-specific terms, such as "adequate" and "mitigation" are also used throughout presented arguments and argument patterns in this report. The precise definition of these concepts cannot be made until a system is developed and stakeholders approve the definition. When patterns are instantiated, it is up to the argument engineers and system stakeholders to decide if it is more appropriate to concretize these concepts and reference explicit definitions in GSN context elements, or to rely on the argument structure itself to provide reviewers with the definition as implied by an argument trace.

3.2.2.2 Success Argument Organization

The success argument builds upon problem frames proposed by Jackson [8] and the enhanced reference model proposed by Strunk [9] (see Section 3.1.4). Specifically, the success argument builds upon the problem frame notion where the argument is used to stipulate that the system, as developed within its operating context, *entails* the requirements. The SIAT success argument justifies successful development by demonstrating that:

1. the problem, context and solution (i.e., requirements) are adequately defined and
2. the implementation of the specification in correspondence with its intended context satisfies the requirements of the solution to the problem.

As described in the previous section, the structure of the argument is conceptually divided into successful definition of the problem, context and solution (requirements) and then successful assessment of the implemented system. The argument structures for successful problem, context, and solution definition (to be defined under Goals 3.1, 3.3 and 3.5 in Figure 10) are shown in Figure 11, Figure 12, and Figure 13 respectively. The argument structure for successful solution assessment (to be defined under Goal 3.7 in Figure 10) is shown in Figure 14.

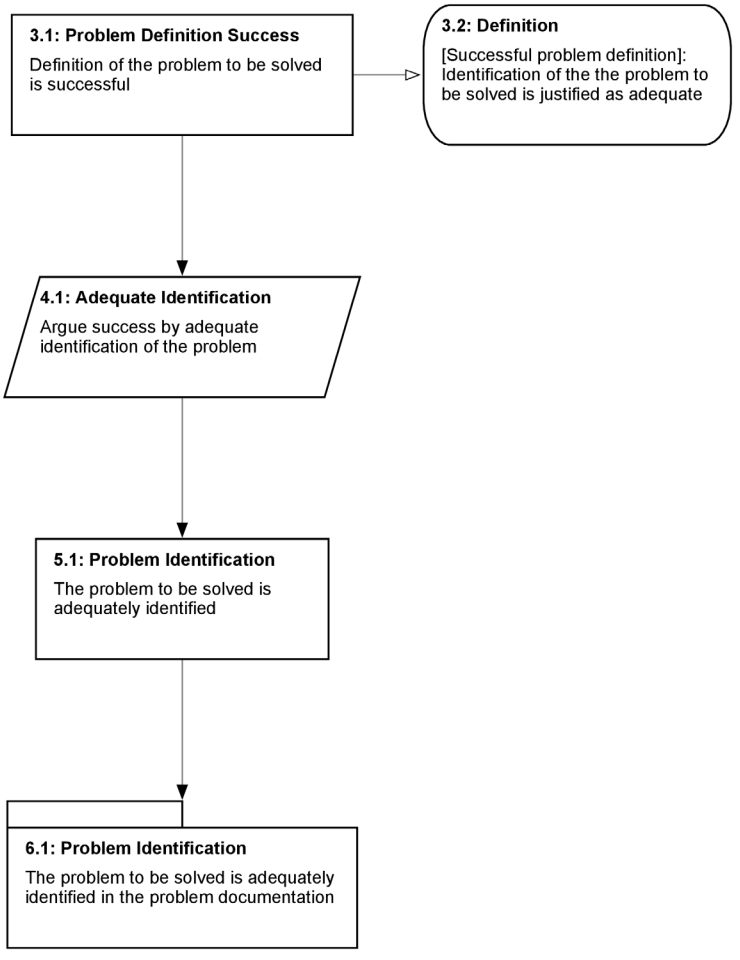


Figure 11: Successful Problem Definition

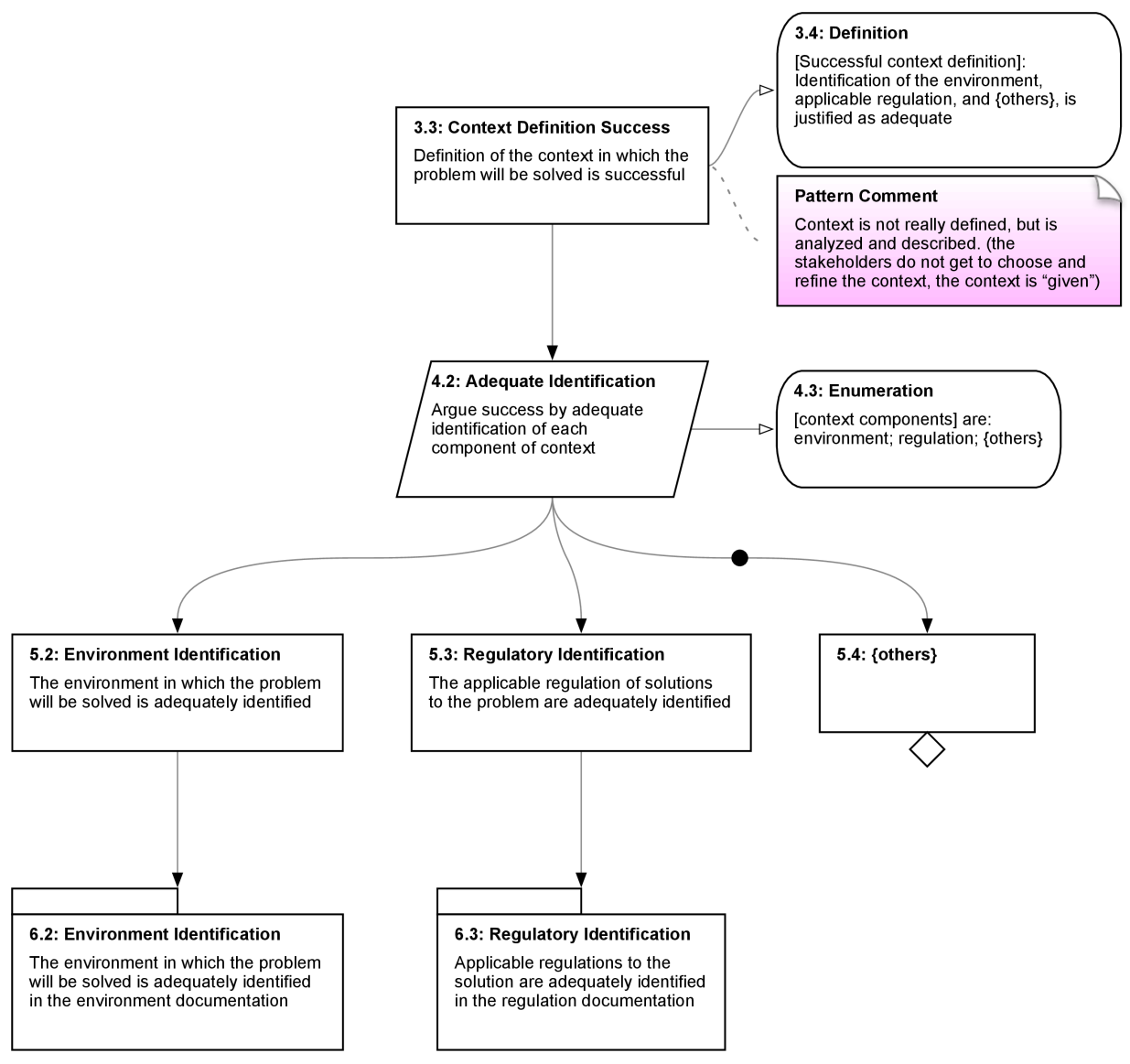


Figure 12: Successful Context Definition

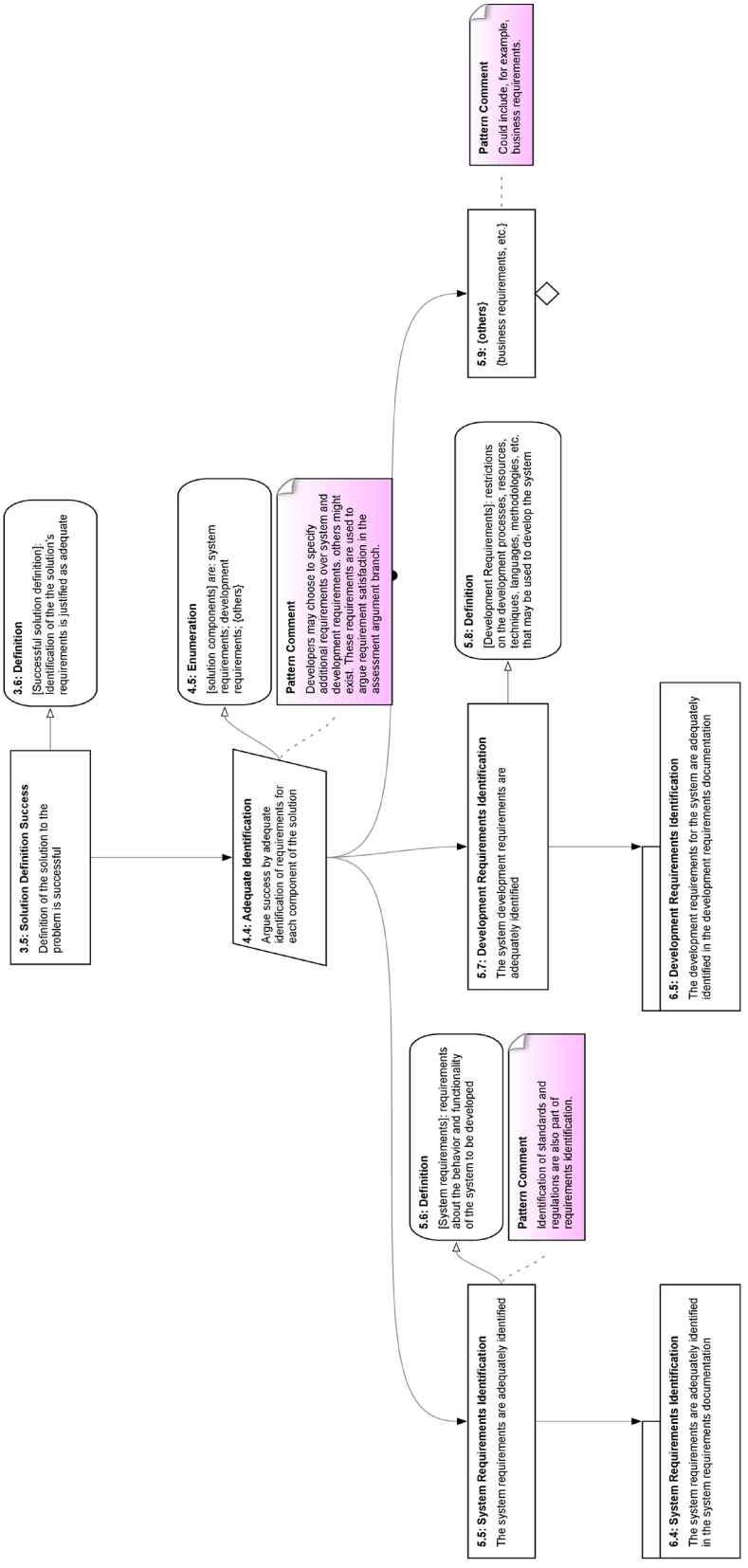


Figure 13: Successful Solution Definition

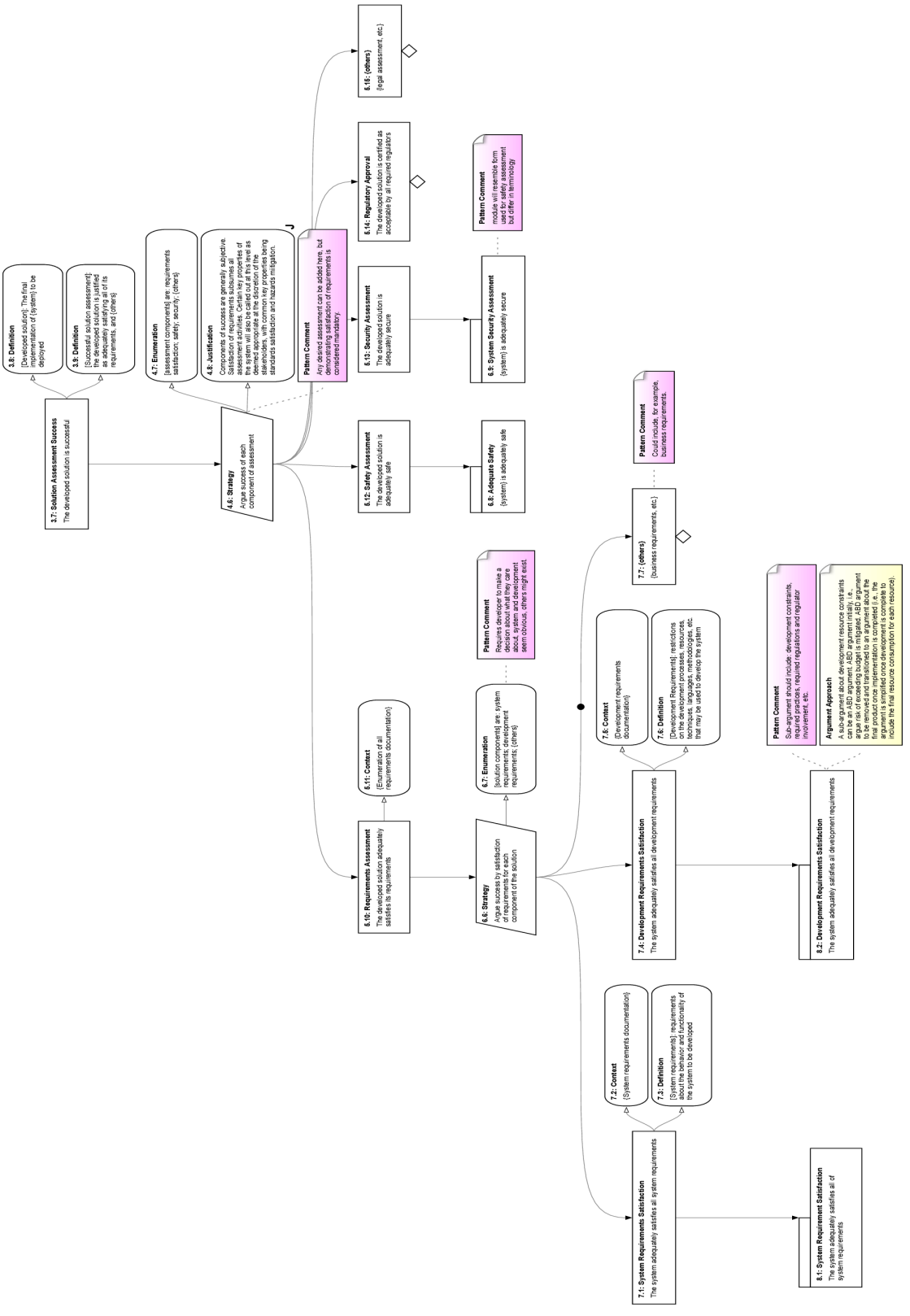


Figure 14: Successful Solution Assessment

The leaves each of the primary branches of the success argument typically terminate with argument modules. These modules are not used to express modular system development, but rather are used to simplify the complexity of of the argument structure. It is possible to remove these modules and continue the argument as a monolithic entity; however, we found this modularity provided a structure that was more accessible for discussion and review.

3.2.2.3 Addressing Development Risk

Successful development may include alternative notions of development that include more than just the system design. For example, arguments may be considered necessary to justify that the development process will be completed within in budget and on schedule. Such arguments have been suggested in prior work on process synthesis using assurance-based development (ABD) [24] [10] [25]. In this prior work, two arguments are developed for a subject system: one argument justifying that the system is fit for purpose, and another argument justifying that the development effort under way will yield an adequate system on time and within budget. The latter argument is continuously developed, managed, and updated during the development process to track that development is inline with development goals/restrictions. The argument of "development process success" is not completed until the development process is complete. Once the development process success argument is completed, it is typically rendered moot (development risk is typically no longer a concern once the system has been completely implemented).

Similarly, a deployed system might have to fit into a larger business vision that may evolve over time. Arguments might be necessary to demonstrate that the system allows the business to meet certain financial expectations and goals. These arguments might be updated much as ABD arguments are updated over a period of time (e.g., the fiscal year) and become moot once the specified time has past, in which case a new argument might need to be developed.

The use of process arguments to track management/development/business concerns, if desired, can be adopted within the success argument structure, and is currently represented in the argument structure. Under Goals 5.7 and 5.9 in Figure 13 it is possible to define alternative requirements other than design requirements. Likewise, under Goals 7.4 and 7.7 in Figure 14, the satisfaction of these requirements are justified. It is envisioned that the development and use of the satisfaction arguments for these requirements would be consistent with the processes and uses as described under ABD. Within the given effort, further investigation into the use of ABD is out of scope.

3.2.3 Problem, Context, and Solution Definition

Arguments supporting successful definition of the problem, context and solution (requirements) are based largely on the expectations of the system stakeholders. In some cases, a simple review and approval by experts or compliance with a given regulation might be considered sufficient. In other situations, adequate identification is defined by a careful examination of chosen elicitation processes and artifacts produced from elicitation processes.

The concerns of problem, context and requirements definition can be generalized into an "identification/elicitation" concern, where the specific entity being identified does not matter. The leaves of all three successful definition branches (Figure 11, Figure 12, and Figure 13) reference a separate argument module where these identification concerns are addressed. We observe that to have "adequate identification" generally, the argument must at least justify

complete, appropriate and correct (accurate) identification of entity being identified. We further generalize these concerns as confidence characteristics that affect the legitimacy of inferences and artifacts throughout the argument. An identification argument is therefore a specific type of confidence argument in SIAT; however, because confidence in these entities is critical under the SIAT reference model, these confidence arguments are not separated from the main argument structure as would typically be the case for confidence arguments (see Appendix A). Other more traditional uses of confidence arguments throughout the argument would be separated to simplify the argument structure. The structure of separate confidence arguments would also consider completeness, appropriateness and accuracy. Other confidence concerns are added as is deemed appropriate/necessary. The top-level identification argument is shown in Figure 15.

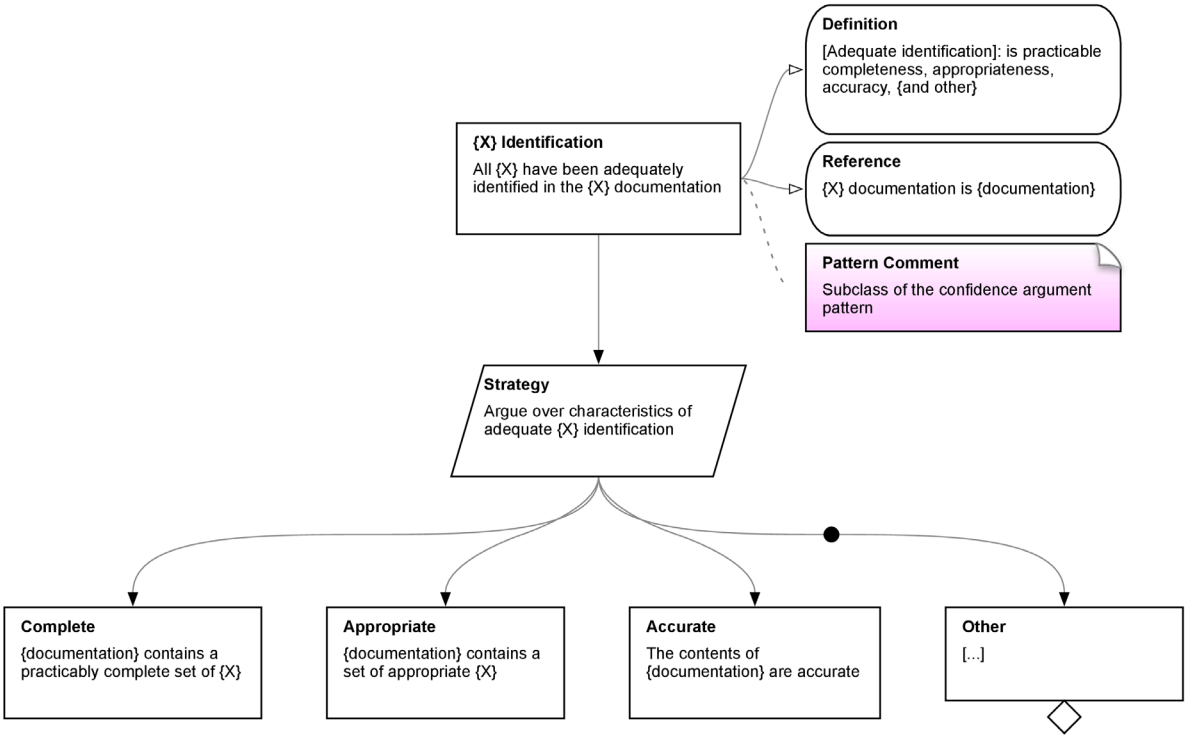


Figure 15: General Entity Identification Pattern

From these basic identification properties (correctness, appropriateness, and accuracy), we informally elicited a more detailed subset of properties that could be used to justify each of the basic identification properties. Elicitation of these more detailed properties was not intended to provide a definitive or well-accepted set, but rather the set was derived to provide a discussion of the kinds of arguments necessary to justify adequate identification. The set was supplemented with additional properties as the pattern was applied for specific purposes, e.g., requirements elicitation literature such as IEEE 830–1998 [26] and IEEE 29148–2011 [27] suggested additional properties.

Generally, correctness, appropriateness, and accuracy can be justified directly by assessing the produced artifact (e.g., the requirements document) or indirectly by assessing the processes used

for identification/elicitation. Direct assessment is typically preferred but often either direct assessment is not well established or residual doubts remain that can only be addressed by examining the processes used to develop the artifact.

Properties to support correct, appropriate and accurate identification could be argued in GSN; however, the pattern must document potentially complex structure to account for domain-specific variations in identification confidence. While we did develop some GSN patterns initially for this purpose, the utility of the argument structure was questionable. For simplicity, we instead list potential properties for correct, appropriate and accurate identification below and leave definition of the associated argument structure to domain experts. Characteristics such as these and relationships between these characteristics can be used to define a common characteristic map [28] as a more general mechanism for describing confidence patterns in terms of key properties without necessarily prescribing a GSN argument structure.

Potential properties of complete identification:

- complete identification of accepted sub-classes of the identification problem (for example classes of requirements, such as functional and non-functional)
- use of well-established elicitation processes (i.e., prior vetting of elicitation processes)
- correct application of elicitation processes (e.g., using prescribed methods with trained elicitors)
- use of reliable documentation and storage procedures (e.g., justification that documentation procedures do not accidentally omit or delete contents)
- assessment and approval by stakeholders of elicitation activities and/or the produced artifact
- assessment of completeness with respect the needs of other artifacts or activities (e.g., the context - environment and regulation - may be considered complete or partially compelete if all requirements and the problem description only refer to contents described within the context description).

Potential properties of appropriate identification:

- consistent artifact contents with respect to other internal contents and other relevant artifacts (e.g., the requirements are consistent with the problem description)
- unambiguous artifact contents
- lack of redundancy in artifact contents
- credible artifact contents
- realistic artifact contents
- necessary and/or relevant artifact contents
- atomic artifact contents, i.e., of simplest expected form (e.g., atomic requirements)
- verifiable artifact contents
- traceability of artifact contents to other relevant artifacts
- comprehensible artifact contents
- prioritized artifact contents
- well-structured artifact contents (e.g., categorized or organized in a standardized form)
- correct and up-to-date artifact

Potential properties of accurate identification:

- correct application of elicitation processes (e.g., using prescribed methods with trained elicitors)

- use of reliable and/or well-established elicitation processes (i.e., prior vetting of elicitation processes)
- verification/validation of the artifact's contents (i.e., the form and semantics of the contents are correct)
- use of reliable documentation and storage procedures (e.g., documentation is not unintentionally or maliciously modified to reflect incorrect information)

3.2.4 Solution Assessment

In principle, if the system requirements are completely identified, then justifying that all requirements are satisfied should provide a sufficient assessment of the implemented system. This notion is more aligned with the original Jacksonian problem frames argument. In practice, however, reviewers, such as certifiers, often prefer different "views" of system assessment. For example a dedicated safety assessment (justification of hazard mitigation), regulation assessment (justification of compliance to regulations/standards), and security assessment (justification of mitigation of threats to system assets). The successful solution assessment branch (Figure 14) provides several example assessments that could be further instantiated as desired by system reviewers and certifiers. At a minimum, this branch should include a requirement satisfaction argument.

This section discusses an argument pattern for requirement satisfaction, safety assessment, security assessment and regulatory compliance.

3.2.4.1 Requirements Satisfaction

As commented on above, the success pattern supports the identification and satisfaction of various types of requirements; however, for simplicity of this effort, we focus on typical system requirements, to be justified as adequately satisfied within Module 8.1 of Figure 14.

The requirement satisfaction argument approach is illustrated in Figure 16. Requirements are assumed to be adequately identified by the solution definition argument branch (Figure 13) and the focus of requirement satisfaction is placed on demonstrating that the provided requirements are satisfied. Requirements are satisfied by a recursive refinement of each requirement into sub-requirements, specifications, high-level architecture, lower-level design specs, and ultimately by evidence about the implemented system itself.

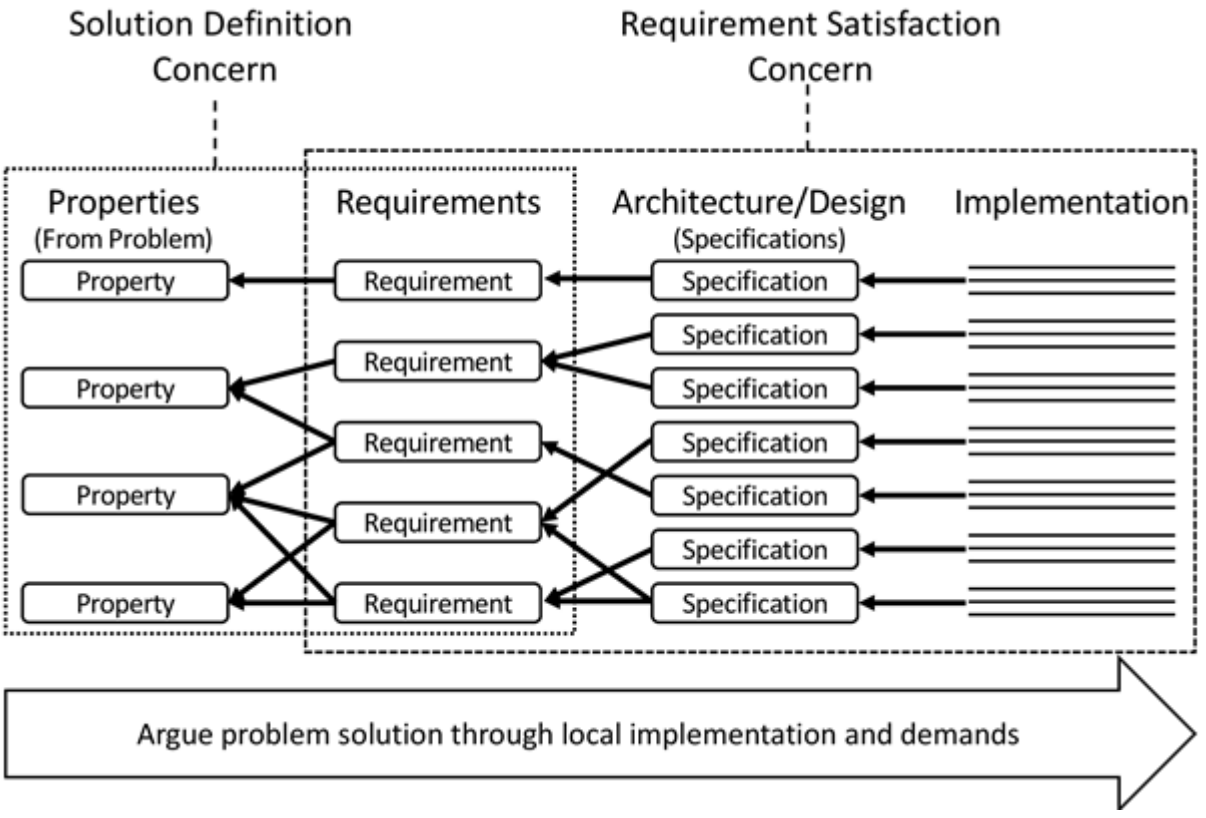


Figure 16: Requirements Satisfaction Overview

The requirements satisfaction pattern (Figure 17) enumerates each requirement, and justifies that the detailed spec/design for each requirement:

1. actually entails the requirement and
2. is satisfied by the implemented system.

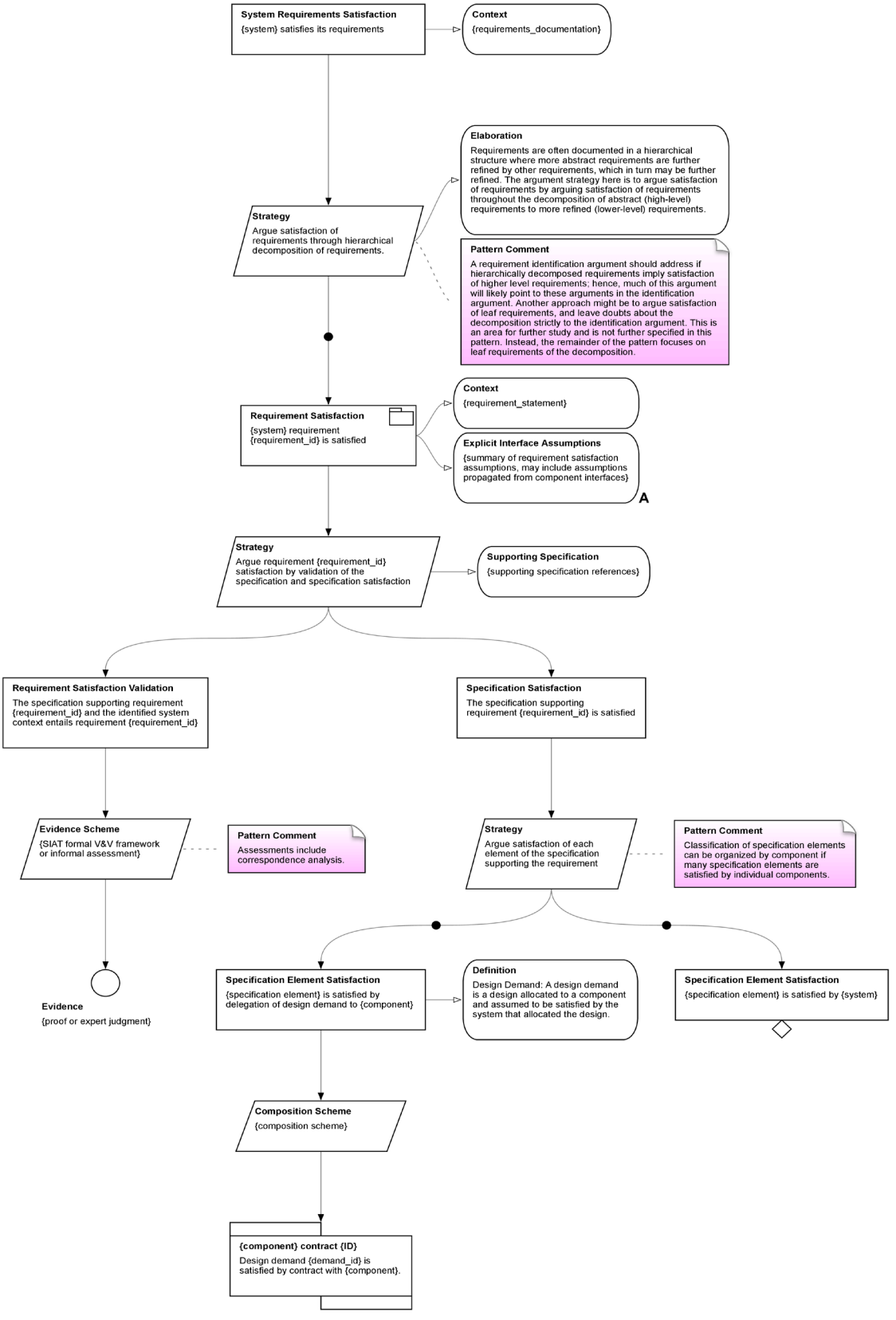


Figure 17: Requirements Satisfaction Pattern

In arguing the specification entails the requirements, the assessment may need to consider retrenchment and correspondence with machine-world types to real-world types. The assessment is likely to be based on some combination of expert judgement and formal proof. The argument for requirement satisfaction validation could be decomposed into sub-properties of validation (retrenchment, expert approval, etc.). The presented argument is simplified to assume one item of evidence that encapsulates a complete assessment of entailment (requirement satisfaction validation assessment).

Satisfaction of each element of the specification is achieved by further decomposition into more detailed design elements or by evidence about the system itself. The pattern also illustrates that it is possible to support satisfaction of the specification by use of a modular component, i.e., another system component to be implemented elsewhere or to be specifically developed in order to support modularity. In this instance, further detail of the design and implementation of the component is not available at this level of development. A *demand* is specified but the details of how the demand is satisfied are explicated by a contractual argument with another component and a separate development process for the component. Modular development is further discussed in Section 3.3.

Both requirements and the detailed specification/design of the system can be developed hierarchically. The argument could similarly capture the hierarchical decomposition or the hierarchy could be flattened as appropriate or desired. The presented requirement satisfaction pattern illustrates a flattened hierarchy approach. If the pattern were extended to a hierarchical decomposition approach, each level of refinement in the requirements and specification would justify satisfaction of a higher-level requirement or specification. There is therefore some doubt that the refinement is itself adequate (complete, correct, appropriate, etc.). For requirements, these doubts should be addressed in the solution definition argument. For the specification and further system design, however, these concerns must be explicitly addressed either once at the beginning of the hierarchical decomposition (as is illustrated in the above pattern) or at each level of refinement.

The requirements satisfaction pattern is presented in GSN, but given the repetitive nature of the argument, the argument could also be expressed in a tabular structure. We leave the choice to simplify argument structures into tabular forms to the discretion of those applying these patterns.

3.2.4.2 Safety Assessment

The top-level safety assessment pattern used within Module 6.8 of Figure 14 is illustrated in Figure 18. This pattern admits the possibility of multiple safety assessments, but focuses primarily on a hazard mitigation assessment.

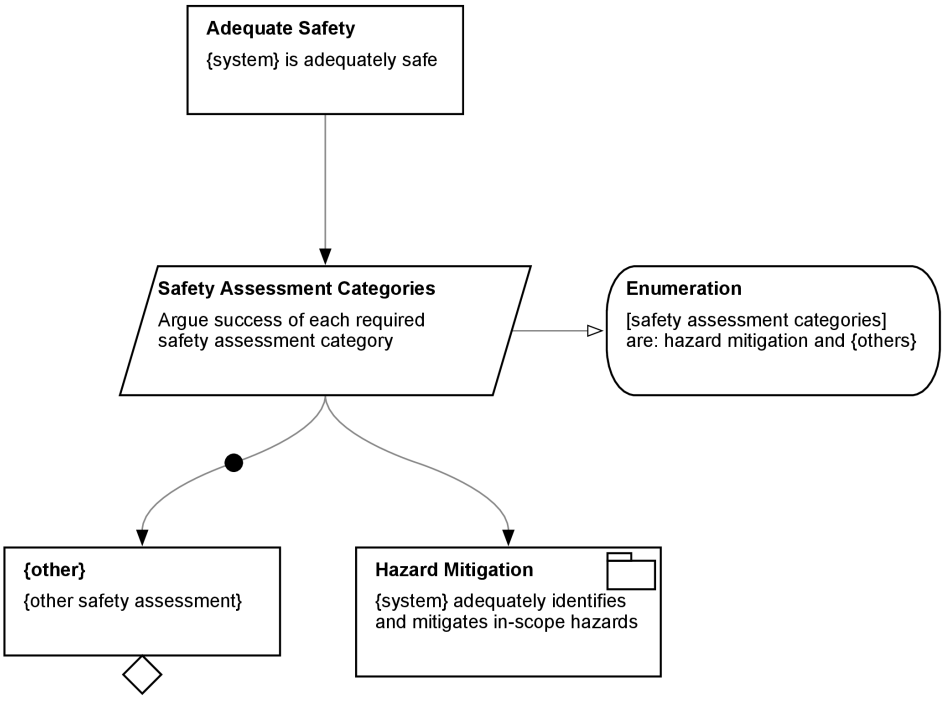


Figure 18: Safety Assessment Pattern: Top-Level Structure

The hazard mitigation argument is an adaptation/extension of Hawkins tiered arguments [29] [30], where each "tier" within SIAT refers to composable modules within a system of systems (Section 3.3). At each tier, the hazard mitigation argument justifies "in scope" hazards are adequately mitigated. In scope hazards in SIAT consists of three categories of hazards:

- Current Tier Hazards: Hazards that are applicable only to the given system (the hazard is not part of the set of hazards that are part of the higher-level system).
- Lower Tier Hazards: In-scope hazards of composed components (at the next level down in the composition of components).
- Induced Hazards: Hazards applicable to higher-level systems that can be induced by failure modes of the given system.

To argue these hazards are mitigated, these hazards must be justified as adequately identified. The argument structure therefore decomposes into identification and mitigation, shown in Figure 19. Arguing adequate identification can in principle be achieved by the same argument structure used for problem, context, and solution definition over each category of in scope hazards. One primary difference is that adequate identification may rely upon the identification of hazards by modular sub-components, if sub-components are used. Specifically, the identification of lower tier hazards is justified only by identification arguments found in composed components. In such instances, there is therefore a delegated hazard identification demand for lower tier hazards, illustrated in Figure 20.

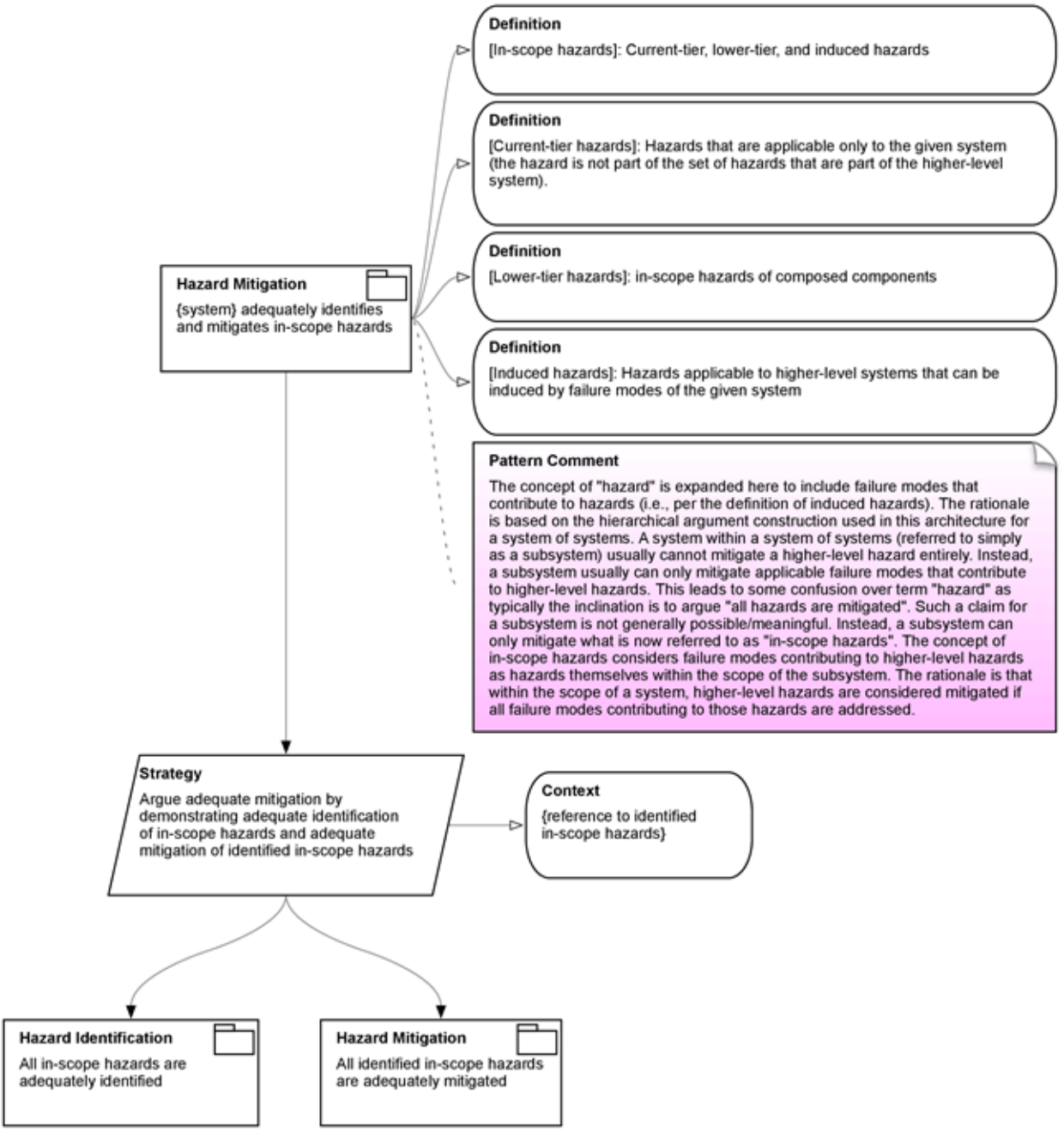


Figure 19: Safety Assessment Pattern: Hazard ID and Mitigation

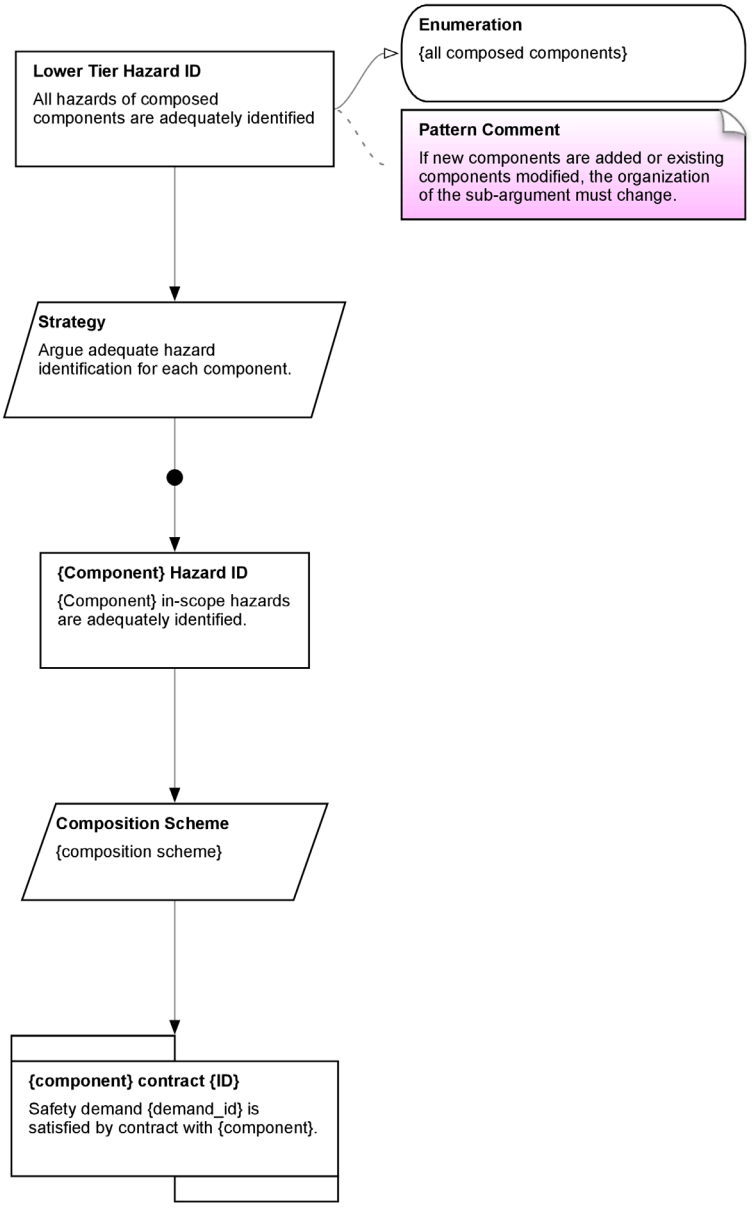


Figure 20: Lower Tier Hazard Identification Delegation

With exception to lower tier hazards, all hazards are justified as adequately mitigated by some combination of the following (illustrated in Figure 21) :

1. evidence in direct support of mitigation,
2. further argument decomposition, or
3. delegation of hazard mitigation to another component (i.e., the given system mitigates the hazard by relying on mitigators in a separate sub-component).

Lower tier hazards are mitigated entirely by composed components themselves, if sub-components are used in the design; hence, each modular sub-component is delegated the responsibility of mitigating its own in scope hazards.

As with requirements satisfaction, hazard mitigation could also be justified using a tabular notation if desired.

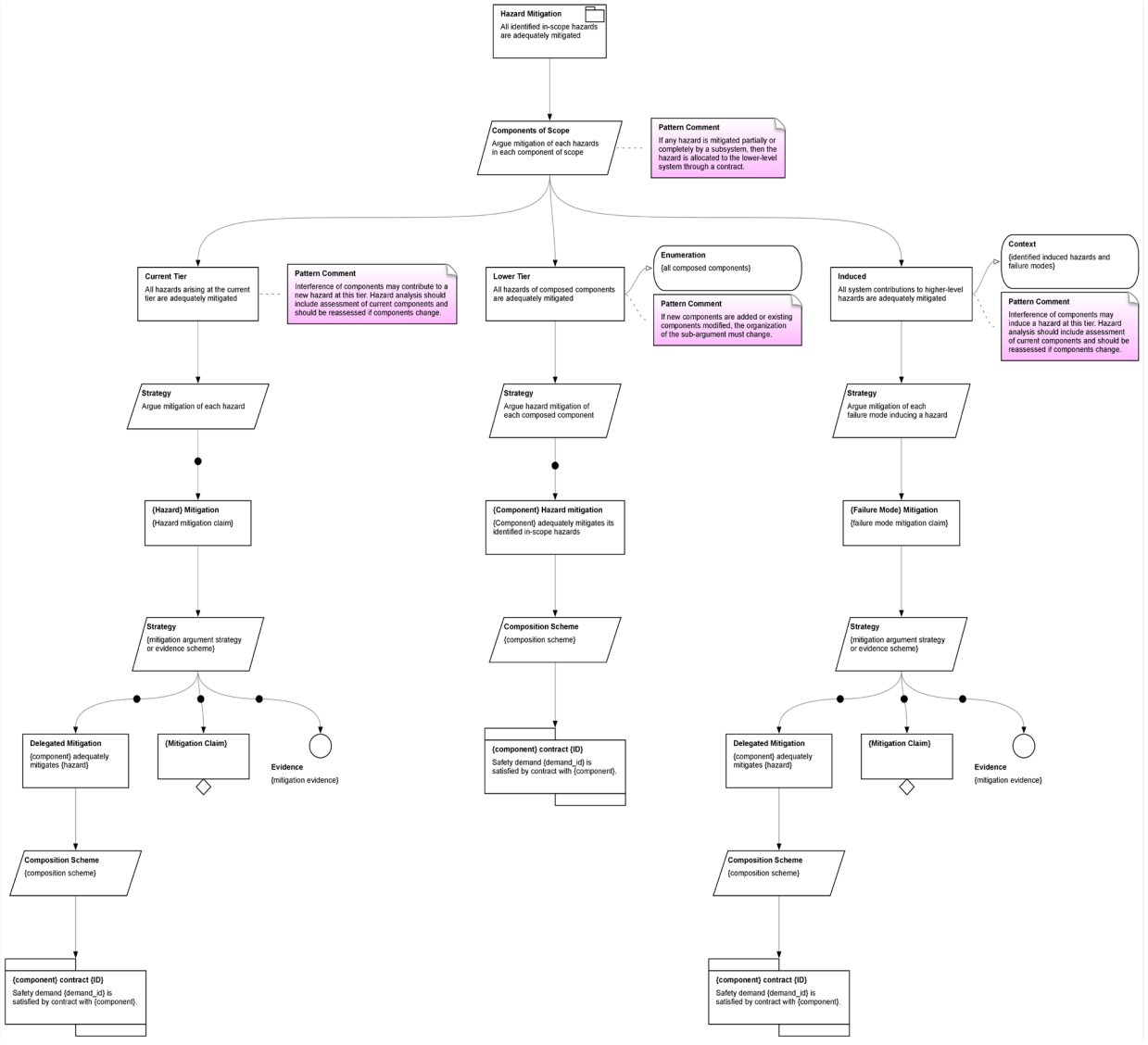


Figure 21: Hazard Mitigation Pattern

3.2.4.3 Security Assessment

The majority of this effort focused on requirement satisfaction and partially on safety assessment. We note, however, that a security assessment argument (Module 6.9 in Figure 14) would in principle be similar to that of the safety assessment argument described above. The key differences would be in the terminology. In security, threats are mitigated, not hazards. Mitigation of threats is design to protect assets from malicious third parties. For further discussion of the use and application of security arguments and patterns, readers are referred to prior work by Rodes [31] [32] [33] [34].

3.2.4.4 Regulatory Compliance

Regulator compliance/approval (Goal 5.14 in Figure 14) is not specified with a supporting module due to the potential simplistic nature of compliance. Specifically, once regulations have been identified, demonstration of the compliance could involve a check list of regulations with associated evidence. This could in principle be argued within a separate module using the pattern shown in Figure 22; however, unless there is a more complex justification of compliance, the use of an explicit argument is likely neither necessary nor desirable. GSN argument is best served when the rationale for compliance is not as apparent as a direct mapping of evidence to regulations. If compliance with a regulation, for example, is justified by a reference to a complex hazard mitigation claim, an argument may be useful to point to relevant mitigation claims (using GSN away goals). The exact argument structure to capture a more nuanced compliance argument is not expressible within a pattern as the argument will vary drastically on a case-by-case basis.

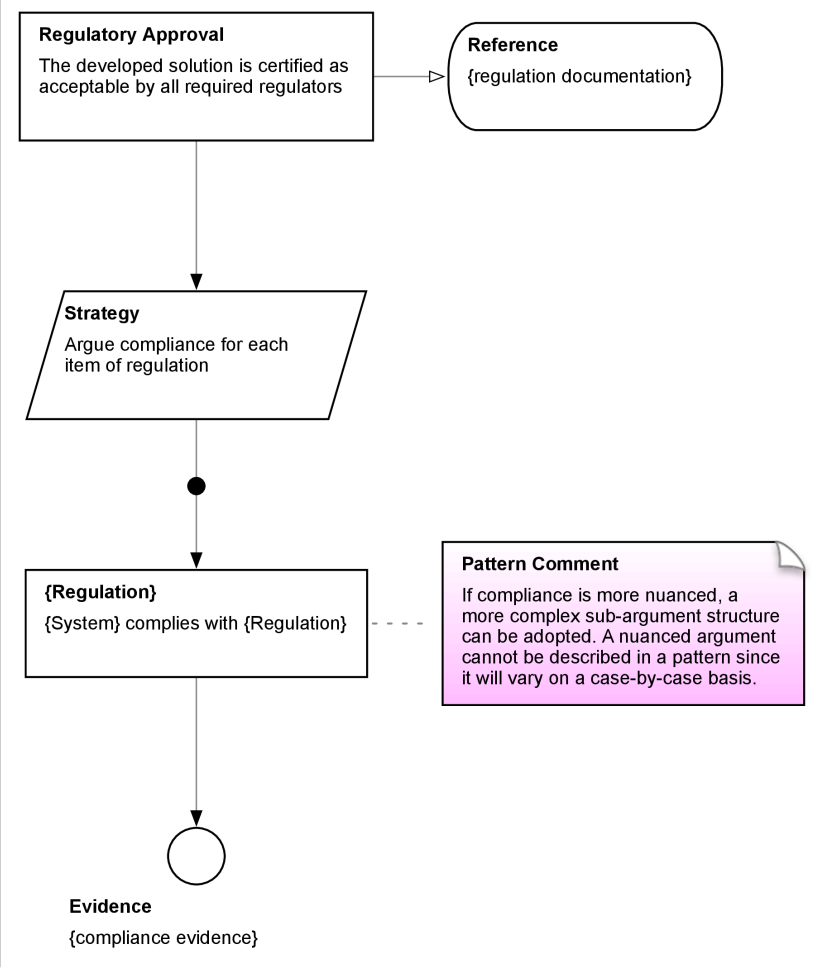


Figure 22: Regulatory Compliance Pattern

3.3 Practical Argument Modularity

Assurance-case arguments naturally reflect the complexity of the systems or systems of systems for which they provide rationale of justifiable assurance of success. As system complexity increases, there is a concomitant increase in argument complexity. For complex systems and systems of systems, an assurance case based upon a single, monolithic argument is difficult to produce, difficult to review, and difficult to maintain.

GSN addresses this challenge through the provision of standardized notation for modular arguments that has been widely applied [23] (see Appendix A). Modular arguments are built upon modules with the following characteristics [35] [36]:

- **High cohesion**: the module supports a well-focused and logically cohesive assurance goal.
- **Low coupling**: the module has minimal interconnection with other modules.
- **Well-defined interfaces**: the module has explicitly defined "allowed collaboration" with other modules.
- **Information hiding**: the number of defined interfaces should be minimized to expose minimal information.

In common modular arguments, argument modules encapsulate and organize logical structures. For example, an argument module might be developed to describe requirements identification, requirement satisfaction, hazard mitigation, or confidence. This style of argument modularity provides useful organization, but insufficiently addresses the fundamental concerns identified above: arguments relying on these kinds of modules remain difficult to produce, difficult to review, and difficult to maintain when engineering arguments about complex system of systems.

The complexity of the systems and systems of systems developed using system-interface abstraction technology requires a *practical* argument modularity. The complex systems and, in particular, systems of systems that are developed using system-interface abstraction technology are not built monolithically. Instead, these systems are naturally built from integrated system components, where each system component encapsulates a solution to an identified problem. Moreover, the problem that a system component solves is considered likely to repeat or is considered of sufficient scope that modularizing the problem facilitates ease in managing the development and maintenance complexity and costs. The selection and integration of system components during system development and, later, the change, replacement or removal of system components post-deployment provides a compelling example of practical modularity for system design.

A similar practical argument modularity is possible. If argument modules align with system components, we can more easily develop and maintain the assurance arguments for systems that are built compositionally. Argument modules are developed that encapsulate the argument for successful development of each system component. As system components are integrated into the larger design of the system, their corresponding argument modules are integrated into the larger argument for the system. The assurance case architecture, therefore, mirrors the problem-oriented design of the system or system of systems itself (see Figure 23).

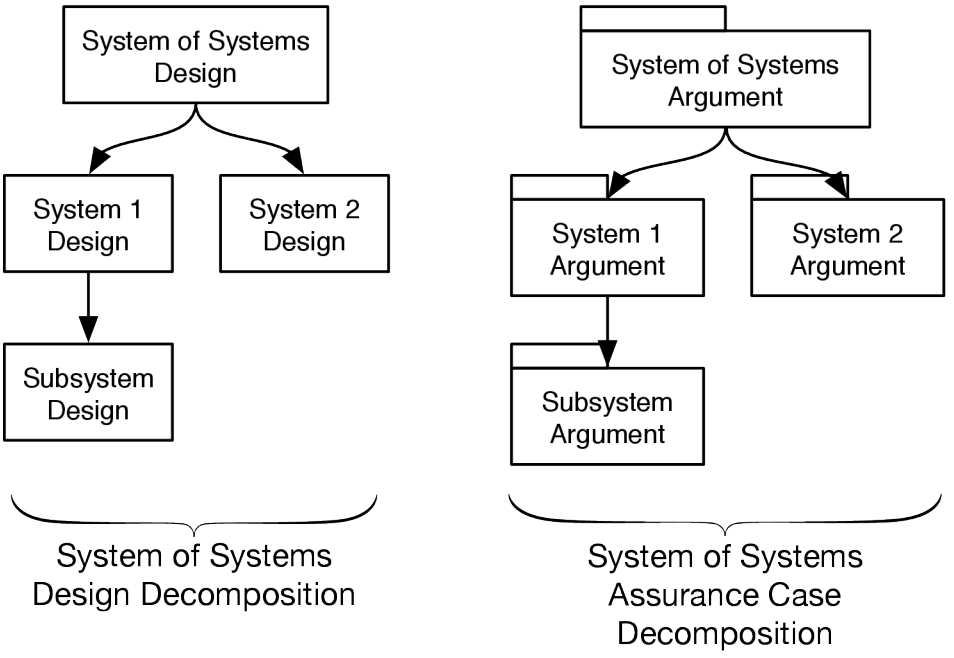


Figure 23: Argument Design Tracking

Following this practical modularity for arguments, the composition of interest is *not* with respect to modularizing and integrating argument structures that encapsulate logical concepts such as requirement satisfaction, hazard mitigation, or problem identification. Instead, the composition of interest is with respect to modularizing and integrating arguments for successful development of each system component (Section 3.2). By matching argument modularity with system component modularity, the argument naturally fits into the development and maintenance of a system or system of systems. As a result, the argument is organized to facilitate practical development and maintenance and ease of review.

The key motivating question in development, review and maintenance of a practical modular argument is:

Are design demands satisfied by integrated system component behaviors?

This question is natural and appears easy to answer:

System components are selected and integrated into the system design because the behaviors they provide satisfy specified demands.

Careful consideration of how the rationale for justifiable assurance of successful development should be established reveals that this question cannot be asked in isolation. In particular, the behavior of a system component cannot be assumed if its argument is not of sufficient quality or if the component is used outside of the context for which it was designed, so consideration of context, both in design and use, is critical. Additionally, there is potential for new hazards that arise from the integration of components, so consideration of hazards is also critical.

In all, there are three fundamental questions that must be asked of practical modular arguments:

1. Are design demands satisfied by integrated system component behaviors?

2. Are all system component contexts (both in development and use) compatible?
3. Are there new, unaddressed hazards arising from system component integration?

We answer these questions by advancing a novel architecture for practical argument modularity that is based on a collection of specific argument views. Additionally, we support the architecture with component integration mechanics that describe the crucial detail necessary to effectively and successfully integrate argument modules associated with system components.

3.3.1 Integration Concepts

Practical argument modularity using system-interface abstraction technology relies on a set of related concepts that work together to answer the fundamental questions identified above. These concepts, discussed in detail below, are:

1. Argument views, which provide special-purpose projections of the argument to enable reasoning about integration challenges;
2. Assume-guarantee reasoning for modular arguments; and
3. Contextual compatibility.

3.3.1.1 Argument Views

Argument views are special-purpose projections of the assurance argument. Each argument view provides clarity to reviewers and maintainers of the argument by encapsulating related argument concepts that may not have been closely grouped in the original argument organization. This use of views to highlight and encapsulate related concepts is similar to the grouping of related software aspects in aspect-oriented programming [37]. Views can also be used to organize different levels of design abstraction, including views that encapsulate other views.

Argument views are abstract concepts that do not have direct support in GSN. We denote views using typical GSN notation for modular arguments, according to the purpose of the view. The views used in the SIAT system-of-systems argument architecture are further discussed in Section 3.3.2.

3.3.1.2 Assume-Guarantee Reasoning for Modular Arguments

Assume-guarantee reasoning (Section 3.1.4) provides the foundation for reasoning about the composition of modular arguments. Arguments are composed by mutual satisfaction of required and provided interfaces. A demand within a required interfaces and a guarantee within a provided interface are expressed within the argument as assurance goals (Figure 24). A demand goal is necessarily a leaf goal within a component module (i.e., no further argument is developed underneath this goal within the component's encapsulated argument structure). Demand goals are not otherwise explicitly documented within the argument structure. A guarantee goal can exist anywhere within the providing component's argument structure and are traditionally documented using GSN public goals. We observe, however, that often prescribed portions of the argument are understood to be public by convention. SIAT, therefore, does not specify a particular use of public goals.

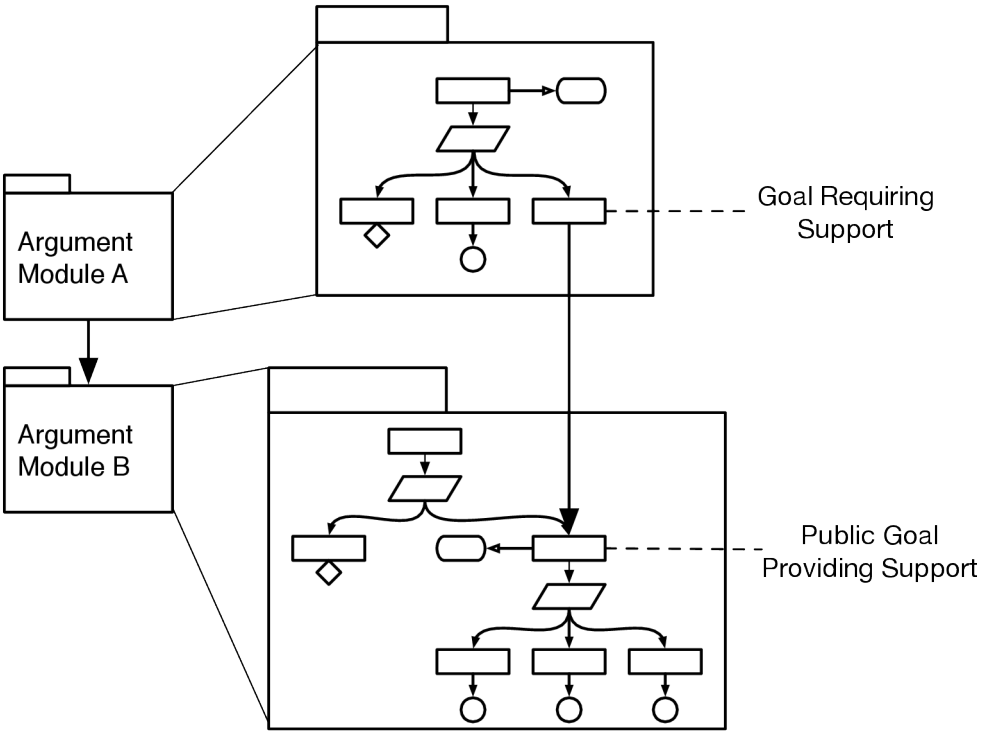


Figure 24: Argument Module Composition

The context associated with a demand goal is not explicitly specified at the required interface within SIAT. The rationale is that integrated components may express arbitrary assumptions, making it difficult to fully specify a required interface context. Instead, context is generated as needed to assess the assumptions specified within a provided interface. GSN context elements are propagated down to required interfaces when interface compatibility is assessed, illustrated in Figure 25.

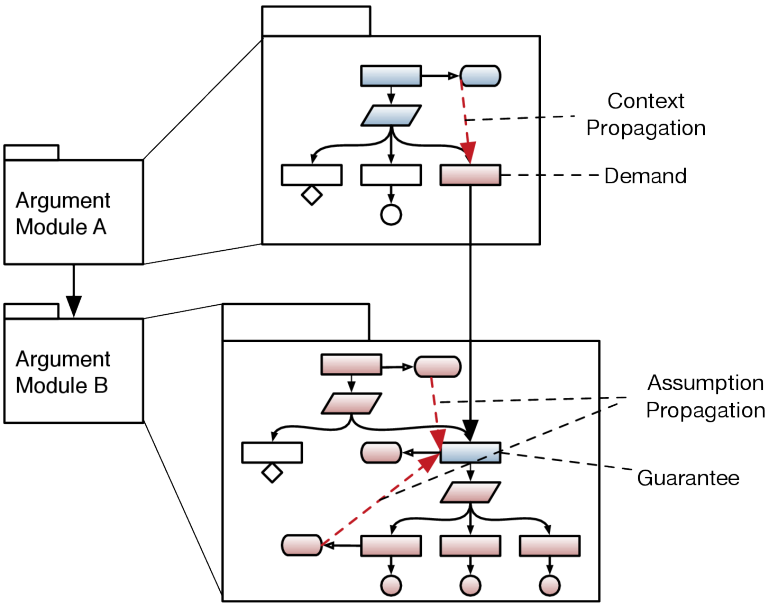


Figure 25: Context/Assumption Propagation

Assumptions for provided interfaces are specified within GSN assumption elements. Assumptions at a provided interface should express all inherited assumptions and any assumptions nested below the guarantee goal, illustrated in Figure 25. Assumptions can be explicitly referenced at the interface assurance goal, or, similarly to context propagation for required interfaces, assumptions can be propagated dynamically to interface goals when performing an interface compatibility assessment. Dynamically propagating assumptions requires an agreed upon argument structure between a consumer and producer (discussed in Section 3.3.8). To allow for flexibility in applying SIAT, we do not impose a particular method for expressing assumptions at provided interfaces.

3.3.1.3 Contextual Compatibility

The development of assurance cases from component arguments is based on a fundamental principle (or "*fundamental theorem*" if we shall permit a loose notion of theorem) of compositional assurance:

A component that is acceptable in one system is acceptable in another so long as both systems have **identical contexts**.

When we say that the component is acceptable, we include its functional and non-functional behavior, its ability to satisfy system demands, and its ability to satisfy stakeholders, including regulatory authorities. Since the ability of a component to satisfy system demands is addressed through assume-guarantee reasoning, we restrict our consideration of context to other concerns, including restrictions, constraints, characteristics, phenomena, acceptability criteria, operating procedures, domain, design, configuration, and dimensions.

The scope and complexity of context is sufficiently broad as to suggest that practically all systems, regardless of how similar they appear, will have different contexts. Practical argument modularity thus requires a practical approach to contextual compatibility: we must be able to say

that context is *practically identical*. An argument must be made that differences between contexts are either inconsequential or are adequately mitigated.

A further complication suggested by the scope and complexity of context is that components, as they are integrated, may expand or restrict the larger system context in subtle and intricate ways. Consequently, fully encapsulating change through a modular argument architecture is not always possible. The impact of composing arguments might propagate beyond component boundaries, and what's more, the propagation might not be linear or hierarchical, thus violating key principles of modular design, i.e., there is an implied violation to information hiding, low coupling, high cohesion and/or well-defined interfaces. Instead of trying to avoid all possible violations to encapsulation when establishing contextual compatibility, we posit the following:

A practical argument modularity must endeavor to maintain encapsulation as much as is possible but recognize when and how encapsulation violation should occur.

Addressing the limitations of both modular encapsulation and assume-guarantee in terms of contextual compatibility motivates the use of argument views and drives many of the argument integration mechanics, both are further described below.

3.3.2 Architecture

The SIAT modular argument architecture is concerned with the interaction between system components and their corresponding arguments, and not the internal structure of individual component arguments. Internal component argument architecture is discussed in Section 3.2.

The SIAT practical argument modularity relies on four key types of argument views:

1. component module views,
2. component contract views,
3. sibling contract views, and
4. system-wide dependency views.

Together, these views (further described below) provide a framework for integrating component arguments and organize a high-level system-of-systems assurance case architecture. Although the views presented below were developed progressively to address integration challenges as described in Section 3.3.1, they share the motivation and argument forms similar to the modular argument structures presented by the Modular Software Safety Case (MSSC) project developed by the Industrial Avionics Working Group (IAWG) [38] [36]. This related work provides some inspiration for our work, and some validation of the concepts we had independently derived, although the principle organization of the SIAT argument architecture differs as it is based on a hierarchical problem-oriented approach to modularity and component integration.

3.3.2.1 Component Module Views

Component module views encapsulate the argument structure associated with individual system components. The boundary of the argument encapsulated in the component module view is aligned with the boundary of the system component described. Each system component encapsulates a solution to a problem. Likewise, each component module view encapsulates the related argument for successful development of the system component (see Section 3.2). The concept represented by a component module view is similar to the notion of a "block" as proposed in related work [38].

Aligning the component module view to the system component boundary ideally provides two key benefits:

1. **Information Hiding**: Changes made to the system component, which necessitate a revision to the argument for the component, will propagate into the broader argument through well-defined interfaces.
2. **Practical Argument Modularity**: Integration of argument modules based on component module views affords practical modularity (i.e., the argument modularity maps to the modularity of the system).

As previously discussed above, perfect information hiding is impossible to achieve in practice. We address limitations in information hiding through arguments associated with additional views (presented below).

We represent a component module view graphically as shown in Figure 26: a component module is conceptualized as a GSN module element encapsulating a successful development argument.

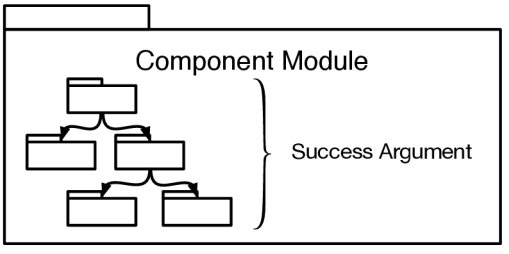


Figure 26: Encapsulated Success Argument

The hierarchical relationship between component modules presents its own "view" in terms of the system design/architecture. The hierarchical relationship between component modules describes a "*design authority*" architecture (an example design authority construction is shown in Figure 27).

When a component delegates responsibilities to other components, there is an implied authority assumed to make design decisions. Design authority is therefore the organization of responsibility for specifying a design.

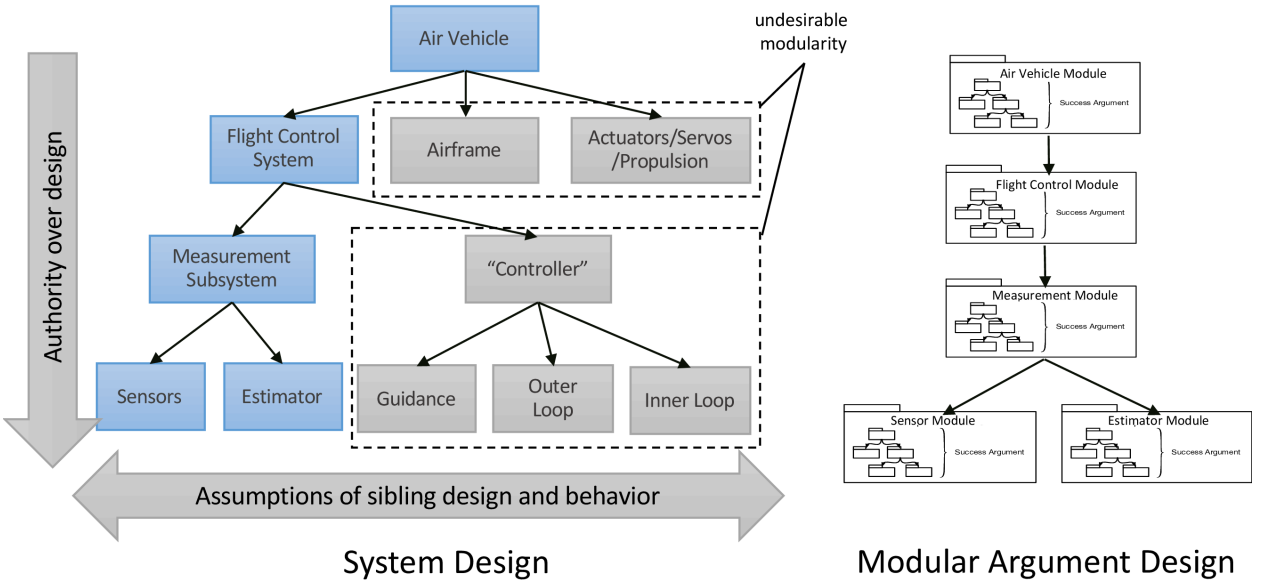


Figure 27: Design Authority — Example

The notion of design authority naturally supports argumentation. As requirements are decomposed into a specification and detailed design, an intuitive argument is developed that justifies the decomposition will satisfy elements higher in the decomposition. When demands are delegated onto components, there is therefore an argument that the components will satisfy these demands. The argument naturally continues hierarchically down into component module arguments to justify demand satisfaction.

In this manner, the composition of component modules reflects a design authority view; however, as illustrated in Figure 27, not all components with a system design authority architecture need be selected for modularization. Stakeholders may consider modularization unnecessary or unwarranted in some instances. These components are considered part of the local implemented of the parent. For example, in Figure 27, the airframe was not chosen for modularity. The airframe then becomes part of the local implementation for the air vehicle (the parent component). The corresponding argument structure does not have a component module for the airframe. Any arguments associated with the airframe would be argued in the air vehicle component module.

There is no prescribed manner for deriving a design authority hierarchy. In some cases, the decision may be natural and obvious, while in other cases, multiple interpretations exist. Stakeholders must assess the alternatives and determine a design authority that is appropriate for their use. When a component is dependent upon a design but does not have authority over the design, these dependencies are assumptions about sibling component behavior. These dependencies are explicitly expressed as assumptions and context in the argument structure.

3.3.2.2 Component Contract Views

Component contract views describe hierarchical relationships amongst two system components: a consumer and provider. In a hierarchical relationship, a system component delegates some design

goal to another system component. The delegating component can be thought of as a consumer; the consumed component can be thought of as a provider.

A *component contract* is an argument structure that maps the demands of the consumer to the behavior of the provider, arguing that consumer demands are satisfied by the provider. At the same time, the component contract argues that all assumptions of the provider are met by the consumer. Conceptually, the assumptions of the provider represent demands on the consumer, thus the general notion of demand satisfaction naturally flows in both directions within a component contract. Component contracts justify the satisfaction of consumer assurance goals through assume-guarantee reasoning, as discussed in Section 3.1.4. References to component contract arguments are depicted in the argument structure using GSN contract module reference elements[1], shown in Figure 28[2]. Contract module references encapsulate contractual arguments that themselves further reference argument structures of the provider component.

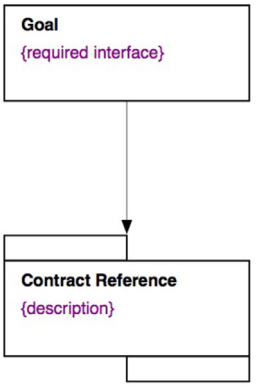


Figure 28: Contract Module Reference

Between any two system component arguments (as represented by component module views), there may be numerous component contracts. The *component contract view* encapsulates all of the component contracts that link any two consumer and provider system components. This encapsulation facilitates quick review of the relationship between a consumer and provider component. Consequently, the component contract view clearly identifies the impact to the argument should details of either the consumer or provider change.

The component contract view thus directly answers the first question raised by practical argument modularity ("Are design demands satisfied by integrated system component

[1] We refer to our specific use of GSN contract module references as "component contracts" because the contract serves to link system components together within the modular system design hierarchy.

[2] We further expand upon the use of contracts beyond what is depicted in this figure using contract schemes, discussed in Section 3.3.8.

behaviors?") by explicitly arguing the satisfaction of design demands by integrated components. We represent the component contract view graphically as shown in Figure 29.

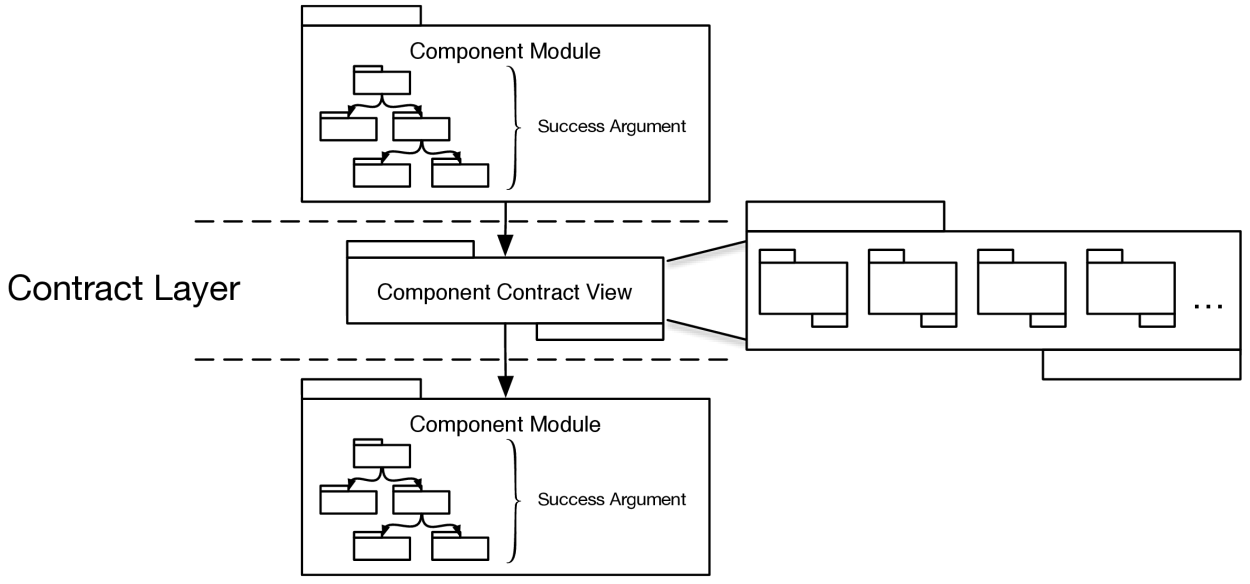


Figure 29: Component Contract View

The component contract view also provides an anchor for addressing concerns that arise from the integration of system components. In particular, residual doubt surrounding the context compatibility between any pair of consumer and provider components must be addressed: the behavior of a component cannot be assumed if it is used or developed outside of its original operational and developmental context.

The component contract view thus also provides support in order to answer the second question raised by practical argument modularity ("Are all system component contexts compatible?") by linking arguments justifying contextual compatibility of integrated component context into the case. We represent the consideration of contextual compatibility graphically as shown in Figure 30. In this representation, context models are depicted as GSN context elements linked to component module views. Contextual compatibility is addressed by comparing context models and arguing compatibility in a confidence argument [12] that is attached to the component contract view as a confidence argument. We further discuss the mechanics of component contracts and contextual compatibility in Section 3.3.7 and Section 3.3.8.

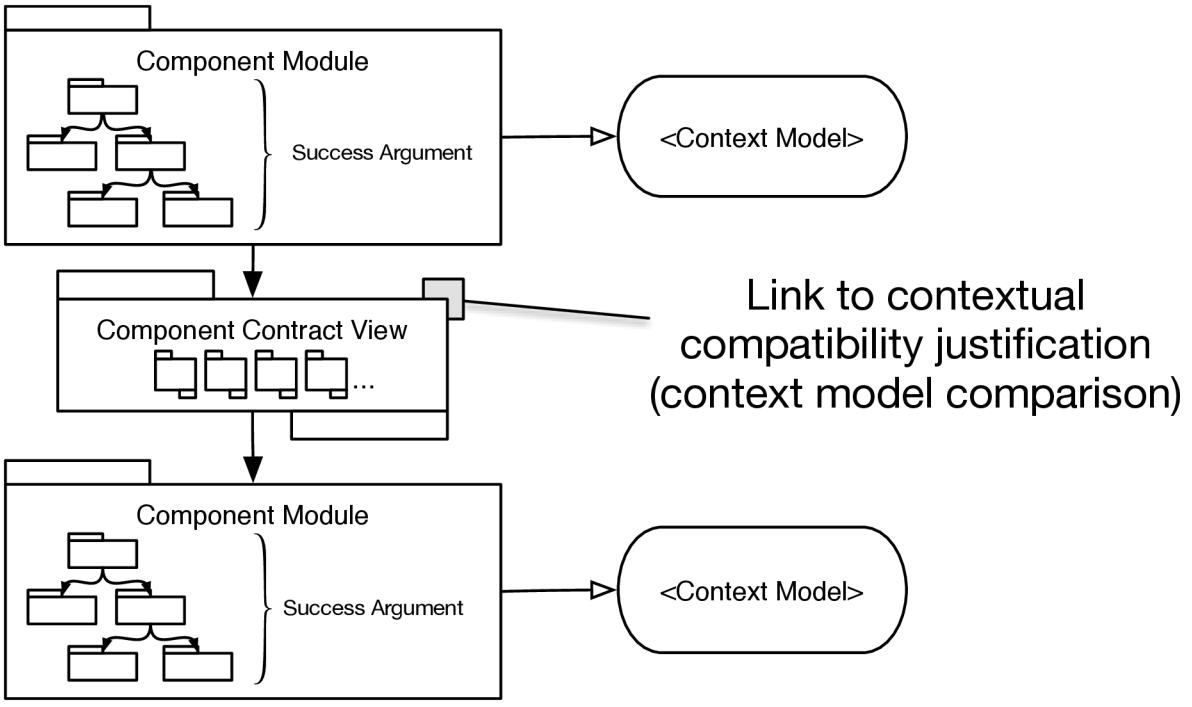


Figure 30: Organizing Contextual Compatibility

3.3.2.3 Sibling Contract Views

In terms of a hierarchical decomposition of components, we refer to any set of components consumed by the same consumer as *sibling components*. Whereas component contract views encapsulate all contracts between a consumer and a singler provider, a *sibling contract view* encapsulates all contracts between a consumer and all providers, i.e., all sibling components (the concept is shown in Figure 31).

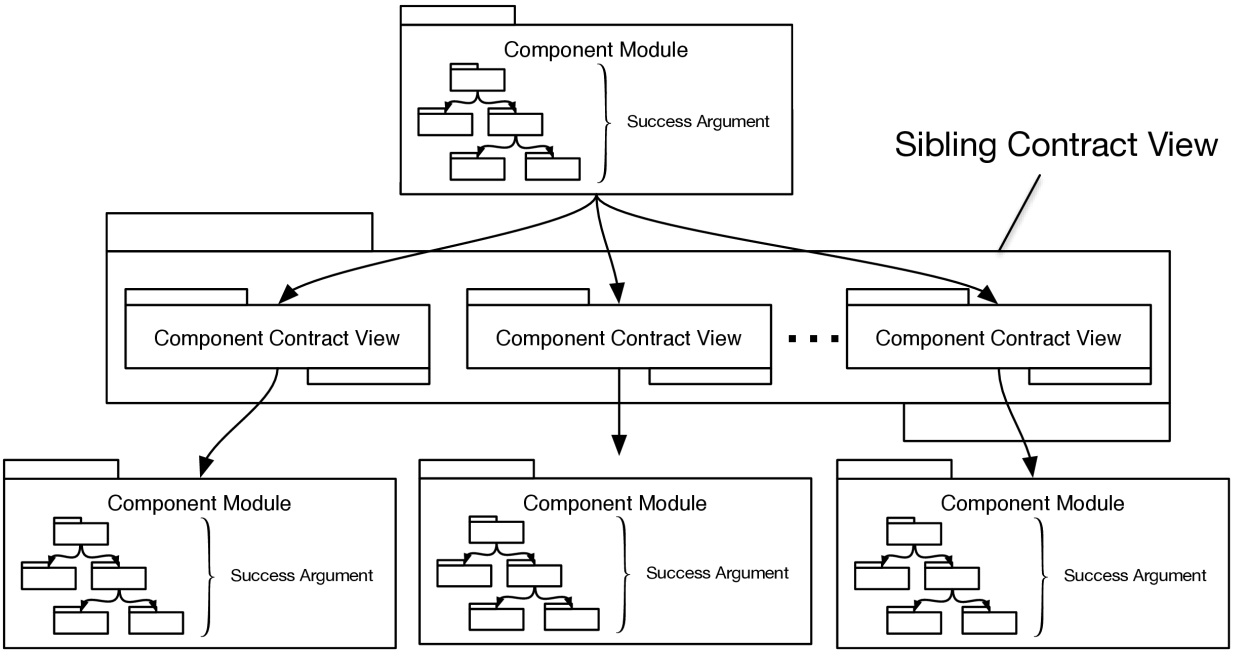


Figure 31: Sibling Contract View

The purpose of a sibling contract view is twofold:

1. It provides a succinct way of representing all contracts that a component relies upon, and can therefore be used to abstractly represent the argument, especially for design authority representations.
2. It provides a convenient anchor for addressing concerns of lateral compatibility (i.e., noninterference) between sibling components that are not directly addressed by the views so far described.

By establishing context compatibility between each pair of consumer and provider components (as is referenced as confidence on component contract views) we have effectively justified that any individual provider component does not adversely interfere with the consumer. While it may be possible to infer that all sibling components will be compatibile by virtue of each individual consumer-provider compatibility argument that has been established, such an inference is indirect and otherwise undocumented. Furthermore, because the issue of sibling compatibility is not directly addressed, it is unclear if such reasoning is sufficient.

Because of the issue of sibling component compatibility is a common and serious concern of compositional reasoning, we provide support for an explicit compatibility justification. As with component contract views, we anchor arguments justifying sibling noninterference using confidence arguments on the sibling contract view (see Figure 32). In this manner, sibling contract views, in combination with component contract views, help provide an answer to the second question raised by practical argument modularity ("Are all system component contexts compatible?").

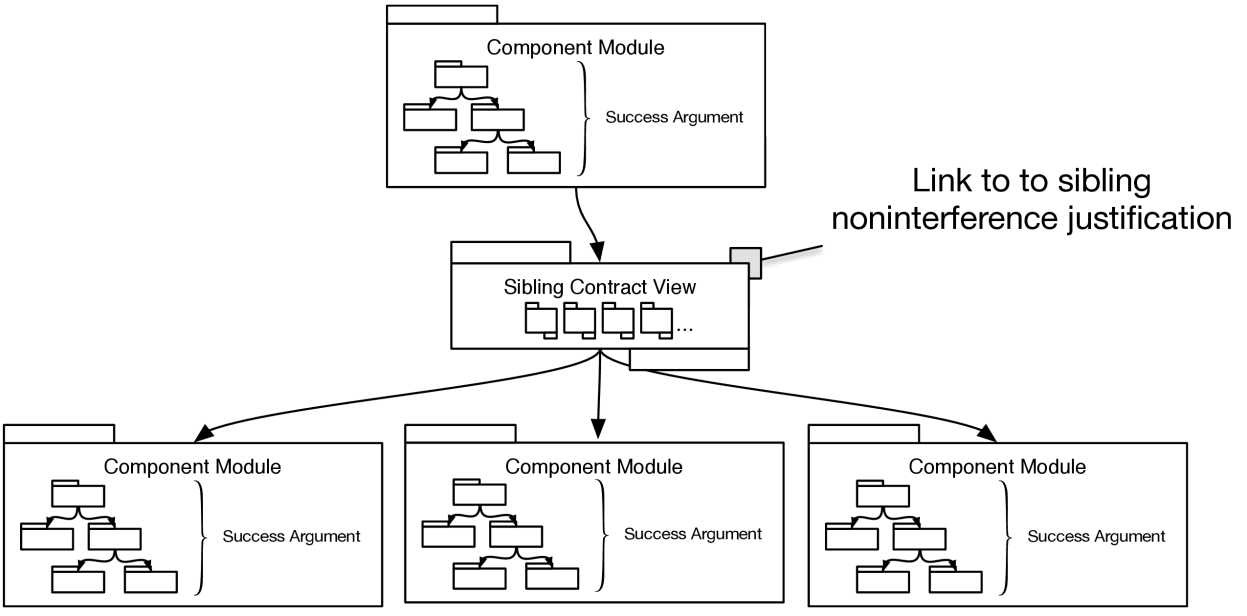


Figure 32: Sibling Compatibility/NonInterference

When the consumer or any one of the sibling arguments change, the sibling compatibility argument must be reassessed. We further discuss the mechanics of sibling compatibility in Section 3.3.3.

The lateral relationships addressed by sibling contract views are only with respect to lateral contextual compatibility. Lateral component design compatibility in order to meet higher-level design goals is addressed as part of the development of design demands. Relationships between siblings are specified as part of the design, i.e., design demands are allocated to components based on a chosen design architecture. For example, the dependency of one component on another sibling component to provide a certain kind of input can be specified as an assumption on a design demand (i.e., specify a design demand under the assumption that the appropriate inputs are provided). While these issues are separate from sibling context noninterference, they are still somewhat related. We further discuss these dependencies as part of the component integration mechanics in Section 3.3.9.

3.3.2.4 System Dependency Views

Ideally, the alignment of argument modules to system components allows all necessary assurance to be gathered compositionally. Unfortunately, the complexity of argumentation allows for arbitrary interdependencies within the argument that have so far not been addressed.

Consider, for example, assurance goals and evidence relating to efficiency, such as run-time or memory efficiency, thermal efficiency, fuel efficiency, etc. Assurance goals about efficiency would typically be argued within higher-level component modules within the design authority hierarchy as these claims are often based on emergent properties of the system as whole. Altering or replacing a component clearly could affect efficiency, thereby necessitating a reassessment of efficiency claims and evidence. The problem is that the impact on efficiency claims might not be noticeable and therefore might not be up to date, especially if there is no hierarchical relationship

between altered system components and the efficiency claims and if altered components are deep within the modular argument hierarchy.

Generalizing this concern beyond efficiency, it is possible that arbitrary system-wide[3] dependencies upon the configuration of components exist throughout the argument that do not follow a hierarchical dependency structure. The *system dependency view* encapsulates systemic cross-cutting concerns. This encapsulation identifies elements of the argument that are likely to be impacted should any system component change. We represent the system dependency view graphically as shown in Figure 33.

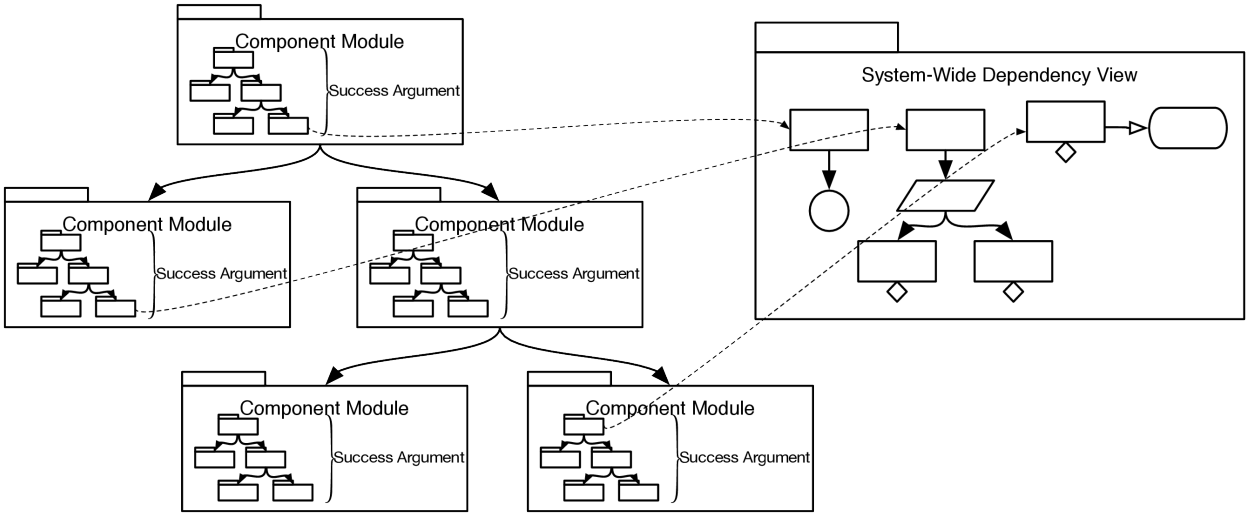


Figure 33: System Dependency View

As arguments are developed, elements within the argument known to rely on a consistent system configuration are flagged. Later, during the system-wide compatibility integration sub-process, the flagged argument elements are compiled into a single view to focus assessment of system-wide compatibility. All those elements within the system-wide dependency view must be reassessed during the integration of any component.

The system dependency view helps to answer the third question raised by practical argument modularity ("Are there new, unaddressed hazards arising from system component integration?"). The view isolates system-wide cross-cutting concerns and dependencies that arise as components are integrated. However, the view is dependent upon correctly and completely identifying all non-structural dependency argument elements, which carries its own risks. Doubts about identification could be addressed using a confidence argument on the view; however, unlike other instances where we have suggested the use of confidence arguments, this instance is confidence

[3] The term "system-wide" is used to refer to the entire system as a whole, where the abstraction of the system is variable, but generally refers to the system as represented by the entire argument structure as is currently available.

not about the integration of components but about the integration process itself. We further discuss doubts about the integration process itself in Section 3.3.11.

The system-wide compatibility is purposefully unoptimized to promote simplicity. Specifically, the view contains all elements that *should* be reassessed, without regard to the component being integrated, i.e., the view is verbose in order to be conservative. It may be possible to prune the view based on the characteristics of the component being integrated; however, we leave these optimizations for future work as such optimizations would require detailed dependency tracking and analyses that introduce further doubt that the view is completely and correctly generated.

The use of the system-wide compatibility view within the SIAT integration mechanics is further discussed in Section 3.3.10.

3.3.3 Mechanics Overview

The concepts and argument structures presented so far provide a foundation for supporting the development of arguments from composed system component arguments; however, more detailed integration mechanics are necessary to facilitate their practical application both for the development and maintenance of complex assurance cases. We identify four primary integration activities focused on establishing the following properties:

1. **Demand Satisfaction**: Justification, through the use of assume-guarantee reasoning, that the demands of the consumer component are satisfied with the guarantees of an integrated (i.e., consumed or provider) component.
2. **Contextual Compatibility**: Justification that the consumer and any given provider component do not have any conflicting constraints, behaviors, etc.
3. **Sibling Compatibility**: Justification that the integrated component is compatible with all other sibling components, i.e., the integrated component does not counteract, degrade, or otherwise conflict with the behavior or properties of other sibling components.
4. **System-wide Compatibility**: Re-evaluation of the validity of any argument structures within the entire assurance case that are dependent upon a specific system configuration or design, and if deemed necessary, updating the associated evidence and argument structures.

The rationale for this division of activities is based on answering the questions posed in the introduction of this section.

We began from a perspective of applying assume-guarantee reasoning to establish demand satisfaction. Further development and investigation of this process yielded cascading limitations with respect to contextual compatibility and integration hazards that we address through the other integration activities.

The ordering of the above activities illustrates a progressive expansion of integration activities in terms of the scope of the involved argument artifacts; however, there is no prescribed order in which these activities are to be carried out. Often, there is an overlap between these activities necessitating context switching between integration processes when practically applied.

To provide a framework in which the integration mechanics are performed and a process model that can be practically executed and expanded, we have documented the above activities in Business Process and Model Notation 2 (BPMN2) [39], shown in Figure 34. This process situates the primary integration activities above within a generic integration process that begins with selecting a component to add or modify, and ends with assessment of the composed argument post-integration. The purpose of these mechanics is to describe key activities when

integrating system component arguments rather than to exhaustively address all possible concerns. As such, the process should be viewed as a generic template of integration mechanics to be refined and altered as necessary, to better address any domain- or application-specific concerns. A detailed description of the sub-processes of the integration process is presented below.

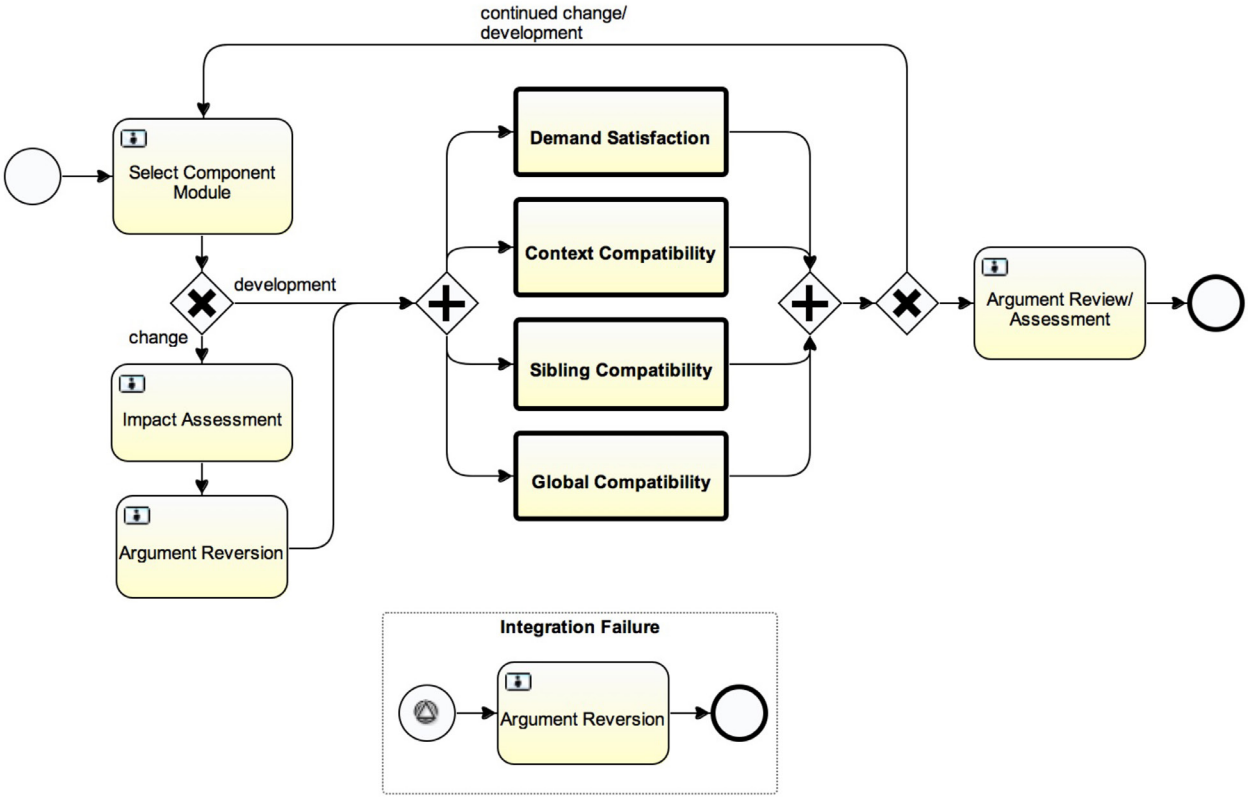


Figure 34: Component Module Integration Process

3.3.4 Integration Scope: Perspective of Component Selection

The SIAT integration process (Figure 34) is designed to address argument development both during initial system development as well as in response to change and general maintenance post-deployment. In both instances, the process is applied from the perspective of the consumer of a system component at any level in a system-of-systems design hierarchy. When the process is performed in response to change, the process is applied from the perspective of the direct consumer of a changed system component . The component module view (Section 3.3.2) provides a container for reasoning about the scope of what is being integrated into a larger argument structure. The SIAT integration process is an activity performed on the argument structure, i.e., the integration of arguments as encapsulated in component module views. The process is assumed to be concomitant with the physical integration of components.

The SIAT integration process is performed at all component module boundaries at some point in the development of a complete assurance case. There are no constraints about the order of system component integration. Integration may proceed hierarchically (top down or bottom up) or occur independently of the development of a larger system, e.g., integration of system components out

of context of any particular system or system of systems. During initial development, the process is performed during solution development (see Section 3.1.3). Post-deployment, the process is performed as needed in response to system change.

How a system component is initially developed or changed is not within the scope of the integration mechanics. The integration mechanics assume that a given system component has already been developed or changed, and instead addresses the impact of integrating the system component's argument within a larger assurance case. The first activity of the integration process is therefore to select or identify the component (potentially from a set of components) in question that will be added or altered[4]. The process repeats for each component identified for integration iteratively.

3.3.5 Integration Failure

Before further describing the integration process, we note that at any point within the integration process, engineers may determine that the process should be paused or terminated. Generally, the integration process allows arbitrary reasons for termination. Obvious reasons include the inability to satisfy demands, justify contextual compatibility, or address other integration hazards as described further in subsequent subsections. To address integration failure, the high-level integration process (Figure 34) defines a generic "exception handling" mechanism that catches any raised integration failure, halts all integration activities, and reverts the argument to a prior or "alternative state".

Repairing the argument to an alternative state admits the possibility that integration failure does not necessarily imply that a new system component must be selected and the prior failing component must be discarded. If a system component cannot be integrated, it still may be viable if other system components change. For example, integration may fail because of interference with a sibling system component. Rather than discarding the new system component, the previously established sibling could be altered or removed.

Argument reversion is therefore a generic concept that accommodates arbitrary causes for termination and provides different failure responses. Furthermore, the activities following integration failure are domain- and failure-specific and consequently left undefined.

3.3.6 Integration for Change: Impact Assessment and Reversion

The integration process is largely the same for initial development and for post-deployment change. The process does, however, differ for post-deployment change when integrating replacements or altered versions of existing system components[5]. Prior to integrating the changed component, the existing component must be excised from the system and consequently excised from the assurance case.

---

[4] The removal of system components without replacement is a degenerate case but applicable to the integration process. A removed component is considered a "changed" component; however, subsequent integration may or may not be applicable.

[5] Changes where a new component is added to an existing system are addressed with the same mechanics as initial argument development.

The goal of impact assessment and argument reversion is to eliminate information and structures from the case that might no longer be applicable given that a system component has changed. Impact assessment determines the extent to which reversion must be applied by identifying elements of the argument and related documents that will require alteration in response to the change. Generally, argument reversion then reverts the prior integration of the system component in question by removing inapplicable information and repairing documentation and argument structures. However, depending on the extent of the change, optimizations may be possible to minimize reversion. For simplicity of discussion, we assume reversion completely restores the argument to a state prior to integration of the component in question.

The specific mechanics of impact assessment and argument repair are considered out of scope for this project effort; however, these concepts have been explored in related work. Argument repair mechanisms have been suggested in Assurance-Based Development [10] [25] and in similar modular argument technologies [38]. Further, impact assessment and argument reversion can be aided by tool support. For example, tools can be developed using a mechanism similar to taint tracking: propagated changes as a result of integrating a component are tracked and recorded during development to assist later impact assessment and reversion. Similarly, version control software, such as Git or SVN, could be used during the integration process. Reversion of the argument could then be effected by reverting prior commits to the repository.

3.3.7 Justifying Demand Satisfaction

During the demand satisfaction sub-process, one or more argument contracts, based on assume-guarantee reasoning (see Section 3.3.1), are developed in order to mutually satisfy provided and required argument interfaces. Each argument contract justifies how a single required interface is supported by one or more provided interfaces of an individual component being integrated.

This process is performed under the explicit caveat that assume-guarantee reasoning alone is not sufficient to compose arguments. In principle, the provided and required interfaces of assume-guarantee reasoning should capture all relevant contextualizing factors, in which case this subprocess would obviate the need for the other primary integration activities. In practice, there are always risks of under specifying interfaces. This risk is primarily an issue of provided interfaces and the failure to list all relevant assumptions (see Section 3.3.1). Failure to specify an assumption could surreptitiously invalidate interface satisfaction, i.e., the interfaces will appear satisfied given available information but are actually incompatible, leaving the demand unsatisfied. Further limitations of assume-guarantee reasoning are discussed in Section 3.3.8.

The role of assume-guarantee within the integration mechanics is to provide the initial foundation for component module integration. The rationale is that if explicit constraints on interfaces have been previously specified, then they provide a foundation for reasoning about interface satisfaction. Within the demand satisfaction subprocess, the provided and required interfaces of assume-guarantee reasoning are fist examined as given, ignoring issues of completeness and correctness. Conceptually, the demand satisfaction process approaches assume-guarantee reasoning deductively[6]. Concerns regarding the quality of the interface and other integration

[6] The separation of deductive and inductive concerns within an argument is similar in concept to related work [40].

hazards that are inherent to inductive reasoning are addressed in other integration processes. In this manner, integration concerns are compartmentalized by integration activity.

A detailed demand satisfaction sub-process diagram is shown in Figure 35, and discussed further in the following subsections.

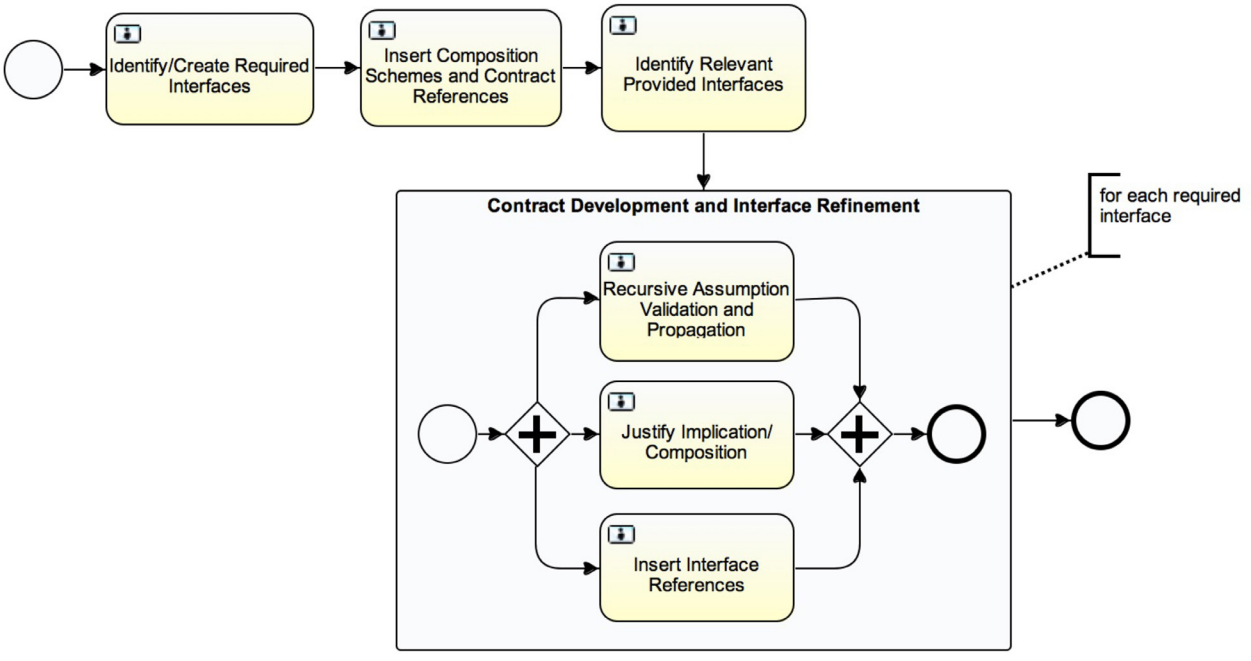


Figure 35: Justifying Demand Satisfaction Sub-Process

3.3.7.1 Identifying Interfaces and Prepping Contracts

The first three activities shown in Figure 35 involve the preparation for argument contract development. Specifically, the identification of relevant interfaces (both provided and required interfaces) and the placement of placeholder contract module references within the argument. In some cases, required interfaces might be previously identified, such as during initial development. Integration in response to change; however, will likely involve an explicit accounting of affected interfaces. In these scenarios, the component contract view for the altered component (the set of all contracts between the system and the altered component – see Section 3.3.2) provides support for quickly isolating relevant required interfaces.

Identified required interfaces are then supported within the argument structure with placeholder references to argument contracts, i.e., references to an empty contract argument. Prior contract references are either removed or updated as desired by the argument developers. In addition to placing reference contracts, placeholder *composition schemes* are also specified. Composition schemes provide reviewers with explication as to the nature of the composition to aid in assessing contextual compatibility. Composition schemes are defined based on the characteristics of the integrated component, further discussed in Section 3.3.8.

Once required interfaces are identified, the provided interfaces of the component being integrated are identified that will serve to support each identified required interface.

3.3.7.2 Contract Development

To effectively justify that a required argument interface is satisfied by one or more provided argument interfaces, a *component contract argument* is developed between each identified required interface and relevant provided interfaces. In SIAT, a component contract argument provides an assume-guarantee-based argument structure that justifies that:

1. the integration of the provided assurance guarantees imply satisfaction of the assurance demand,
2. the assurance guarantees are provided under the specified guarantee's assumptions, and
3. the guarantee's assumptions are valid.

The component contract argument pattern to justify the above assurance goals is shown in Figure 36. The pattern is instantiated for each component contract. Items 1 and 3 above are justified within the contract argument itself. Item 2 is justified indirectly within the contract argument by referencing relevant argument structures of the integrated component's provided interfaces.

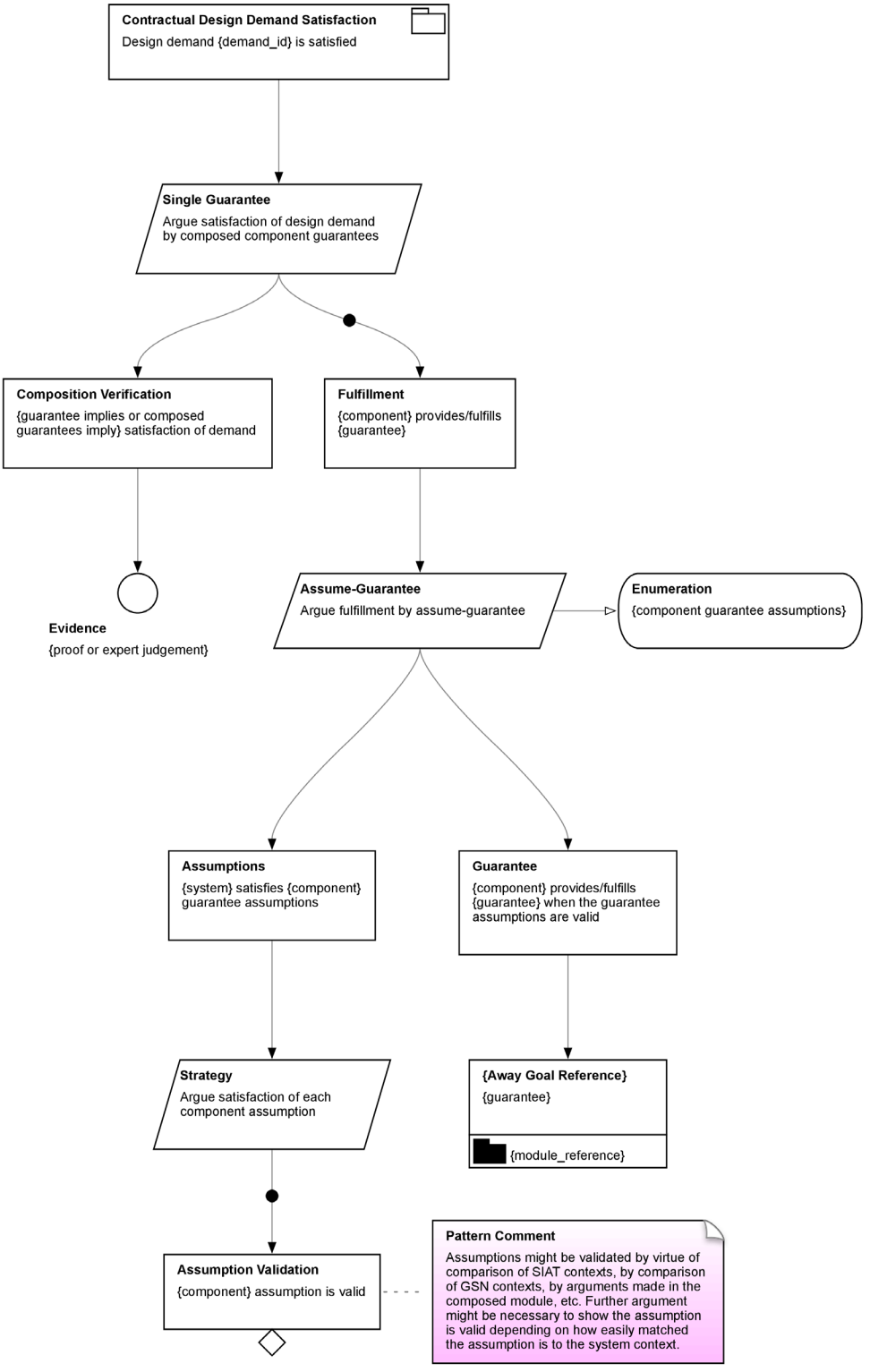


Figure 36: Component Contract Argument Pattern

3.3.7.3 Recursive Interface Refinement

Assumptions that are explicitly invalidated result in integration failure; however, it is possible that assumptions are neither valid nor invalid. In these instances, the consumer's context is insufficient to compare the assumptions of the provided interface, requiring an expansion to the consuming component's context to complete the comparison. The consumer's argument must first be updated with the additional context to complete the comparison. The additional context may be generated from comparable notions already within the consumer's context, but not properly explicated. It is also possible that no comparable contextualization exists, in which case, the assumption of the providing component becomes part of the assumptions of the consuming component.

The addition of a new assumption to the consumer's argument requires re-examination of affected contracts at higher-levels of the argument hierarchy that reference updated portions of the consumer's argument. This in turn might require further updates to the context of components higher in the argument hierarchy. The assumption effectively propagates up the argument hierarchy to any provided interfaces of the consuming component, illustrated in Figure 37. Consequently, any existing contracts based on these updated interfaces must be re-evaluated to verify the assumptions are valid.

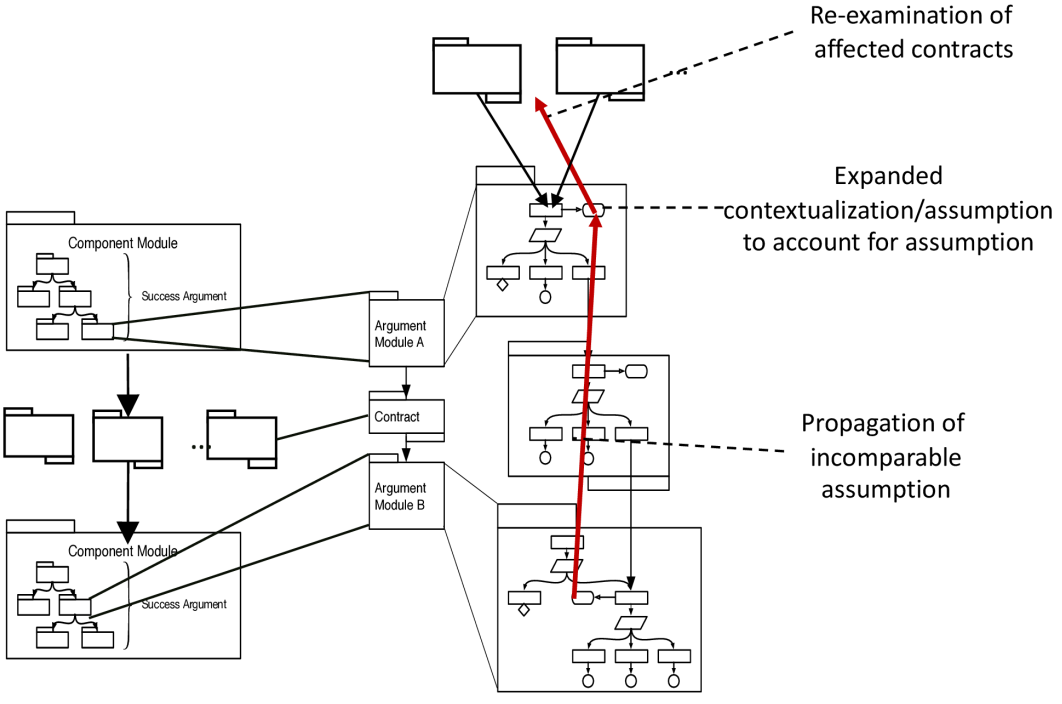


Figure 37: Assumption Propagation

The process of assumption validation and propagation continues recursively up the argument hierarchy until either:

1. all assumptions are validated,
2. the assumption is explicitly invalidated (resulting in integration failure), or
3. the assumption can no longer be propagated (there are consumers of the affected argument).

The benefit of assumption propagation is that integrating a component does not require assessing any children components nested within the design hierarchy. Integration for demand satisfaction can instead focus at interface boundaries. Each consumer throughout the argument hierarchy subsumes the assumptions of its children. Each provided interface therefore includes all relevant assumptions, including relevant assumptions of children components. Propagated assumptions also become part of the context of consuming components to which they propagate (context is further discussed in Section 3.3.8) providing similar benefits to reasoning at interface boundaries.

A potential negative consequence of assumption propagation is that propagation increases coupling between modules in order to preserve reasoning at interface boundaries. Increased coupling increases the difficulty of reverting an integrated component in response to component changes in the future (Section 3.3.6). Reasoning at interface boundaries at low coupling are both desirable properties of modularity; however, in this instance we must violate one modularity property to perserve the other. This tradeoff between necessary violations to modularity principles to preserve other desirable aspects of modularity is an example of practical modularity (see Section 3.3.1). We sought to maintain encapsulation, but found that in order to address the issue of expanding assumptions, a modularity violation is required. As previously discussed, tool support, such as version control software, may alleviate much of the burden of reverting propagated assumptions by managing how assumptions are coupled.

The result of successful contract development and interface requirement is that all interfaces are satisfied and up to date. There are, however, no guarantees that the interfaces are somehow incomplete in other respects. Further assessment of the integration is necessary, discussed below.

3.3.8 Justifying Contextual Compatibility

Contracts formed on assume-guarantee reasoning are based on the underlying assumption that satisfaction of identified interfaces is sufficient to establish a contractual agreement. While practically all current techniques for modular arguments, including SIAT, rely on some notion of assume-guarantee reasoning and contract arguments (see for example the SafeCer [41] and MSSC [42]), there is a wide consensus that such reasoning is, by itself, insufficient to provide assurance that the argument composition is valid. A summary of the challenges/inadequacies associated with assumption-guarantee reasoning for arguments is as follows:

- Assume-guarantee reasoning typically captures functional properties not qualitative/non-function properties (e.g., safety or security).
- Assume-guarantee reasoning is typically used for verification (i.e., demonstrating the system will work correctly) and not certification (i.e., demonstrating the system cannot "*go badly wrong even when other things are going wrong*" [43]).
- The assumptions used for assume-guarantee reasoning are themselves based on assumptions, i.e., the assumptions that are explicitly provided were selected based on the intuition of developers as to what assumptions will be relevant. Missing assumptions might undermine the given guarantee or other goals throughout a larger assurance case. Further, proving that an assumption-guarantee formalism is not over simplified is typically impossible [44].
- Some properties, like safety and security, are system-level concepts e.g., hazards and threats. Consequently, it is unrealistic to expect a module to provide the necessary detail within the defined assumptions to obviate an additional top-down system analysis.

We observe that many of the inadequacies of assume-guarantee reasoning for arguments can be summarized residual doubts about contextual compatibility between argument modules. Defining

the characteristics that must be compared and assessed to validate contextual compatibility is an open problem [45] [42] [44]. A further complication is that the context in which the system and its corresponding argument are developed are just as important as the context in which the deployed system operates. Consider, for example, the composition of a spurious argument developed with known logical fallacies. While the operational contexts may be compatibile, the composed argument will provide a false sense of assurance as it is based on faulty logic. We therefore must consider both operational and *developmental context* compatibility.

The challenge in reaching a consensus on contextual compatibility is that comparison characteristics and assessment criteria are (1) domain-specific and (2) based on the characteristics and use of the argument; however, we observe that there is an additional contributor to variability: (3) *the characteristics of the composition itself*: e.g., composition for argument reuse or composition of a bespoke component. The composition characteristics motivate comparison of specific characteristics of the composed argument modules.

We combine the above three observations within a flexible framework allowing for customization and instantiation in any domain. The framework separates contextual compatibility concerns based on two questions:

1. Is the integrated argument and any prior assessment of the argument independently "*trustworthy*": i.e., is the argument in isolation (ignoring composition into a larger system) of sufficient quality to believe it is complete, sound and valid[7].
2. Is the integrated argument sound and valid once composed into the larger system: i.e., are there contextual incompatibilities between the consumer and provider that undermine assurance goals of the provider.

The first question addresses the concern of contextual compatibility in terms of the component's development (developmental context compatibility). That is, if a component was not developed with a comparable assurance rigor and with common notions of assurance (including common notions of safety hazards, security threats, etc.) as is expected within the larger system, the component's argument may fail to provide the necessary level of support. The goal in supporting compositional arguments is to reuse existing arguments and assessments of those arguments with little or no alteration/reassessment. Simply put, the first question asks to what degree are we able to meet this goal. The second question then addresses compatibly of the system and its component in operation (operational context compatibility) by asking if contextual inconsistencies exist in the composed system that undermine the integrated component's argument.

The SIAT integration subprocess for justifying context compatibility is shown in Figure 38. Developmental context compatibility is resolved through domain comparison and assimilation.

---

[7] The terms "sound" and "valid" are often referred to within deductive reasoning, and therefore may be arguably considered inappropriate when referring to real-world/inductive systems. Comparable terms for inductive reasoning have been suggested such as "consistent" and "cogent"; however, for simplicity, we adopt the soundness and validity concepts with the understanding that the systems we are describing are inductive.

Operational context compatibility is resolved through context model instantiation and comparison. These processes are further discussed below.

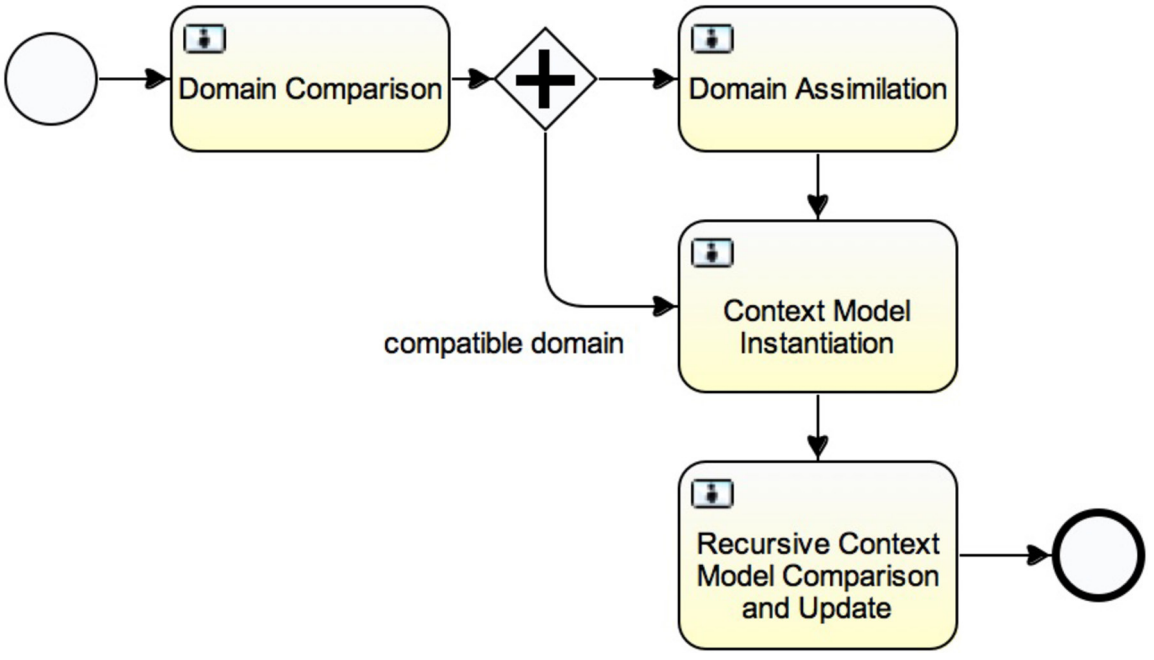


Figure 38: Context Compatibility Justification Subprocess

3.3.8.1 Domain Comparison

In order for a component's argument to be appropriate within a larger system argument, there must be a comparable notion of developmental context, i.e., an agreement must be established about what adequate assurance means. Without some agreement, it is possible that the argument justifies assurance goals with illogical, insufficient, or inappropriate argument and evidence.

We define developmental context as the quality, standards, practices, etc. that go into the development of a component and the development of its assurance case. We include in this definition system-level concerns, such as safety hazards and security threats, to address common concerns that components might fail to consider important system-level properties [44]. As system-level properties, hazards and threats might not be directly applicable to a component nested within the system's design hierarchy; however, we observe that components are developed under some preconceived notion of a larger system that includes common hazards and threats so as to mitigate possible contributing failures.

The developmental context in which an assurance argument is generated is more comprehensively captured in terms of the domain in which the component and its argument are developed. Consider, for example, the domain for commercial aircraft. Any component, such the engines or flight control software, developed within this domain will have known standards and regulation governing the rigor by which components are developed and maintained, and consequently, how assurance is defined and justified. Further, there are known system-level concerns, such as the hazard of NMAC, that are understood by all engineers and stakeholders

within the domain. Components that have been approved for one aircraft might not be appropriately applied on another within the same domain (i.e., operational context compatibility remains to be established), but the question of whether these components have met established criteria of acceptability in and of themselves has been resolved.

We observe that in order to provide complete developmental context compatibility, the domain of the consumer and provider components must either be the same or considered compatible. The first process of contextual compatibility is therefore to determine if the component originates from the same or compatible domain.

Domains, however, are not easily comparable as there is no universally accepted method for domain comparison. While some domain knowledge can be captured and stored in repositories, and used a source for comparison, such as that provided by CLASS (Section 3.1.4), domain knowledge largely exists in the minds of domain experts. Ultimately domain experts must determine the compatibility between domains.

The primary benefit in establishing domain compatibility is that prior assessment and approval of the argument can be reused. If prior assessment does not exist, assessment of the argument is necessary. The benefit of domain compatibility in this scenario is that a common understanding of the existing component and argument is established providing a foundation for assessment, either by the original component developers or by the consumers of the component. The argument must be approved for use in its original context before continuing with integration.

3.3.8.2 Domain Assimilation

If the domains of a consumer and provider are considered incompatible by domain experts, the argument and prior approval of the argument cannot be trusted. Consumers of the component may either (1) choose to discard the component and find a new one, raising an integration failure, or (2) bring the component within their domain. We refer to this later activity as *domain assimilation*. Domain assimilation is largely undefined as it is definitionally a domain-specific activity. Generally, assimilation will likely require the following:

- A complete assessment of the component and argument.
- Modifications to the component and/or argument to make the component and argument compliant with the new domain.
- Communication and support from the original developers.
- Approval of the argument within the new domain as providing acceptable assurance within its original/assumed operational context.

Once domain compatibility of a component is established, by assimilation or otherwise, the prior approval of the argument can be reused as support that the argument is sound, valid and complete in and of itself and can be reused without further developmental context assessment of soundness and validity for other systems developed within the same domain. Further risks of using the component within a new operational context (for example, reusing an engine for a new aircraft) must be addressed separately. These risks differ from system to system even within the same domain, and therefore prior operational context compatibility cannot be reused in the same manner as developmental context compatibility. Addressing operational context compatibility is discussed below.

3.3.8.3 Context Models Instantiation and Comparison

To provide the basis to address operational context compatibility, we establish the concept of a *context model*. A context model specifies a set of domain-specified contextualizing properties to compare to establish operational context compatibility and is similar in concept to a common characteristic map [28]. The types of properties captured in a context model are selected by experts within a given domain. For example, domain experts could stipulate that a component problem description specified as a problem frame [8] is a necessary property in a context model. Because there is no universally accepted definition of context, selected properties and their corresponding forms will be based on domain-specific definitions; however, some properties have been suggested in the literature, which could serve as an initial bases for defining a context model:

- GSN context elements (contexts, assumptions and justifications)
- Standards and practices followed
- Environment descriptions
- Problem descriptions
- Safety and security analyses
- Dependency diagrams (e.g., an interface control document)

To the above list, we stipulate that domain compatibility as established by domain experts should also be part of the context model, although its purpose is to address developmental context compatibility specifically. While individual notions of context models and how they are compared will vary, there will likely be some common properties consistent with many if not all domains[8]. For example, GSN context elements should likely be part of a context model regardless of the domain.

There is similarly no established techniques for comparing context. Comparison strategies ultimately depend on the types of artifacts being compared and the degree of risk accepted by the stakeholders.

[8] There may exist a notion of an assurance argument domain that is subsumed by all domains adopting argumentation. In which case, common notions of context might be absorbed into multiple distinct domains that share the general assurance argument philosophy. We refer to this notion as an *assurance domain*, but leave further investigation of this topic for future work.

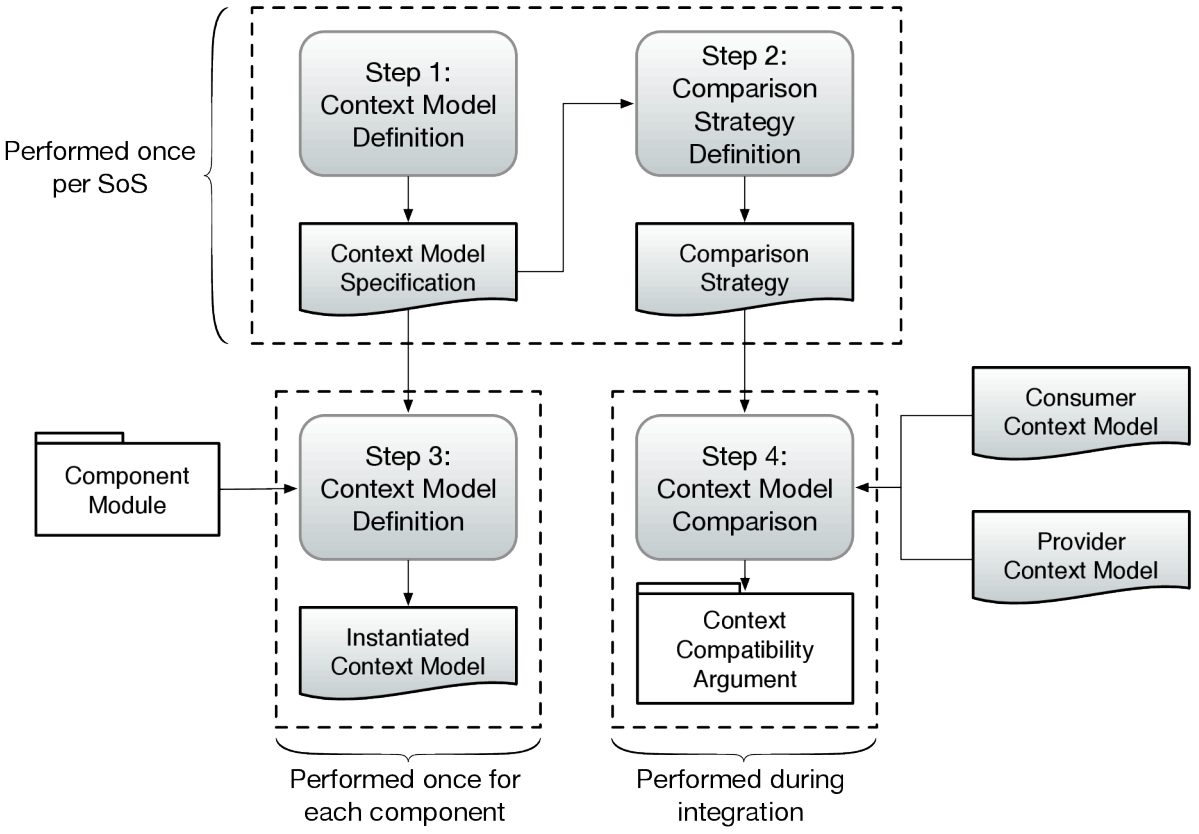


Figure 39: Instantiating and Comparing Context Models

Generally, instantiating and comparing context models involves the following steps (shown in Figure 39):

1. **Context Model Definition**: Domain experts choose the properties that will define the context model or models (discussed further below) for their system of systems. Choosing appropriate context model properties is performed once for the system of systems and the same properties are used for all instantiated context models for the system. Depending on the domain, it may be possible to specify one definition of a context model to be used for all systems within the domain.
2. **Comparison Strategy Definition**: Domain experts define a strategies for comparing models and evaluating the comparison. The strategy might influence the context model definition as specific structures/forms of context properties are chosen. Comparison strategies should include comparison metrics and methods as well as acceptance criteria. As with context model definition, comparison strategy definition is performed once for the system of systems (or once for the domain if applicable).
3. **Context Model Instantiation**: A context model is instantiated for each component module, i.e., the specific engineering artifacts associated with each context property are isolated to form an instance of the context model. Context models are instantiated for every component module as they are integrated. In principle, context models may be dynamically generated based on a given context model definition; however, they may also be predefined. Existing instantiations of context models may be used if (1) the model is still valid within the given

system of systems and (2) the model is representative of the current system (the model is not stale).

4. **Context Model Comparison**: When a component module is integrated by a consuming component module, the two context models are compared. Comparison is performed once between two component modules, regardless of how many component contracts are made between the two modules. Comparison is performed based on the previously defined comparison strategy.

Steps 1 and 2 are prerequisites for context model instantiation and comparison and should be performed in advance of integration; however, when exactly the model and comparison strategy are defined is not specified within the SIAT integration mechanics. Instantiation of the context model is largely the responsibility of the component consumer, although, provider components may specify a context model in advance. Use of existing context models is based on the validity of the definition for the given system of systems.

The degree of scrutiny involved in defining and comparing context models will likely vary depending on the characteristics of the component and the characteristics the composition itself. For example, components developed entirely within the same organization and for the same system of systems may not require the same analyses for contextual compatibility as those components originating from other domains and developed by third parties. To support varying degrees of context compatibility assessment, we do not restrict the number of context model and comparison strategy definitions. The next section provides a discussion on the use of *composition schemes* to organize and explicate the use of alternative context models and comparison strategies. Because of the potential for variability in context model definitions, consumer context models are likely best instantiated as needed (i.e., dynamically) based on component's context model being compared; however, the precise mechanics of context model instantiation are not specified to allow for alternative approaches.

Context model instantiation requires that any relevant development artifacts associated with defined context properties are assembled, often by the consumer. An obvious challenge would be if the required artifacts do not exist or are not well organized so as to be easily found. By virtue of requiring that the component domain be compatible with the consumer, we expect a certain quality to the form and structure of component arguments. As such, the required artifacts should be referenced within the argument structure in a manner than is understood and/or required within the domain; however, we anticipate that additional effort will be required to translate existing artifacts into a standardized form for comparison.

Context comparison may reveal portions of context that are neither valid nor invalid, suggesting an expansion to the consumer's concept of context. The alteration to context must propagate up the design hierarchy to relevant documentation, triggering a recursive update and partial re-evaluation of context compatibility hierarchically. The rationale and methods for supporting context propagation are similar to those previously discussed for assumption propagation, see Section 3.3.7.

As a result of the comparison, the consumer and provider are either:

1. considered contextually compatible with respect to the definition of context compatibility established within the domain and all context models have been updated recursively as required or
2. considered incompatibile triggering integration failure and integration reversion.

The results of successful comparison between models can be documented within an argument as *confidence* in the relationship between the consumer and the provider. The confidence argument should also include a reference to the results of domain comparison described in the prior section. The component contract view (Section 3.3.2) provides a summary of the complete relationship between the consumer and provider components, and therefore serves as an appropriate anchor for linking within the case a confidence argument on contextual compatibility (see Section 3.3.2 for an illustration). Figure 40 presents a confidence pattern for contextual compatibility that can be instantiated based on any context model and comparison strategies.

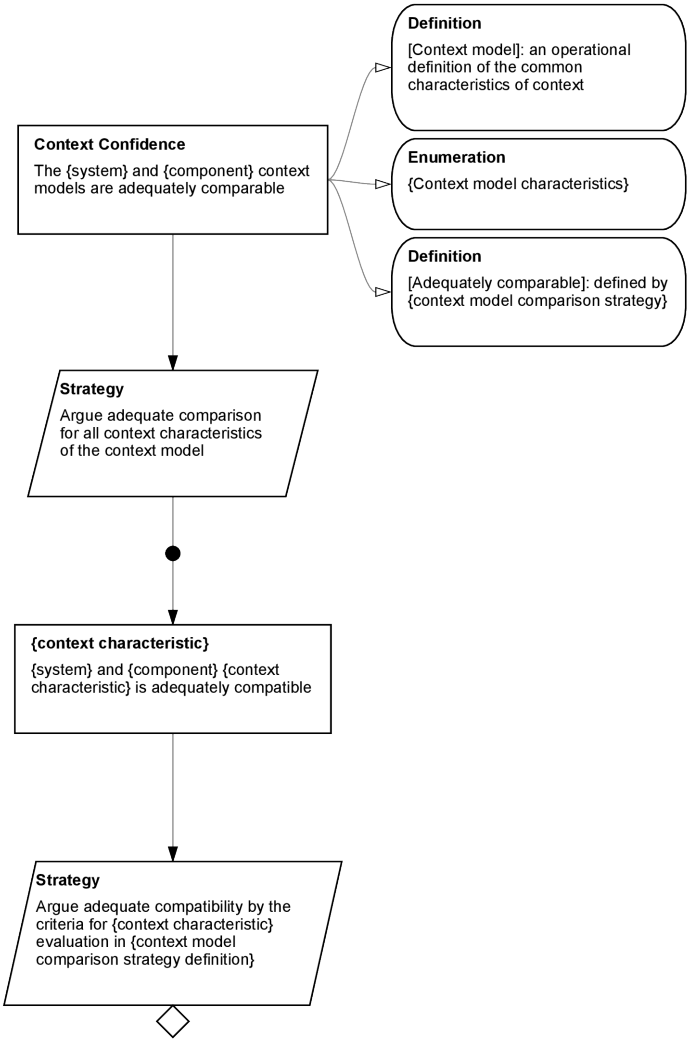


Figure 40: Component Contract Confidence Argument Pattern

3.3.8.4 Composition Schemes

While mechanics for both domain comparison and assimilation and context model instantiation and comparison provide the basis for establishing developmental and operational compatibility, there are two unresolved challenges faced by argument reviewers (e.g., certifiers):

1. The underlying rationale and principles (referred to as the *warrant*) by which arguments are composed is not cited within the case. Because a warrant expresses how the composition of argument modules should be interpreted and the validity of the composition assessed, lack of explicit warrants promotes inconsistent argument development and inconsistent assessment.
2. Assessing the validity of argument composition is currently not well established and subject to change over time and across different domains. Consequently, as composed argument modules are reused and systems and domains evolve, the prior criteria used to support the authority of the warrant may be considered invalid in the future, even within the same domain.

These challenges are emphasized by the SIAT mechanics for contextual compatibility. Multiple context models and comparison strategies may be defined and applied based on the characteristics of integrated components. Without explicit documentation of what context models and comparison strategies were used, the rationale for compatibility is likely unclear to reviewers. Furthermore, we acknowledge that how developmental and operational context compatibility are established will undoubtedly change over time. When these changes occur, affected argument module compositions should be flagged and contextual compatibility reassessed. Without some organization and management of the impact of changes to contextual compatibility criteria, locating what portions of the argument that need to be reassessed and updated is obscured, thus reducing the practicality of modular arguments.

We address these challenges in part through the application of *composition schemes*, an extension of a recently proposed concept for providing warrants for evidence, called *evidence schemes* [46]. Rather than immediately support a required interface goal with a contract reference element, as is traditionally the case, a composition scheme, notated as a GSN strategy element, is placed between the goal and the contract, shown in Figure 41. The strategy element explicitly identifies the "scheme" by which doubts about the composition are assessed. For example, the following properties may be used to identify a composition scheme:

- The provenance of the provider component domain, i.e., endogenous or exogenous.
- The provenance of the prover component system, i.e., endogenous or exogenous, or system "type".
- The cardinality of the composition, i.e., one-to-one between required and provided interfaces or one-to-many.
- Prior approval/certification.

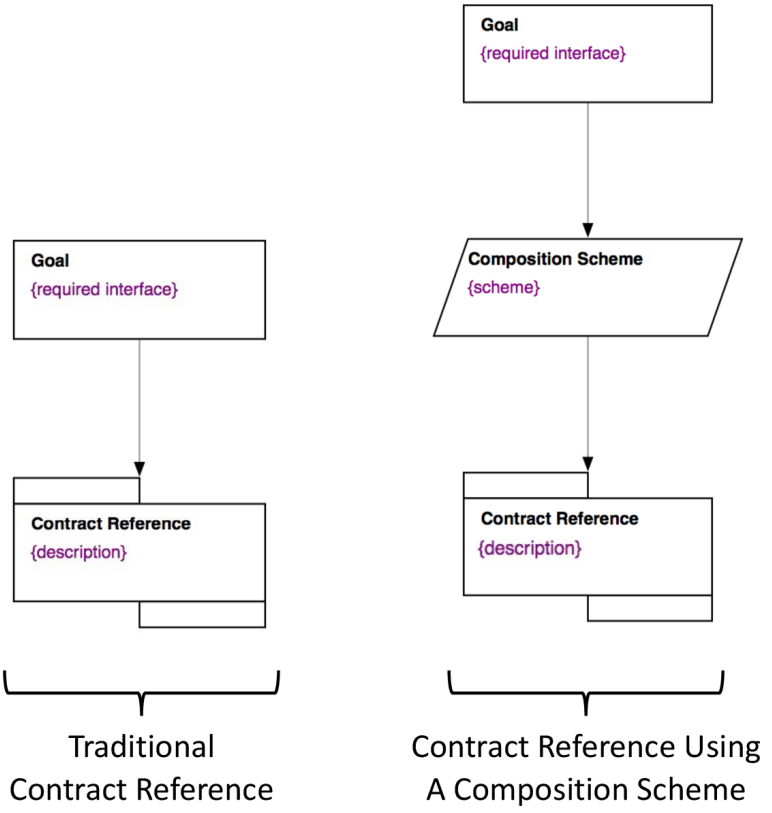


Figure 41: Composition Schemes

Schemes are selected from a prescribed set of schemes established within a domain to ensure consistent use. The scheme communicates to reviewers the compositional concerns that must be addressed, and links to the domain-specific practices for addressing these concerns.

If consistently applied, changes within the domain to the practices for composing modular arguments can be easily propagated to relevant portions of existing arguments and contextual compatibility can be reassessed as necessary.

For example, composition schemes may be defined in terms of an instantiation of the following form:

Appeal to a component from an (endogenous | exogenous) system within an (endogenous | exogenous) domain with (no | prior) approval providing (direction, i.e., one-to-one | indirect, i.e., one-to-many) support

An instantiation includes the following:

Appeal to a component from an *exogenous* system within an *endogenous* domain with *prior* approval providing *indirect, i.e., one-to_many* support

This scheme conveys useful information for the development and review of the argument:

- An exogenous system highlights operational context compatibility as a concern. Reviewers will expect a rigorous context model comparison.

- An endogenous domain means developmental context compatibility is less of a concern. Reviewers might not expect any documentation about domain assimilation, but will question how domain comparison was performed[9].
- Prior approval within the an endogenous domain means the need for further assessment of developmental context compatibility might not be necessary. Reviewers might require a reference to the prior approval, but will likely not need to scrutinize the approval given the approval was derived within the same domain.
- Indirect support, i.e., a one-to-many mapping between a demand and provider guarantees is an indication that the specific contract must be developed to show how the sum of all guarantees implies satisfaction of the demand. Reviewers should expect to find an implication justification within the contract.

If, for example, the context model for comparing exogenous system components changes in the future, schemes like the one above referencing exogenous systems can be easily identified and flagged to focus further review of the case. In this manner, composition schemes provide a novel kind of modularity and support within the argument:

Composition schemes modularize and organize the use of domain knowledge within the case.

The detail of the composition scheme hints at the scale of the contextual compatibility assessment that is performed. Highly abstract composition schemes requires that contextual compatibility assessments cast a wider net in terms of the context properties are assessed and to what degree. More detailed schemes indicate highly refined, focused and perhaps more established/accepted compatibility assessments. As the detail increases down to the very specific characteristics of components, composition schemes aid in defining a product line of argument/system composition.

While composition schemes provide benefits to reviewers to explicate composition and to manage the connection to evolving domain knowledge, they negatively impact modularity by increasing the coupling between modules: consumer components using composition schemes will reference properties of the specific component being consumed. It may be possible to decrease coupling by encapsulating the composition scheme within the contract argument; however, we leave notation refinements for future work. Because the coupling is localized to required interfaces, our current model of composition schemes can be easily updated and is amenable to automated tool support to minimize the effort in altering schemes as components are changed in the future.

3.3.9 Justifying Sibling Compatibility

Contextual compatibility as so far described addresses compatibility only between a consumer component module and an individual consumed (provider) component module. The rationale for this approach was to compartmentalize and focus integration concerns incrementally. A consumer may, however, consume more than one component module. The collection of consumed component modules are referred to as *sibling components*.

[9] A composition scheme could further reflect how the provenance of a component is derived.

In principle, if new assumptions and context of a consumed component are propagated as suggested in the above mechanics, sibling component compatibility should be inferred by transitivity. That is, for a consumer X and sibling components A and B, if the context and assumptions of A are compatible with X, and the context and assumptions of B are compatible with X, then the context and assumptions of A and B should be compatibile. As each component is integrated, the consumer component (X) is updated as necessary with new context/assumption of its components (A and B). A consumer component subsumes the context/assumptions of all currently consumed components. As a result, the order of composition should not matter (argument composition is commutative) and sibling compatibility is maintained progressively.

In practice, sibling compatibility is a property of specific concern for compositional arguments, both in argument development and review. Lack of an explicit argument may be considered unacceptable, especially as components are altered and added over time, obscuring implicit inferences of noninterference. If an explicit justification of sibling component noninterference is desirable, a noninterference (i.e., sibling compatibility) argument can be expressed as a confidence argument on the sibling contract view (Section 3.3.2).

Since noninterference may be achieved by previously described integration mechanics, the exact form on a noninterference argument is left open to address any domain-specific doubts; however possible options include:

- a composite of all consumer-provider contextual compatibility assessments,
- a matrix of compatibility between every subset of siblings
- an argument expressing confidence in the integration mechanics described above as not requiring other arguments of sibling noninterference.

3.3.9.1 Related Sibling Concerns: Design Siblings

Sibling component compatibility as described is largely an issue of contextual compatibility between consumed components. There is also a concern about the compatibility of behaviors/demands on components, especially as components change over time.

Assurance demands placed on components should be specified as generally as possible to promote flexibility; however, it is often the case that a demand is based on limitations of a known component or a set of components. Furthermore, a demand may be carefully balanced amongst the constraints of other sibling assurance goals (Figure 42) in the argument structure. The issue is that there may be inherent coupling of assurance demands to the component. As components change, the design and relevant design demands may need to change. As a result, other sibling assurance goals may be affected. Whats more, the extent of the change may propagate up the argument hierarchy, affecting goals related detailed design, system specification, and potentially even requirements.

Consider the example illustrated in Figure 43. In order to satisfy the reliability goal (probability A per operational hour), reliability specs are developed for relevant subsystems. In this example, reliability probability X and Y are balanced between two subsystems, but how are these probabilities chosen? It is possible they are arbitrarily defined (e.g., an even division), but more likely, they are chosen based on known or expected constraints of each subsystem, which may be implemented by other modular components. In this example, it is possible that component1 may provide better or worse reliability than is demanded:

- If the reliability is better, it is possible to relax probability Y, and perhaps alter the implementation of subsystem2 to take advantage.
- If the reliability is worse, the component could be discarded as an integration failure; however, if there was some flexibility in subsystem2, the reliability constraints could be rebalanced in a way to make the component's reliability acceptable.

We refer to this kind of compatibility as *design sibling* compatibility. During component integration, the design can be reconsidered and rebalanced as necessary; however, for simplicity of the current effort, we instead focus on the satisfaction of demands as given, and leave the precise mechanics for rebalancing the design and assurance demand for future work.

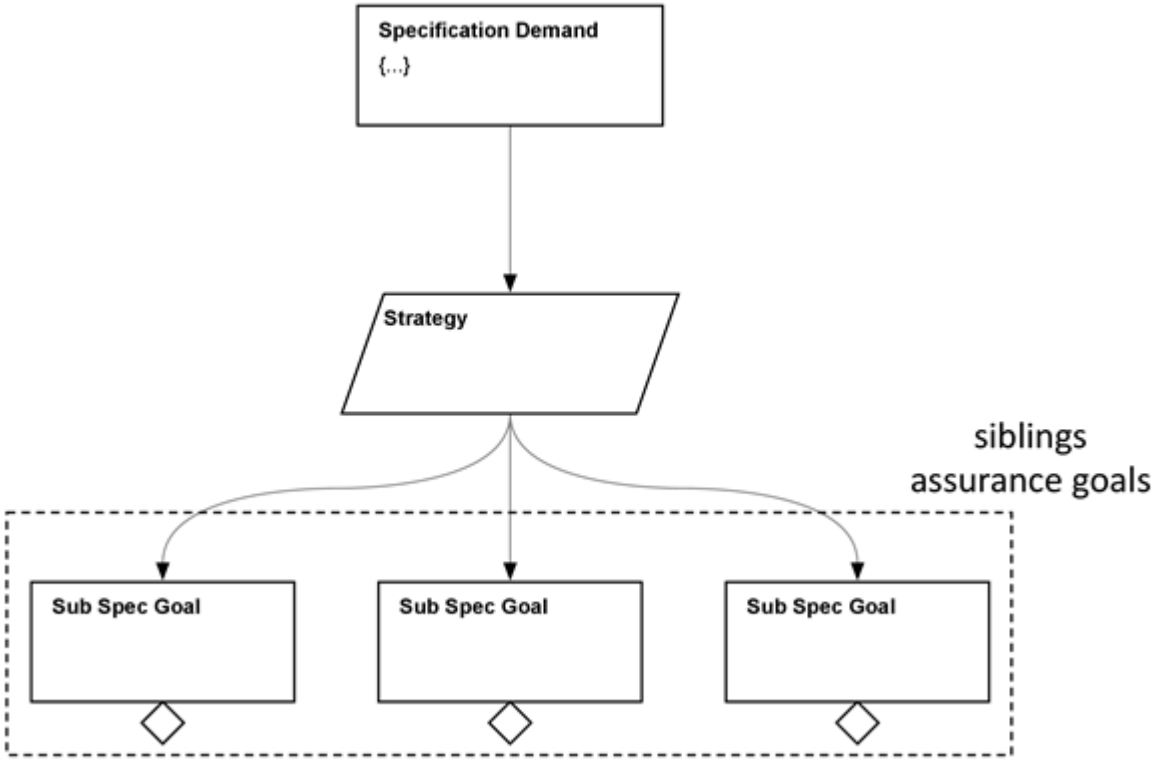


Figure 42: Sibling Assurance Goals

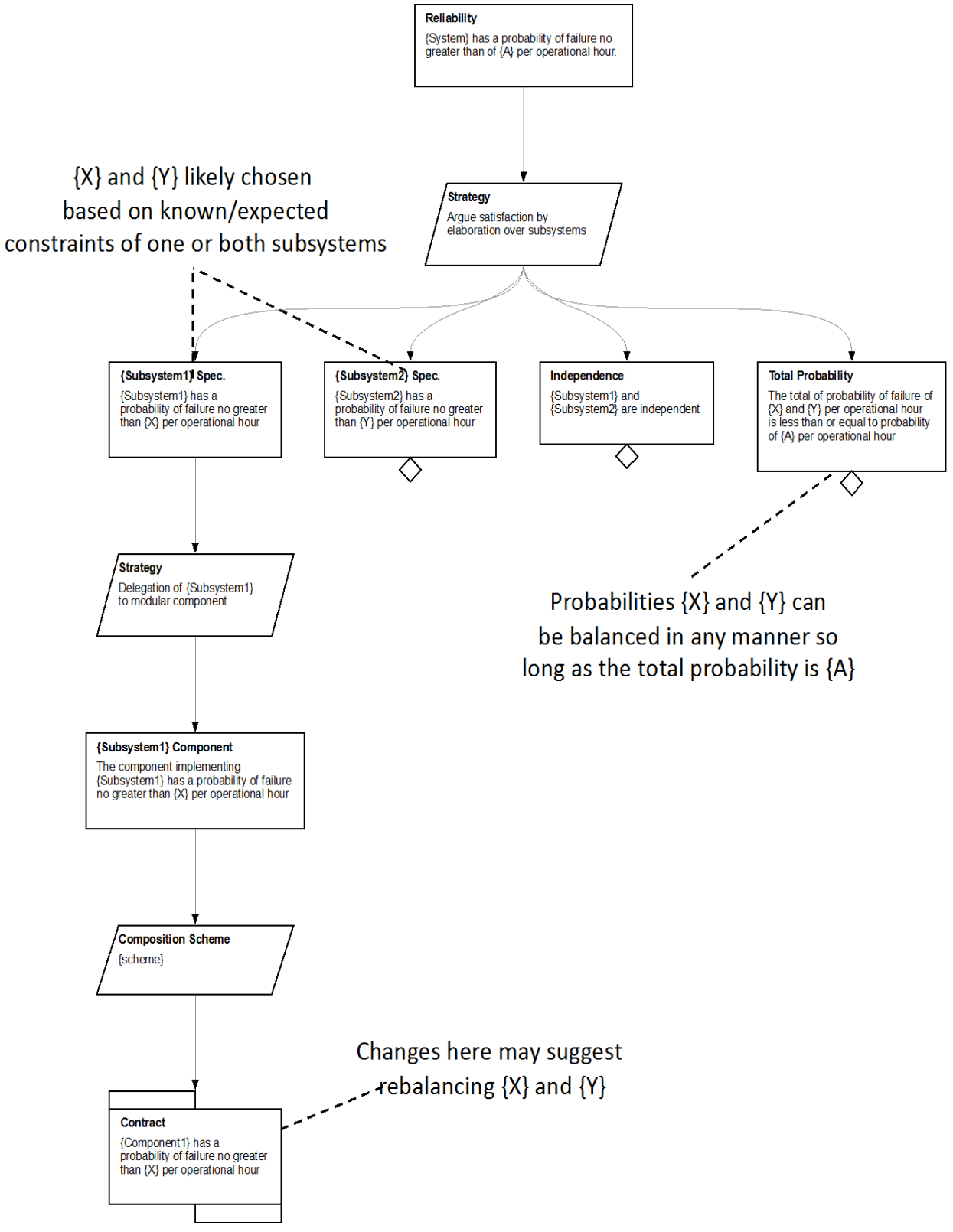


Figure 43: Sibling Assurance Constraint Balancing

3.3.10 Justifying System-wide Compatibility

The mechanics as so far described address the concerns of integration systematically but under implicit assumptions about the hierarchical structure of the argument and the locality of dependencies. In a simplistic argument hierarchy, there are likely no "system-wide" cross-cutting constraints. Each branch of the argument could then be considered to be independent of other branches. Further, any dependencies between components would exist only at the interface boundary. The complexity of argumentation, however, allows for arbitrary interdependencies within the argument that have so far not been addressed (the complete problem description is given in Section 3.3.2).

To address arbitrary system-wide dependencies, i.e., to provide assurance of system-wide compatibility between components, we proposed the system-wide dependency view (Section 3.3.2). The system-wide dependency view is based upon identifying elements of argument that are at risk should any component change within the design hierarchy. Goals supported by testing data have already been suggested as an example; however, Figure 43 provides another example. A goal supporting independence of the two subsystems is potentially affected by any changes whatsoever to either subsystem.

The identification process is not specified within the integration mechanics explicitly, and is assumed to occur during argument development or during domain assimilation (Section 3.3.8). This approach is clearly dependent upon complete identification of argument entities based on system-wide dependencies. Doubts about identification could be addressed within a confidence argument anchored on the system-wide dependency view, however, these doubts are an instance of many possible doubts in applying the SIAT integration mechanics, and not doubts about the system the argument represents. Further discussion about doubts about the integration process itself is given in Section 3.3.11.

If sufficient confidence exists that the system-wide view is complete, then justifying system-wide compatibility is relatively straightforward. Each argument fragment within the system-wide dependency view is either reengineered (new evidence is developed and/or the argument fragments are replaced), or evaluated on a case-by-case basis and updated as necessary.

3.3.11 Argument Assessment

An assurance case is assessed to determine if there is sufficient belief in the top-level assurance claim. While the assurance case documents the rationale for belief in a top-level claim, it is up to system stakeholders to pass final judgment as to whether the top-level claim is adequately supported. The assurance case is therefore a tool to aid decision makers as to whether a system should be deployed, but the final decision is always made by the system stakeholders, not the assurance case itself. Argument assessment is therefore a stakeholder-driven process of reviewing and approving the argument. The precise use of the argument in the approval process may vary by domain.

Using the problem-oriented argument structures as previously described (Section 3.2), we can select a component at any level of abstraction within a system-of-systems design, including the entire system of systems itself, and the top-level goal is always the same: the component in question was successfully developed. A benefit of this approach is that the argument assessment activity can be similarly modularized and recursively applied (either top down or bottom up), and consequently serves to potentially decrease assessment effort and cost.

Modular argument assessment involves answering the following questions:

1. Does the argument module for each system component, viewed in isolation, adequately justify successful development of the associated system component?
2. Where a system component is itself composed of integrated system components, are those system components adequately integrated?

Modular argument assessment is recursive: these two questions are asked for every component in the argument structure, conceptually starting at the highest level of the currently available argument structure. Furthermore, the assessment is modularized to take advantage of prior assessment and approval of system component arguments. If, for a given system component, both of the questions asked by modular argument assessment have been previously answered affirmatively within the same domain, assessment does not need to continue recursively down the argument structure for the system component in question.

The first question addresses the fundamental concern of any traditional argument assessment, i.e., is the top-level assurance claim is adequately supported. Component modules are assessed in isolation by assuming that the context of the component is valid and that any consumed components will adequately support associated claims. By modularizing the assessment in this manner, prior assessments can be reused. Other than these assumptions, this assessment activity follows a traditional argument assessment. We therefore focus on answering the second assessment question.

The second question is answered by evaluating the artifacts and activities of the integration mechanics so far described. The premise of the integration mechanics is that to successfully integrate arguments about system components we must address three fundamental concerns:

1. Are design demands satisfied by integrated system component behaviors?
2. Are all system component contexts (both in development and use) compatible?
3. Are there new, unaddressed hazards arising from system component integration?

These questions serve as the driving motivation for all integration activities and argument structures. The first question is addressed by assume-guarantee reasoning, and the development of component contracts discussed in Section 3.3.7. The first question is also partially addressed by reassessment of design siblings and rebalancing constraints as necessary discussed in Section 3.3.9. The second and third question overlap, in that failure to maintain contextual compatibility is a hazard of system component integration. We highlight the contextual compatibility hazard in particular because of its importance given the known limitations of assume-guarantee reasoning. We partially address the limitations of contextual compatibility for assume-guarantee reasoning in Section 3.3.8. We further address the limitations with respect to contextual compatibility between sibling components (i.e., sibling noninterference) in Section 3.3.9. The third question addresses miscellaneous integration hazards. We identify that there are system-wide compatibility concerns that must be addressed when integrating components in Section 3.3.10[10]. Assessment of the adequacy of integration is therefore a

[10] This hazard of integration may also be viewed as a contextual compatibility hazard.

stakeholder evaluation of each of these integration activities by evaluating any associated altered or developed argument artifacts.

There are, however, additional integration hazards not addressed in the mechanics discussed so far that are a consideration during argument assessment:

the integration mechanics themselves might be incomplete or incorrectly applied.

Example doubts include:

- Have all argument structures that have system-wide dependencies been appropriately tagged so as to facilitate system-wide assessment mentioned in Section 3.3.10?
- Have assumptions been properly propagated and interfaces properly updated during when developing contracts (Section 3.3.7)?
- If interfaces are dynamically generated (see (Section 3.3.7), how do we know the generation process didn't miss anything?
- Are the set of composition schemes (Section 3.3.8) up to date, i.e., have changes to contextual compatibility assessments been propagated sufficiently to consistently inform reviewers and argument developers?
- Are there other integration hazards not addressed by the integration mechanics?

The rationale for the completeness of the integration mechanics is justified through the progressive evolution and discovery of limitations as presented in the above sections. In principle, this implicit argument could be documented using GSN; however, the underlying concern is the authority of our proposed approach to integration (the "backing" in the parlance of the Toulmin model of argumentation [11]). A GSN argument for the approach does not give the approach authority (i.e., acceptance of the approach), only stakeholders can grant authority. In principle, the authority of the approach would be given within an *assurance domain*: a domain prescribing common/accepted practices for the development and maintenance of assurance artifacts, including assurance arguments. The development of the integration mechanics has revealed the need for assurance domains, but this topic extends beyond the current project effort. Further research into the concept of assurance domains is therefore left for future work. For simplicity of this effort, we assume the authority of the integration mechanics.

If the integration mechanics are accepted as complete, the correct application of the integration mechanics is a special case of a more general concern common to assurance arguments: epistemic doubt [46] [40]. These doubts can be addressed through the development and assessment of confidence arguments [12]. Since these doubts are universal to any argument, regardless of modularity, we simplify this effort by focusing more on assumed correct application and leave organizing residual epistemic doubts within a modular argument architecture for future work.

3.4 Compositional Analysis Framework for Systems of Systems

The compositional analysis framework provides an environment in which formal validation and the verification of composition can be performed. Evidence provided by formal validation and verification supports critical claims in the assurance case:

- Evidence from validation supports claims that component requirements are satisfied by component implementations.

- Evidence from verification of composition supports claims that component interfaces are compatible with system interfaces, forming a contract between the system and its component.

This evidence, when coupled with supported claims of contextual compatibility, yields a high degree of confidence that composition is valid and will enable the system to satisfy its requirements.

A critical component of SIAT is the identification of the system context — including the environment in which the system must operate. Complete, accurate and early consideration of context is essential to system development, to system composition and to the corresponding argument.

Assume-guarantee reasoning provides a powerful framework for reasoning formally about the composition of system components and, moreover, underlies argument composition. As discussed in Section 4.1.4.5, system components are described in terms of formal interfaces that document both the syntax and semantics of what the component assumes about and guarantees to its environment.

Semantics are real-world concerns that typically have no representation in formal systems. In the development of SIAT, we developed frameworld for including real-world type information, manipulating real-world types, and documenting and reasoning about the correspondence between real-world types and their machine-world representations.

3.4.1 Primitive Real-World Types

Primitive real-world types are drawn from base measures identified by the International System of Units, and include:

- Length
- Mass
- Time
- Electric Current
- Thermodynamic Temperature
- Amount of Substance
- Luminous Intensity
- Angle

One way to set up syntax supporting these base measures is:

```
measurement: NONEMPTY_TYPE =
  [#
    value:       real,
    scaling:     posreal,
    length_dim:  real,
    mass_dim:    real,
    time_dim:    real,
    current_dim: real,
    temp_dim:    real,
    intens_dim:  real,
    angle_dim:   real
   #];
```

Along with support for mathematical operations, this allows us to write PVS statements such as:

```
  distance: measurement = 3 * m;
  N: measurement = kg * m / s^2;
  piston_pressure = 3 * N / cm^2;
  c_dist: measurement = sqrt(a_dist^2 + b_dist^2);
```

An important consideration is that we want to prevent certain types of operations, such as adding meters to centimeters *implicitly* and combining units from different systems *implicitly*. There are times when it makes sense to perform either of these actions, but making such operations require *explicit* steps allows us to detect errors in models (e.g., Simulink models) where adding 1 meter to 25 centimeters will yield 26 with ambiguous units.

Support for these primitive real-world types has been added to PVS in the form of measurement libraries. Handling systems of units to prevent the implicit combination of units from differing systems requires tradeoffs between simplicity and rigorousness. To explore these tradeoffs, we have created two different libraries for defining how units can be combined, a *system-field library* and a *system-templated library*.

3.4.1.1 System-Field Library

In the system-field library a measurement type is defined as previously discussed, except that a field is added to track the system:

```
  measurement: NONEMPTY_TYPE =
    [#
      value:       real,
      system:      system_enum,
      scaling:     posreal,
      <same as before>
     #];
```

In this library, the possible values of `system_enum` are `NOT_APPLICABLE` (for dimensionless measurements), `ANY` (for units that are system agnostic, such as seconds), `METRIC`, and `IMPERIAL`. Lengths are defined in their own theory, and are defined by the following predicate:

```
  length?(m: measurement): bool =
    valid_measurement?(m) AND
    dimension_match?(m, zero_measurement WITH [`length_dim := 1]);
```

The `lengths` theory also pre-defines several units:

```
  zero_length: length = zero_measurement WITH [`system := ANY, `length_dim :=
1];
  unit_length: length = zero_length WITH [`value := 1];
  m: poslength = unit_length WITH [`system := METRIC];
  cm: poslength = m WITH [`scaling := 1/100];
  mm: poslength = m WITH [`scaling := 1/1000]
  ft: poslength = unit_length WITH [`system := IMPERIAL];
```

The `times` theory is similar, but the definition of second is system-agnostic:

```
zero_time: time = zero_measurement WITH [`system := ANY, `time_dim := 1];
unit_time: time = zero_time WITH [`value := 1];
s: postime = unit_time;
```

The proper rules for mathematically combining measurements must also be defined. For example, the addition operation for two measurements is specified as:

```
+(x: valid_measurement, y: {m: valid_measurement | unit_match?(x, m)}):
  {m: valid_measurement | unit_match?(x, m)} =
    IF preferred_system?(x, y) THEN
      x WITH [`value := x`value + y`value]
    ELSE
      y WITH [`value := x`value + y`value]
    ENDIF
```

This operation requires (via the `unit_match?` predicate), that the second operand (`y`) has the same dimensions, system, and scaling as the first operand (`x`), and the result likewise has the same dimensions, system, and scaling as `x`, with only the `value` field modified. Specifically, the `unit_match?` predicate is defined as:

```
unit_match?(x: measurement, y: measurement): bool =
  dimension_match?(x, y) AND
  system_match?(x, y) AND
  (x`scaling = y`scaling);
```

The `dimension_match?` predicate is defined as:

```
dimension_match?(x: measurement, y: measurement): bool =
  (x`length_dim = y`length_dim) AND
  (x`time_dim = y`time_dim) AND
  (x`mass_dim = y`mass_dim) AND
  (x`current_dim = y`current_dim) AND
  (x`temp_dim = y`temp_dim) AND
  (x`intens_dim = y`intens_dim) AND
  (x`angle_dim = y`angle_dim);
```

The `system_match?` predicate is defined as:

```
system_match?(x: measurement, y: measurement): bool =
  valid_measurement?(x) AND valid_measurement?(y) AND
   ((x`system = y`system) OR (NOT explicit_system?(x))
    OR (NOT explicit_system?(y)));
```

The `valid_measurement?` and `explicit_system?` predicates are defined as:

```
valid_measurement?(m: measurement): bool =
```

```
    (m`system /= NOT_APPLICABLE) OR
    (dimension_match?(zero_measurement, m))

  explicit_system?(m: measurement): bool =
    (m`system /= NOT_APPLICABLE) AND (m`system /= ANY)
```

The `preferred_system?` predicate in the specification of the addition operation is required to ensure that addition is commutative (i.e., that $a+b=b+a$):

```
  preferred_system?(x: measurement, y: measurement): bool =
    IF explicit_system?(x) THEN
      TRUE
    ELSIF explicit_system?(y) THEN
      FALSE
    ELSIF (x`system = ANY) OR (y`system = NOT_APPLICABLE) THEN
      % either they're both all, both n/a, or x is all and y is n/a
      TRUE
    ELSE
      % x is n/a and y is all
      FALSE
    ENDIF
```

The specification of the multiplication operation multiplies the values of the measurements and sums their dimensions:.

```
  *(x: valid_measurement, y: {m: valid_measurement | system_match?(x, m)}):
    {m: valid_measurement | system_match?(x, m)} =
      (#
        value       := x`value * y`value,
        system      := IF preferred_system?(x, y) THEN
                         x`system
                       ELSE
                         y`system
                       ENDIF,
        scaling     := x`scaling * y`scaling,
          length_dim  := x`length_dim  + y`length_dim,
        time_dim    := x`time_dim    + y`time_dim,
        mass_dim    := x`mass_dim    + y`mass_dim,
        current_dim := x`current_dim + y`current_dim,
        temp_dim    := x`temp_dim    + y`temp_dim,
        intens_dim  := x`intens_dim  + y`intens_dim,
        angle_dim   := x`angle_dim   + y`angle_dim
       #);
```

Whether measurements have compatible systems for multiplication/division is defined by the previously discussed `system_match?` predicate. Accidental mixtures of systems of units are prevented by disallowing multiplication or division between different systems. As with addition, the `preferred_system?` predicate is required to ensure that multiplication is commutative. The multiplication rule allows any two measurements to be multiplied (e.g., force and distance), and defines the resultant measurement as having the combined dimensionality of the multiplicands. Similarly, a division rule is defined allowing two measurements to be divided

(e.g., length and time) with the result having the appropriate dimensionality (e.g., speed), and an exponentiation rule allows a measurement to be raised to a power (e.g., length squared to become area).

Conversions require their own theory and must be explicitly defined as transmutations:

```
transmutation: NONEMPTY_TYPE =
    [#
      to_factor:   {n: nzmeasurement | explicit_system?(n)},
      from_factor: {n: nzmeasurement | explicit_system?(n) AND
to_factor`system /= n`system}
     #] CONTAINING
    (#
      to_factor   := unit_measurement WITH [`system := METRIC],
      from_factor := unit_measurement WITH [`system := IMPERIAL]
     #);
```

3.4.1.2 System-Templated Library

The system-templated library is similar to the system-field library, except that instead of `system` being a component of the measurement, `measurement_systems` parameterize the theory, where `measurement_systems` is an enumeration of `{METRIC, IMPERIAL}`:

```
  measurements[(IMPORTING measurement_systems) S: system_enum]: THEORY
```

Units are defined in this library by instantiated versions of templated theories, where different systems use different scaling factors. For example the `imperial_lengths` theory is defined as:

```
  imperial_lengths: THEORY
  BEGIN

    IMPORTING measurement_systems;
    IMPORTING lengths[IMPERIAL];

    ft_to_m: real = 0.3048;

    ft: poslength = unit_length WITH [`scaling := ft_to_m];
    inch: poslength = ft WITH [`scaling := ft_to_m * 1/12];
    yard: poslength = ft WITH [`scaling := ft_to_m * 3];
    mi: poslength = ft WITH [`scaling := ft_to_m * 5280];

  END imperial_lengths
```

Within this library, the specification of the addition operation is a little simpler:

```
  +(x: measurement, y: {m: measurement | unit_match?(x, m)}):
    {m: measurement | unit_match?(x, m)} =
      x WITH [`value := x`value + y`value]
```

Note that addition is automatically commutative without a need to check for a preferred system, and the specification of the predicate `unit_match?` is also simpler:

```
unit_match?(x: measurement, y: measurement): bool =
  dimension_match?(x, y) AND
  (x`scaling = y`scaling);
```

The `dimension_match?` predicate is identical to that in the system-field library.

The multiplication operation is also simpler:

```
*(x: measurement, y: measurement): measurement =
  (#
    value       := x`value * y`value,
    scaling     := x`scaling * y`scaling,
    length_dim  := x`length_dim  + y`length_dim,
    time_dim    := x`time_dim    + y`time_dim,
    mass_dim    := x`mass_dim    + y`mass_dim,
    current_dim := x`current_dim + y`current_dim
    temp_dim    := x`temp_dim    + y`temp_dim,
    intens_dim  := x`intens_dim  + y`intens_dim,
    angle_dim   := x`angle_dim   + y`angle_dim
   #);
```

Unfortunately, with the system-templated library, it becomes possible to inadvertently combine units from different systems so that `m * ft` is valid. One *can* define a predicate to check for whether a unit is consistently scaled as a power of ten:

```
% i can be < 0
power_of_ten_measurement?(m: measurement): bool =
  EXISTS(i: int): (10^i = m`scaling);
```

Because imperial units are always defined with a scaling factor that is not a power of ten (with the exception of units that are system-agnostic such as seconds), the `power_of_ten_measurement?` predicate can be used to identify measurements that are metric. However, this predicate will miss certain situations where custom metric units have scaling factors that are *not* powers of ten. The decision to use `METRIC` as a baseline (so that `m` has a scaling factor of 1, for example) instead of `IMPERIAL` was made primarily for our preference of the metric system, but is also supported by the potential utility of the `power_of_ten_measurement?` predicate.

3.4.1.3 Library Comparison

In the system-templated library, all measurements are valid and compatible with respect to multiplication. This results in fewer TCCs and often simpler proofs. It also fails to automatically detect cases where units from different systems are multiplied or even added; however, units are defined in this library so that conversions would happen automatically, per their `scaling` field. So, while the system-field library supports rigorous analysis of units and eliminates the possibility of multiplying `m * ft`, its use often results in complex type-correctness conditions

(TCCs) and increases the difficulty of proving theorems, compared to the system-templated library. Either library can provide the foundation of the compositional analysis framework.

3.4.1.4 Discussion

While initially we considered the possibility of having domain-specific primitive types (such as longitude/latitude), experimentation has suggested that domain-specific types can better be represented using compositional combination as discussed in Section 3.4.2. Additionally, the measurement libraries can be extended with the addition of domain-specific constants, unit names (which are really just a type of constant), or even new systems of units.

3.4.2 Real-World Type Manipulation

The primitive real-world types described in Section 3.4.1 can be combined to describe any kind of measurement. The rules by which the types combine form the basis of a type theory for real-world types. As an example, a speed measurement should have the dimensions of LT^{-1}, where L refers to the length dimension and T to the time dimension. This type of manipulation relies on basic arithmetical operations, as discussed in Section 3.4.1.

In addition to measurements being combined through basic arithmetical operations, measurements can be composed into more complex objects, such as vectors, matrices, and other structures.

Vectors of measurements can either be all of the same unit (e.g., a vector describing a location in 3-space) or can contain a combination of units. An example of a vector of measurements with homogeneous units is a velocity vector:

```
mv: Measurement_Vector =
  (: 3, 4, 5 :) * (m / s);
```

With this homogeneous vector, one can determine its magnitude (13 m /s), and intuitively combines the properties of measurements discussed so far with the properties of vectors. However, consider an example of a measurement with heterogenous units:

```
mv: Measurement_Vector =
  (: 3 * m / s, 4 * Hz, 5 * N :);
```

With this heterogenous vector, there is no meaningful definition of its magnitude, but yet this vector could legitimately describe a state vector.

Rules for mathematical operations on vectors of measurements follow from the rules of operations on their components, and a theory of measurement vectors has been established to describe these rules. For addition of two vectors, the rule is simple: the vectors must be of the same size, and each element in one vector must match the units of the corresponding element in the second vector. For example, if you have a two element vector whose first element is measured in meters and whose second element is measured in centimeters, than it can only be added to another vector whose first element is measured in meters and whose second element is measured in centimeters. For dot products, the rule is a little more interesting: while the vectors elements do not need to correspond, all pair-wise products must have the same units. For example, if you have a three element vector whose elements are measured in `m`, `m/s`, and `mm`,

then it cannot be multiplied by another vector with the same units, but it *can* be multiplied by a vector whose elements are measured in `m`, `s`, and `km`:

```
mv1: Measurement_Vector =
  (: 10 * m, 0.25 * m^2 / s, 2 * mm :);
mv2: Measurement_Vector =
  (: 0.1 * m, 4 * s, 0.5 * km :);
% Cannot add m^2 + m^2/s^2 + mm^2
invalid_meas: Measurement =
  mv1 * mv1;
% Can add m^2 + m^2 + m^2
valid_area: Measurement =
  mv1 * mv2;
```

A motivating example for vectors and matrices composed of differing units is a model of a Kalman filter. Kalman filters take a state estimate vector ($\hat{\mathbf{x}}$), an actuator vector (**u**), a state transition matrix (**A**), a matrix describing the effect of the actuators on the state (**B**), an estimate of error covariance (**P**), and a process noise matrix (**Q**) to generate a new state estimate vector. Each of these vectors and matrices will typically be composed of elements with a mixture of units. To check unit consistency in a model using Kalman filters means understanding how vectors and matrices of varying units interact. In addition to the simple addition and multiplication rules covering vectors and matrices, the Kalman filters introduce additional operations to consider: transition matrices, transformation matrices, and matrix inversion.

As with vectors, the rule for addition is straight-forward: the matrices must be of the same size in both dimensions, and each element in one matrix must match the units of the corresponding element in the second matrix. For multiplication, the rules are definitely more complicated. For matrix A to be multipliable by matrix B, the following properties of the two matrices must hold.

1. The number of columns in A must equal the number of rows in B (as with regular matrices).
2. The matrices must have elements such that, when multiplied, terms being added to form the resultant matrix have the same units. This means that:

(a) For matrix A to be multipliable by *any* matrix, $units(A_{ij}) = \frac{units(A_{i1})units(A_{1j})}{units(A_{11})}$.

(b) For matrix B, it must be the case that $units(B_{ij}) = \frac{units(A_{i1})units(B_{1j})}{units(A_{11})}$.

With unitless square matrices, if matrix A can be multiplied by matrix B, then matrix B can be multiplied by matrix A. This is not necessarily true when considering units. When proving properties with unitless matrices, one can use the fact that $\sum_{j=1}^{n}\sum_{k=1}^{m} A_{ij}B_{jk}$ is identical to $\sum_{j=1}^{n} A_{ij}\sum_{k=1}^{m} B_{jk}$, but this equivalence is not valid when $B_{jk} \neq B_{jm}$ for some k and m. One important property of matrix multiplication that does still hold is that of associativity: if matrix A can multiply by the product of matrix B and matrix C, then the product of matrix A and matrix B can multiply by matrix C, and the results will be identical.

Transition matrices are matrices that when multiplied by a vector will yield a vector with the same units. Note that this is not true in general when multiplying a matrix by a vector as is the case when with **Bu**, which transforms a vector describing actuators into a vector describing states. Transition matrices must have the following properties:

1. Transition matrices must be square matrices.
2. For transition matrix **A** and state vector **x**, the units of element A_{ij} must equal the ratio of the units of x_i to the units of x_j, thus the units of the vector upon which a transition matrix operates uniquely defines the units of the transition matrix itself.
3. From (2), it follows that element A_{ij} must have inverse units of element A_{ji}.
4. From (3), it follows that diagonal elements (A_{ii}) must be dimensionless.
5. From (2), it also follows that the units of $A_{ij} \times A_{jk} = A_{ik}$.
6. From (5), it follows that the units of elements $A_{i,i+1}$, for $i<numcols(\mathbf{A})$ uniquely define the remaining elements of **A**. Without reference to **x**, the units of these elements cannot be further inferred. Thus the number of elements in the matrix that define the units for the remaining elements in the matrix is $numcols(\mathbf{A})-1$.
7. From (2), it also follows that if a transition matrix can operate on vector **x**, then the transition matrix can also operate on $\mathbf{x} \odot \mathbf{h}$, where $\odot$ denotes component-wise multiplication, and **h** is a vector with homogenous units.

Transformation matrices must have the following properties:

1. A transformation matrix transposing from a vector with length *a* to a vector with length *b* must be of size $b \times a$ (i.e,. have *b* rows and *a* columns).
2. For transformation matrix **B**, vector **u** to be transformed and vector **x** to be transformed into, the units of element B_{ij} must equal the ratio of the units of x_i to the units of u_j.
3. From (2), it follows that the ratio of units of element B_{ij} to B_{ik} must equal the ratio of the units of u_k to the ratio of the units of u_j, allowing us to determine whether a transformation matrix **B** can operate on vector **u**.
4. Also from (2), it follows that the ratio of units of element B_{ij} to B_{ki} must equal the ratio of the units of x_i to the ratio of the units of x_k, allowing us to determine whether a transformation matrix **B** can transform a vector into a type with units matching vector **x**.
5. Without reference to **u** or **x**, it also follows that a transformation matrix must have the property that $B_{ab}B_{cd} = B_{ad}B_{cb}$.

Using the rules for transition matrices and transformation matrices, we can write the `predictState` step of the Kalman filter as:

```
StateSizeSquareM: NONEMPTY_TYPE = SquareMM(StateSize)
  CONTAINING I(StateSize);
StateVector: NONEMPTY_TYPE = Measurement_Mat(StateSize, 1)
  CONTAINING (# rows := StateSize, cols := 1,
    matrix := LAMBDA(i: below(StateSize), j: below(1)):
        zero_measurement #);
BSizeMatrix: NONEMPTY_TYPE = Measurement_Mat(StateSize, ActuatorSize)
  CONTAINING (# rows := StateSize, cols := ActuatorSize,
    matrix := LAMBDA(i: below(StateSize), j: below(ActuatorSize)):
```

```
          zero_measurement#);
  A: VAR (a: StateSizeSquareM | transition_matrix?(a));
  B: VAR BSizeMatrix;
  predictState(A, (priorState: (transitions?(A))), B, (u: (transforms?(B,
priorState)))):
    (unit_match?(priorState)) =
      (A * priorState) + (B * u);
```

In the above example, `StateSize` and `ActuatorSize` are parameters to the `measurement_kalman` PVS theory. These values are used to define `StateSizeSquareM`, `StateVector`, and `BSizeMatrix` types. The `A` transition matrix and `B` transformation matrix are defined relative to these parameters. The `priorState` argument to `predictState` is defined as a vector upon which the transition matrix `A` can operate, and the `u` argument is defined as a vector upon which the transformation matrix `B` can operate such as to generate a vector with the same positional units as `priorState`.

Solving matrix inversion for matrices using units is an ongoing area of research.

3.4.2.1 Proof Automation Support

Throughout the development and application of the PVS libraries for the `measurement` type, attention has been given to supporting proof automation.

While PVS is a very powerful theorem prover, it has a reputation for requiring significant input from a human to complete proofs. Proof automation can be used to reduce or, in many cases, eliminate the need for human guidance during proofs. Several approaches are provided by PVS to enable proof automation.

1. Judgements. Judgements are pre-proven type equivalencies that can be leveraged by the type checker. When a judgment is available, it often allows the type checker to avoid issuing a proof obligation for type correctness (a Type Correctness Condition, or TCC).
2. Rewrite rules. Rewrite rules are used by PVS to automatically rewrite an expression in a different, often simpler form. When a rewrite rule is available, it often allows PVS to automatically simplify an expression, eliminating the need for human guidance.
3. Lemmas. Lemmas are pre-proven expressions that can be used in proofs to simplify an expression. While lemmas are often not automatically applied by PVS, they can be applied through proof-lite scripts, increasing the power such scripts and reducing the need for human guidance.
4. Proof-lite scripts. Proof-lite scripts are a sequence of proof commands that are applied automatically when the command-line tool `proveit` is called. These scripts can be included in PVS files as structured comments, and can use wildcard matching so that they are automatically applied to many proof obligations. Proof-lite scripts are particularly useful for helping to discharge TCCs.

All four of these approaches have been taken with the `measurement` libraries, and significantly reduce the level of human guidance required to complete proofs involving real-world types.

3.4.3 Correspondence Analysis with Retrenchment

Retrenchment provides a rigorous framework for reasoning about the transition from real-world types to machine-world types.

3.4.3.1 Background

Retrenchment is a variation of program refinement. Program refinement is an iterative method which involves adding detail to abstract specifications until they become concrete enough to be implementations. Importantly, refinement requires that the refining specification can be proven to:

1. maintain the invariants of the refined specification;
2. maintain any assertions of the refined specification;
3. maintain the correctness of the initialization;
4. not require narrowing (strengthening) of pre-conditions;
5. not require weakening of post-conditions and
6. not change the signature of any operation.

Retrenchment acknowledges that real-world constraints can cause some of these proofs to fail. In general, retrenchment is a narrowing of the pre-condition and a constrained weakening of the post-condition [47], although it can weaken any of the above requirements of refinement. Consider the following specification of an adder:

```
adder: THEORY
BEGIN
  plus(a: nat, b: nat): nat = a + b;
END adder
```

A simple specification, except that in PVS the naturals are unbounded, while in practice there will be a maximum value that can be represented. For example, one might posit the following as a refinement:

```
adder2: THEORY
BEGIN
  max_nnint32 : int = 4294967295;
  nnint32?(n: real): bool = integer?(n) AND (0 <= n) AND (n <= max_nnint32);
  nnint32 : NONEMPTY_TYPE = (nnint32?) CONTAINING 0;

  plus(a: nnint32, b: nnint32): nnint32 =
    IF (a + b > max_nnint32) THEN
      max_nnint32
    ELSE
      a + b
    ENDIF;
END adder2
```

This clearly violates the requirements of a refinement:

1. The signature of the operation was changed.
2. The revised signature narrows the pre-conditions.
3. The post-condition is weakened for the case where `a + b > max_nnint32`.

Thus, the proposed refinement is not a refinement, but a retrenchment.

In many cases, constrained equivalence between models [48] can be considered a form of retrenchment, if one considers the models to be formal specifications. For example, if one model

is the abstract OEM model and the second model is the model as implemented by the supplier, then likely the supplier's model will be a retrenched version of the OEM model, and constrained equivalence will provide a formal means of defining the rules of that retrenchment.

3.4.3.2 Application to Correspondence

As discussed in Section 3.1.2, SIAT calls for an explicit correspondence between environment phenomena that are modeled as real-world types and their representation in the system as machine-world types.

Most real-world phenomena are continuous in nature and are described using real numbers. Additionally, bounds are not always identified for real-world phenomena. For example, while there is a practical limit to displacement for a vehicle, arising from time or fuel constraints, a maximum displacement is unlikely to be identified in the problem or the requirements.

For digital computer systems, the representations of real-world phenomena are not real numbers and are never unbounded. Instead, fixed- or floating-point numbers are used and bounds are either explicitly identified or implicitly defined by the representation used. This essential discretization introduces unavoidable loss of precision, which represents a retrenchment from the real-world phenomenon. Moreover, the essential bounding introduces unavoidable inaccuracy whenever the real-world value exceed the bounds of the machine-world representation, which represents another retrenchment from the real-world phenomenon.

Additionally, system representations of real-world phenomena are driven either directly by sensor measurements or indirectly by models based on sensor measurements. Sensors introduce additional loss of precision and inaccuracy by virtue of the physical processes through which they make their measurements and by their own internal system representations, when they include digital computers or digital outputs. Sensors and digital computers also introduce latency.

Correspondence models describes the inaccuracy, loss of precision, latency, and other discrepancies between the true value of an environmental phenomenon and its system representation. These discrepancies represent a retrenchment between the requirements and the specification of the system: we cannot say that the specification is a refinement of the requirements, because the data types have changed and the data values have become imprecise. Retrenchment therefore provides a framework in which the impact of these changes can be assessed, increasing our confidence that the specification is correct.

3.4.3.3 The Role of Correspondence

Correspondence supports claims of requirement satisfaction. Using correspondence models, the retrenchment from the requirement to the design is argued to be acceptable. Then the design can be shown to be correct which, under the retrenchment, satisfies the requirement. Using the Ultra Stick example (see Section 4.2), as example argument is shown in Figure 44.

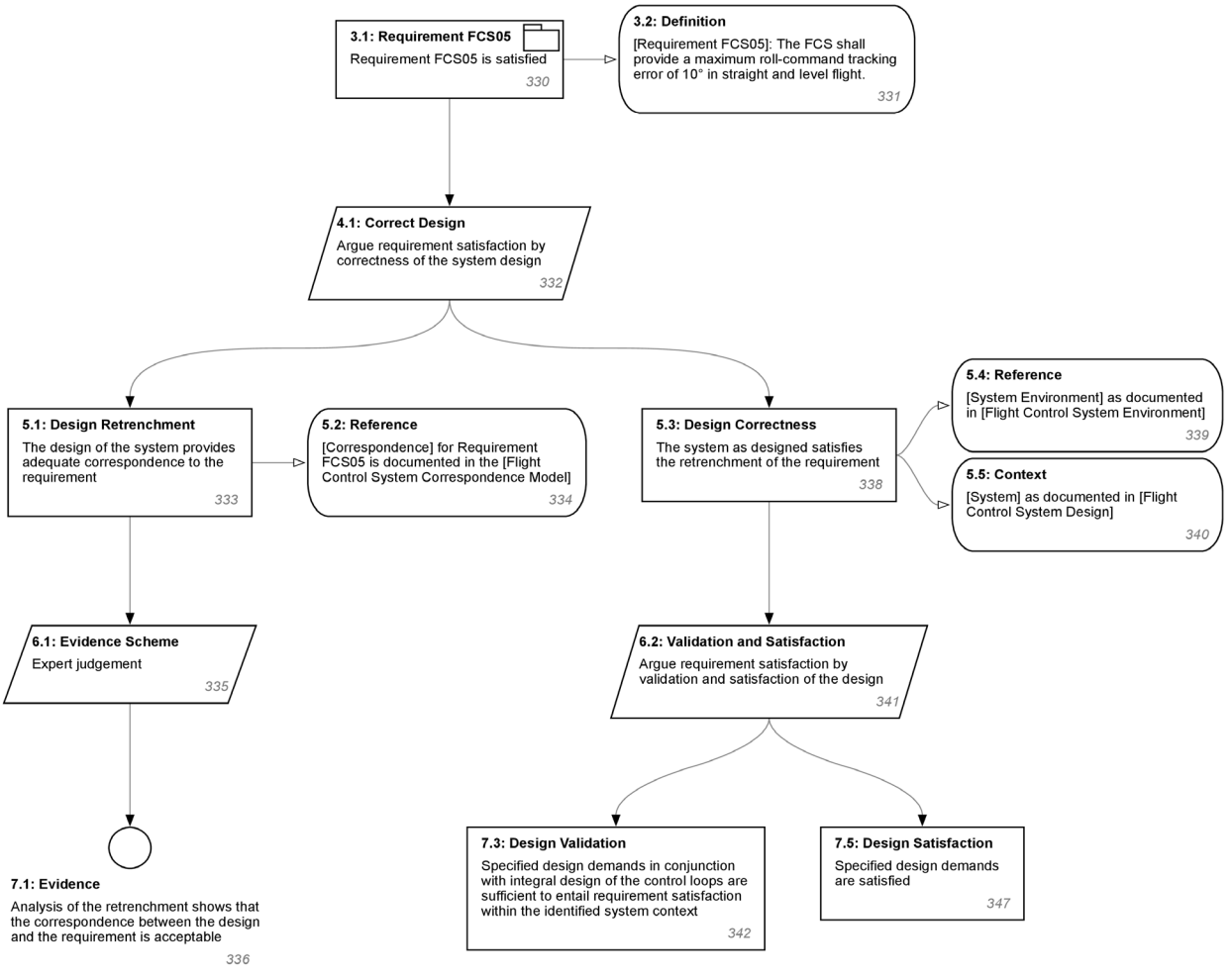


Figure 44: An Example Argument using Correspondence

3.4.3.4 Representing Correspondence

Correspondence models include four elements:

1. explication,
2. real-world semantics,
3. machine-world representations, and
4. approximation.

A correspondence model is attached to each element of an interface, to document unambiguously and formally the relationship between the machine-world element of the interface and its syntactic components and the real-world phenomenon being represented.

As an example, a simple correspondence model for the pitch estimate that is provided by the measurement subsystem in the Ultra Stick example (see Section 4.2) is shown below[11].

```
pitch: machine world correspondence {
  explication:
    Body-axis pitch angle, measured in degrees.

  real-world semantics:
    quantity        : angle
    units           : degrees
    range           : -90 .. 90

  machine-world semantics:
    representation  : 16-bit integer
    scale           : 1/100
    offset          : 0
    range           : -9000 .. 9000

  approximation:
    noise           : 0.43
    bias            : 1.20
}
```

The representation shown above is intended to be easy to read and easy to write, while also being easily machine-translatable to an appropriate formalism. We have developed a prototype translation tool that parses this format and builds an internal, intermediate representation. From this intermediate representation, we can generate a variety of formal representations. For example, the prototype tool is currently configured to output PVS, as shown below.

```
% Body-axis pitch angle, measured in degrees.
pitch: machine_world_correspondence =
  (#
    real_world_semantics :=
      (#
        quantity :=
          ANGLE,
        units :=
          DEGREES,
        range :=
          (#
            min := -90,
            max := 90
           #)
       #),
```

[11] The machine-world semantics section of this example specifies a fixed-point representation of a real number. The *scale* parameter indicates how the integer value stored in computer memory should be scaled and the *offset* parameter indicates how the resulting value should be offset to yield the interpreted value of the parameter.

```
    machine_world_semantics :=
      (#
        representation :=
          A16_BIT_INTEGER,
        scale :=
          1/100,
        offset :=
          0,
        range :=
          (#
            min := -9000,
            max := 9000
           #)
       #),
    machine_world_semantics :=
      (#
        noise :=
          0.43,
        bias :=
          1.20
       #)
   #)
```

As a second example, also drawn from the Ultra Stick example, consider the correspondence model for pitch rate, shown below.

```
pitch rate: machine world correspondence {
  explication:
    Body-axis pitch rate, measured in degrees per second.

  real-world semantics:
    quantity        : angle rate
    units           : degrees/s
    range           : -245 .. 245

  machine-world semantics:
    representation  : 16-bit integer
    scale           : 1/100
    offset          : 0
    range           : -24500 .. 24500

  approximation:
    noise           : 0.24
    bias            : 1.80
}
```

The prototype translation tool generates the following PVS, for this correspondence model.

```
% Body-axis pitch rate, measured in degrees per second.
pitch_rate: machine_world_correspondence =
  (#
    real_world_semantics :=
      (#
        quantity :=
```

```
        (#
          value := angle,
          unit  := RATE
         #),
       units :=
         DEGREES_PER_S,
       range :=
         (#
          min := -245,
          max := 245
         #)
     #),
  machine_world_semantics :=
    (#
      representation :=
        A16_BIT_INTEGER,
      scale :=
        1/100,
      offset :=
        0,
      range :=
        (#
          min := -24500,
          max := 24500
         #)
     #),
  machine_world_semantics :=
    (#
      noise :=
        0.24,
      bias :=
        1.80
     #)
#)
```

3.4.4 Contract Analysis

The use of real-world types and correspondence models to document the relationship between elements of a component's interface and real-world phenomena enables more in-depth analysis of contracts than would be possible using only machine-world types.

A traditional interface for a component includes the machine-world type for each element of the interface and, typically, a meaningful identifier for the element. For example, an interface might include "velocity" and "float", indicating that there is an element that reports on velocity and that it is represented in the machine as a floating-point number.

Often, additional information about the element of the interface is available in documentation. The documentation might specify, for instance, that the velocity is to be interpreted as meters per second and is, moreover, constrained to fall between zero and 100 meters per second. Unfortunately, this kind of documentation is typically informal and is not available to automated analysis tools.

By incorporating real-world types into the interface, we can formalize aspects of the entity being represented. By incorporating correspondence models, additional critical information can be

included. Together, these richer formalized semantics allow stronger checking of contractual compatibility, increasing confidence in composition.

4 RESULTS AND DISCUSSION

This section presents the result of applying system-interface abstraction technology, our system-of-systems enabling technology. Section 4.1 describes the use of system-interface abstraction technology and, in particular, the patterns for arguing successful development as a novel approach to responding to a request for proposals. Section 4.2 describes the application of system-interface abstraction technology to a hypothetical small UAS based on the Ultra Stick platform. Section 4.3 describes the application of domain-argument recovery, a reference mechanic for system-interface abstraction technology, to relevant standards.

4.1 Cooling Tanks Example Problem

This section presents the development of an illustrative example of SIAT by application to the cooling tank challenge problem. Specifically, we use the cooling tank challenge problem provided by RQQA as a hypothetical Request For Proposals (RFP).

The goal of this effort is to demonstrate an experimental response to the RFP in the form of an initial assurance case for a cooling tank system. The assurance case provides RFP response reviewers with the direction that will be taken to develop the cooling tank system and to provide adequate assurance that the developed system provides appropriate properties and behaviors. The case is presented to illustrate how iterative and modular development and assurance will proceed if the RFP response were accepted.

The intent is to develop an argument-based rationale for a development approach with a clear focus on assurance and "successful development". The emphasis of the RFP response is the application of SIAT concepts, specifically identification, separation, and documentation the problem, its solution, and its context, including both regulatory and environmental context. The RFP response therefore serves two primary purposes:

1. the assurance case illustrates a potential new acquisition approach that would require or prefer responses delivered with initial assurance arguments, and
2. the documentation within the response provides a detailed explanation of the application of SIAT concepts and mechanics that have been previously described.

4.1.1 Experiment Overview

The substance of this effort is an experimental RFP response that is developed and documented entirely within an initial assurance case. Typical RFP response sections and documentation is provided along with initial arguments in an evolving assurance case structure. The mockup organization and case structure will be used to explain the application of the technologies developed in this effort. The argument does not justify why the proposal should be accepted directly, but rather provides a high-level outline of how successful development would be justified if the proposal were accepted. The arguments presented will therefore provide a general direction/structure but will require further development which is assumed to take place after the hypothetical response is accepted. The mockup argument demonstrates how the proposed system solution will be justified as successfully developed and will illustrate/describe the general mechanics by which the SIAT technologies are applied to complete the argument.

The combination of the RFP response documentation and argument provides response reviewers with novel view of how the development of the proposed system will be shown to be "adequate". The combination of argument with documentation itself forms a "meta-argument": an argument

that the SIAT approach (which itself includes development of an argument) will be successful. In principle, the meta-argument could be expressed as an assurance case; however, for simplicity of the example, we do not explore this approach.

The primary input to this effort is the cooling tank CONOPS (Appendix Appendix B). Other inputs include previous requirements and design documentation that has been collaboratively refined and reviewed with DCi and RQQA. These inputs are modified as necessary to better develop a RFP response and to better apply the SIAT technology. Previous requirements and design will serve as an initial prototype for discussion in the hypothetical RFP and provide a direction for further development/refinement if the hypothetical RFP response were accepted.

A typical RFP response must address concerns that the proposed effort will be completed within appropriate time and cost, and if the response will meet the proposing company's business goals. These concerns can also be argued in the case. We will point to where these arguments could be made within the existing argument infrastructure and provide a discussion, but we will largely consider these arguments out of scope for the example.

Since the complete RFP response is an assurance case, it is provided in a separate document artifact. To provide a high-level view of the RFP response, the executive summary is copied below. Note that the response, as an experiment, speaks to hypothetical organizations and facilitates. Hypothetical entities are not important for this experiment, but necessary for documentation. These entities are documented within curly braces as placeholders.

4.1.2 Executive Summary

{Dependable Computing} proposes to develop a cooling tank system to be incorporated into the {industrial facility} {system requiring cooling} system. The developed cooling tank system will consist of a modular design to facilitate redesign, upgrades, replacements, etc. of components in the future. The design goal is to not only meet the stakeholder needs as outlined in the RFP cooling tank CONOPS (Appendix Appendix B), but also to provide useful modularity in support of practical and cost effective design/development, lifetime maintenance and system evolution. Modular designs promote these goals by managing complexity, enabling parallel work, and accommodating future uncertainty (changes to the system of its lifetime). More specifically, modularity promotes:

1. simplifying complex and large designs by providing high-level abstractions,
2. minimizing the impact of changes through information hiding and low coupling,
3. independent and coordinated design and development through well-defined interfaces, and
4. reuse (both design for reuse and design with reuse) through low coupling and high cohesion which in turn decreases engineering and certification costs.

This proposal provides a prototype cooling tank system design based on the initial CONOPS (Appendix Appendix B) and high-level requirements specified in the RFP. The proposed design is used to illustrate the organization and development processes to be used by {Dependable Computing} and a basis for further system development. While the architecture of the cooling tank system will be designed by {Dependable Computing}, the selection and final delineation of component boundaries (modules) is subject to approval and review by {industrial facility} to best identify components likely to be changed within the context of {industrial facility} and {system requiring cooling}. The proposed delineation of components/modules presented in here is based on {expert knowledge or standard} to illustrate the design approach and can be easily altered to best meet the needs of {industrial facility}.

A key aspect of the proposed cooling tank system development is the co-development of a rigorous *assurance case*. The assurance case consists of a structured argument, supported by a body of evidence, that provides a compelling, comprehensible and valid case that a system is acceptable for a given application in a given environment. In particular, we adopt an argument structure where the top-level assurance goal is that the system development is "*successful*". The success argument's structure and content as well as the general development of cooling tanks system is based upon the System Interface Abstraction Technology (SIAT) developed by Dependable Computing. Successful system development is defined in by:

1. adequate identification of the problem, the context in which the problem exists, and the problem solution (requirements) within the defined context, and
2. adequate assessment of the solution (the designed and developed system) to satisfy requirements, provide necessary levels of safety and security, comply with relevant regulations and standards, etc.

The terms "success" and "adequacy" are used generally throughout the argument to indicate that the associated activity or product justifiably meets all expectations of the stakeholders. Stakeholders provide precise definitions of adequacy for individual expectations.

The proposed cooling tank system assurance case is modularized to directly reflect modularized system components: i.e., the delineation of the argument modules mirrors the delineation of system component modules. Assurance arguments are developed for each system component, and composed to derive a comprehensive cooling tank system assurance case. The close tracking of assurance case development and structure to the design/development of associated system components facilitates early and often assessment of the developed system through the iterative deepening of the system structure throughout the development process. During each iteration, the assurance case is used to incrementally assess the success of the component and the composed system as a whole. A key benefit of the coupled modular system and assurance case design is therefore the "early and often" assurance-driven development of the system through progressive development and composition of the modularized components and their associated modularized assurance cases.

In addition to providing benefits during system development, the modular system and assurance case design are intended to facilitate incremental updates and certification to the cooling system, thereby reducing maintenance costs over the lifetime of the deployed system. Incremental certification is an open and fundamental challenge requiring further study. This proposal will discuss SIAT mechanics by which incremental certification may be achieved under the proposed assurance case design. In the proposed effort, {Dependable Computing} will be responsible for the development of cases for modules with the intent of increasing the acceptance and technology of composable arguments to eventually allow for independent composition of components (both bespoke and reused) and associated assurance arguments in the future. {Dependable Computing} can optionally provide support for incremental certification for the cooling system as the need arises once the system is deployed and as the general discipline of incremental certification matures.

Development of the cooling tank system will be largely self contained, based itself on the principles of modularity: conceptually the cooling tank system is a component within a larger system of systems. {Dependable Computing} will work with {industrial facility} to incorporate the cooling system and the associated case into a larger system of systems including incorporation into any certification/approval constructs used by {industrial facility}.

4.1.3 Response Prototype Conclusions

The scope of the RFP response example provided a direction for a novel approach to developing RFP responses. Specifically, the example illustrates alignment of the principles of successful development described by SIAT as sections and tasks of the response. The illustration is limited in scope to simplify the example, e.g., the example focuses on application of SIAT at the highest level of development of the cooling tank system. Despite this simplification, the general approach taken for this first tier of modular decomposition provides a direction that can be repeated recursively.

Additionally, the illustration is necessarily limited since prior to RFP acceptance, key details of the system are not concretized. Because of the lack of development information during the response, RFP offerers are limited to hypothesizing directions for the argument by hypothesizing requirements, specification and detailed design. This approach is therefore more amenable to prototype-oriented development approaches. The use of the argument illustrates to the RFP program manager that the offerer is taking assurance into account early and often, but, because of the lack of development information, the substance of the argument should not be the primary focus for RFP acceptance. The RFP response can only illustrate the activities that would be performed and artifacts that would be produced if the response is accepted. RFP program managers must therefore accept and have a detailed understanding of the SIAT approach in order to assess the response. If both RFP program manages and RFP offerers agree upon the principles of SIAT, responses can be somewhat standardized to include details about key SIAT artifacts and how the argument could be developed, potentially reducing the effort on the part of program managers. Furthermore, by accepting a general assurance/development methodology like SIAT, respones do not need to include detailed descriptions of these methods, thereby reducing the size of the response.

A difficulty encountered in this activity and throughout general use of the cooling tanks challenge problem was that detailed stakeholder/domain expert knowledge was often necessary but unavailable due to the example being academic in nature. As a consequence, we as non-domain experts for this system often struggled with developing and refining meaningful problem descriptions, context, requirements, architectures, etc. The degree of detail necessary to fully explore the uses of SIAT, whether for this RFP response or in general, will require a heavily refined example (if not a real-world example) with domain experts providing development artifacts and frequent feedback. In this manner, we can focus on the application of SIAT without having to solve engineering challenges outside the domain of assurance.

A related challenge to the lack of expert knowledge and a highly-refined example is deciding when to terminate the decomposition of components. For simplicity of the example, the tasks of the RFP response note that further decomposition and refinement with stakeholders will be performed, but the elaborated system was limited to the first tier of decomposition. Within an actual RFP response, offerers could continue a decomposition either until the granularity of decomposition is prevented or not appropriate because of the lack of development information during RFP response, or the offerer has identified interfaces where components the offerer intends to rely upon third party or previously developed components.

Separate from the application of SIAT, arguments could serve as a fundamental role in RFP responses in the future. As previously suggested, an RFP response could be structured as an argument with a top level claim that the response should be accepted. By proposing a SIAT approach in our illustration, a meta-argument is implied and assessed by program managers

through the evaluation of how SIAT is applied. Without an agreed upon standard development approach, the meta-argument needs to be more explicit, and could potentially be structured as a GSN argument. This approach is related to the concept of success arguments from Assurance-Based Development (ABD) [25] [10] [24] and is an interesting direction for future research in argument-based acquisition approaches.

4.2 Ultra Stick UAS Example Problem

The applicability and utility of system-interface abstraction technology rests on critical assumptions:

1. complex systems and systems of systems of interest are built from components whose behaviors can be described by simple abstractions; and
2. abstractions of component behaviors compose to describe relevant properties about the complete, closed-loop system or system of systems.

Computer software is typically built with an eye towards this kind of modularity. Components expose behaviors through narrowly defined and rigorously documented interfaces. Compositions of components can be analyzed to establish useful properties about the software as a whole.

Mechanical systems share similar modularity. Mechanical components necessarily interact through well-defined physical interfaces. Behaviors of mechanical components can be analyzed to establish useful properties about the mechanical system as a whole.

Sometimes, however, modularity is not employed in the design of complex systems. The benefits provided by modularity — reuse, analysis, incremental or compositional reasoning — come at a certain price. It is often not possible to both design a system with good modularity and also provide an optimal or nearly optimal solution to the problem.

Many cyber-physical systems demand a degree of optimality that threatens modularity. This is particularly true of high-performance control systems. In all but the most trivial of applications, a high-performance inner-loop control system must have many details about the dynamics of the plant, the state estimate, and the effectors to achieve adequate performance. An analysis-appropriate abstraction of any of these system components limits the degree to which the controller can be tuned, reducing its performance. Similarly, a mission planner seeking to optimize criteria such as time, energy, distance, or exposure must have many details about trajectory planning and vehicle dynamics. Abstraction of these details may result in an apparently optimal ordering of objectives that, when actual trajectories and dynamics are considered, is far from optimal.

Additionally, design for modularity has an impact on the design process. For relatively simple systems that can be readily understood by a small team of engineers, design for modularity imposes an extra burden during design. These engineers must identify components, establish abstraction-based interfaces, and design with these interfaces in mind. Since they do not *need* to do this, it increases development cost. More complex systems are less readily understood by a small team of engineers. In this case, design for modularity assists the development process by enabling engineers to break the problem into manageable pieces.

The prevalence of control in cyber-physical systems raises the following research question:

> Can system-interface abstraction technology be successfully applied to control systems?

This question is broad. Rather than trivially answer it by pointing at a very simple control system, we sought an interesting and potentially challenging example from control with which to demonstrate the applicability and usability of the technology as well as some limitations of its use.

Ultimately, we settled on the measurement subsystem for an inner-loop flight control system for a fixed-wing UAS as the core of our example problem. The measurement subsystem incorporates physical sensors and a state estimator that provides a best estimate of relevant control states based on both a model of vehicle dynamics as well as up-to-date sensor measurements. Using this example, we were able to begin to answer the question in the affirmative: system-interface abstraction technology *is* applicable to control systems.

That answer is not without its caveats, however. As noted above, the desire for optimality in control means that controls engineers are not often prepared to think about control systems with reusable components in mind. Moreover, even when modularity and component reuse is sought, there remains a dismaying degree of tight coupling amongst components.

Additionally, many aspects of system-interface abstraction technology rely heavily on domain-specific knowledge and domain-specific argumentation. The development of context compatibility and detailed evidence for requirements satisfaction by proposed system designs, for example, depend upon domain knowledge. While the example begins to answer the question of the applicability of system-interface abstraction technology to control systems, it cannot fully answer that question.

With these caveats in mind, we support our affirmative answer by:

1. Identifying interfaces based on simplifying abstractions for components of the architecture that make up the selected flight-control system — meeting assumption 1.

Starting with the problem statement for the UAS, we identify partial requirements and high-level elements of the architecture for the UAS, the air vehicle, the flight control system and finally the measurement subsystem.

2. Demonstrating compositional reasoning for useful properties of the flight control system based upon the simplifying abstractions — meeting assumption 2.

Using the interfaces identified for the measurement subsystem, we prototype selection and replacement of measurement subsystems — joint replacement of sensors and estimator. The interfaces ensure that both measurement subsystems are appropriate and enable reasoning about the effects of replacement.

Additionally, we validate the compositional reasoning at the level of the closed-loop system. With each measurement system, we first consider whether or not the interfaces are satisfied and what impact satisfaction or failure to satisfy the interface should have on composition. Then, we analyze the complete system to determine if the compositional reasoning yielded acceptably correct conclusions about the rate of successful mission completion.

4.2.1 Scope

UAS represent complex systems of systems. Design and development of artifacts for a complete UAS was therefore out of scope for this effort. We considered subsystems and components typical of a UAS to identify one component that would be a good target of study for the example.

The measurement subsystem of the flight-control system is a good target for the application of compositional reasoning. Sensors represent naturally replaceable components for which requirements are readily identified. Additionally, the inclusion of an estimator in the measurement subsystem allows issues of sibling compatibility to be explored.

We restrict the example to a vertical slice of the UAS that enables us to quickly and efficiently identify the requirements on the measurement subsystem. At each level of system decomposition, we focus on those requirements and design decisions that will ultimately impact the requirements on the measurement subsystem. The result is a deep but narrow set of artifacts related to each level of UAS development.

The identification of requirements for the measurement subsystem is the result of analyzing the impact of noise and bias on all of the measurements and observations upon which the control loops depend. Ideally, the tradeoff space associated with these signals would be fully explored, allowing robust and complete understanding of the impact on the system of changing the noise or bias on any subset of signals by any amount.

For this effort, the development of such a complete understanding of the tradeoff space was impractical. As such, only the impact of changing the noise and bias of individual signals and then changing the noise and bias of all of the signals was explored. This means that there is some loss of precision in the contracts and serves to illustrate the kind of tradeoffs that may be required to apply compositional reasoning.

4.2.2 Design Philosophy

A challenge in applying system-interface abstraction technology to control systems is the degree of dependency amongst the components of the control system. The components are so interrelated that there are a variety of ways to think about breaking up the control system. For example, consider the following breakdown:

- Measurement;
- Control;
- Actuation.

Within these three broad categories, there are both hardware an software components. For example, there are:

- *sensors* — hardware/software components that take measurements of the state of the vehicle;
- *estimator* - software component that converts measurements to state estimates;
- *control law* - software component that computes control outputs based on state estimates and commands to track;
- *computing platform* - hardware component that executes that control law and the estimator software;
- *servos* - hardware/software components that manipulate surfaces on the vehicle.

These components are not comprehensive, but are broadly representative of the elements of the control system.

The flight-control system does not exist as an isolated system: it is a component of a larger system. In the challenge problem, the flight-control system is a component of the Ultra Stick air vehicle. The air vehicle is, in turn, a component of the Ultra Stick UAS. For simplicity, we assume that the other elements of these systems have already been developed.

The interdependence amongst components of the control system allow for significant flexibility in defining the architecture. Moreover, there are a variety of ways in which each component can be categorized and grouped in the architecture. Surfaces, for instance, are part of the airframe, so they could be viewed as components of the airframe. Their role, however, suggests that they are components of the control system.

From the perspective of system-interface abstraction technology and compositional reasoning and assessment, the most productive architectural view is one in which *design authority* is captured. In this view, components are grouped and organized hierarchically to reflect the degree to which their design is dependent on the design of another component. The design-authority view shows the flow of design decisions from component to component and captures a partial order in which design decisions are made.

In the design of a flight-control system, there is flexibility in the order in which design decisions can be made, resulting in many different design-authority views. Commonly, sensors, servos and surfaces are already chosen for a particular vehicle. In this scenario, the design problem is the design of a suitable controller. According to the needs of the controller, a suitable estimator is designed. The corresponding design-authority view is shown in Figure 45.

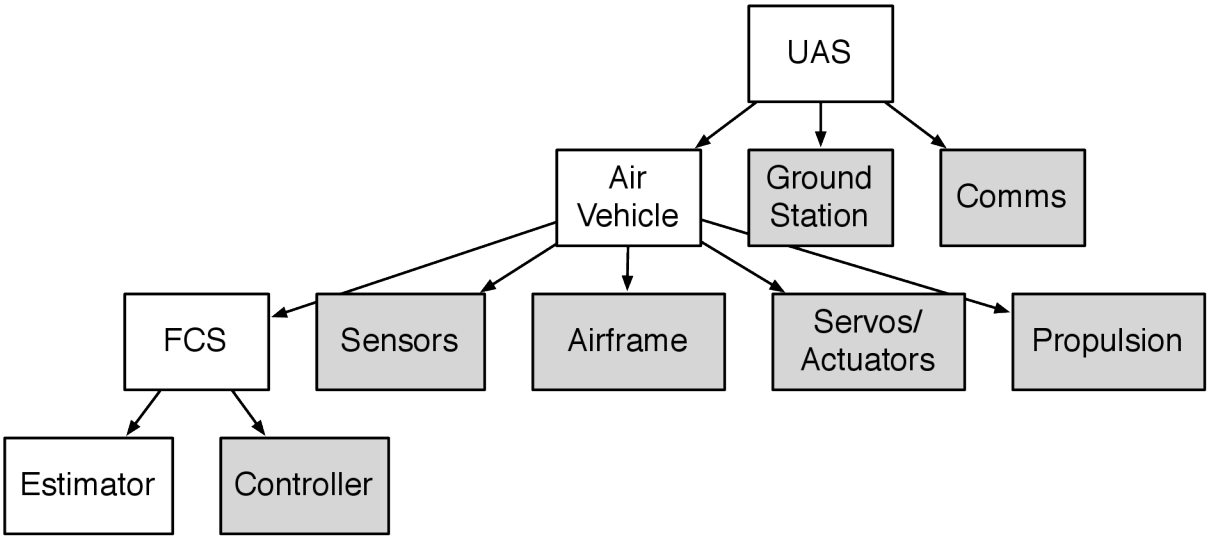


Figure 45: Possible Design Authority View for a UAS

Alternatively, only the servos and surfaces may have been chosen. In this scenario, the design problem is the design of a suitable controller, where suitable sensors and a suitable estimator may be selected according to the needs of the controller. The corresponding design-authority view is shown in Figure 46.

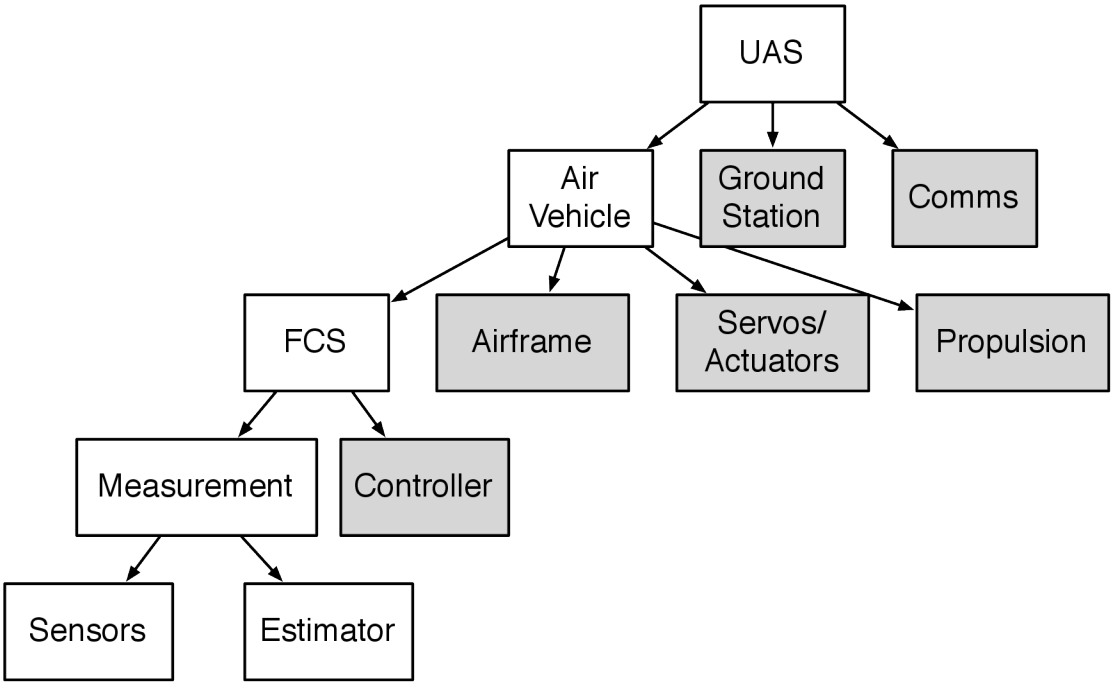


Figure 46: Another Design Authority View for a UAS

In the most extreme case, for example when a brand-new high-performance aircraft is designed, complete flexibility may be possible, allowing a true codesign of the aircraft, surfaces, actuators, sensors, and controller. The order in which design decisions are made is not fixed, in this case, allowing engineers complete flexibility in exploring the design space. Nevertheless, design decisions will be made in *some* order, as experiments suggest optimal designs to solve elements of the aircraft's problem description. Typically, due to the mutability of software and its lack of manufacturing cost, design decisions related to the hardware will be made first, starting with the airframe and working towards surfaces, servos and sensors.

For the challenge problem, we assume that the airframe, surfaces and servos have been chosen. We furthermore assume that the dynamics of the servos are sufficiently fast as to be ignored in the design of the controller. As a result of these assumptions, the full vehicle dynamics are available as context during the design of the controller. This corresponds to the design-authority view shown in Figure 46.

In this architecture, we have chosen to view the control loops as integral to the flight-control system. While they are separate entities that are likely to be developed one-at-a-time, they are unlikely to share abstraction-based interfaces that allow them to be developed in parallel and without access to the full details of inner loops. For example, the outer-loop controller is likely to require full knowledge of the dynamics of the plant and the inner-loop controller. Viewing the control loops as integral to the flight-control system is consistent with the observations made when considering the multi-level control substitution challenge problem, discussed above.

The measurement subsystem, in both the design authority view shown in Figure 45 and the design authority view shown in Figure 46, acts like a *façade* for the composition of the sensor and the estimator. It is not a true component in the sense that it does not have local or internal functionality. Instead, it abstracts the decision to include an estimator and hides details of the

sensors and the estimator from the rest of the system. A measurement subsystem could be created that was only a sensor suite, provided that full state feedback was available and sensors of sufficiently high quality were available. More likely, however, an estimator would be required to both estimate state elements not provided by direct measurement as well as filter sensor measurements for the control system.

In this example, we use the latter approach, and explore two measurement subsystems, both of which include an estimator.

4.2.3 Process

4.2.3.1 Ultra Stick UAS

Development of the Ultra Stick UAS starts at the UAS level with consideration of the problem identified by the customer.

A system is needed to image an operational area.

Image quality is of particular concern for this system. To ensure adequate image quality, images taken when the camera pitch angle is greater than 10° or roll angle is greater than 10° must be rejected. To ensure adequate resolution, pictures must not be taken above an altitude of 160 meters.

The problem statement is incomplete, but, following the discussion in Section 4.2.1, provides sufficient detail to allow development to proceed through identification of requirements on the measurement subsystem.

From this problem statement, a set of partial, high-level requirements can be identified. These requirements include:

1. The system shall overfly the operational area.
2. The system shall take images of the operational area.
3. The system shall assemble images into a mosaic of the operational area.
4. The system shall ensure that there are no gaps larger than 1 m2 of the operational area in the mosaic.
5. The system shall discard images captured when the roll angle of the camera exceeds 10°.
6. The system shall discard images captured when the pitch angle of the camera exceeds 10°.
7. The system shall discard images captured when the distance between camera and ground is greater than 160 m.

These requirements focus on considerations of the UAS as a whole and say as little as is practicable about the design of the system. As such, the focus in these requirements is on the positioning of the camera, rather than a forward reference to the use of an aircraft.

The decision to use an aircraft is part of the design, and is introduced during the development of the architecture. The major architectural components of the UAS are:

- The air vehicle,
- The ground station, and
- The communications system.

4.2.3.2 Air Vehicle

The air vehicle carries the camera and is responsible for satisfying all of the requirements described above. While the ground station and communications system are critical components of the UAS, they do not play a role in establishing the requirements for the measurement system, and are therefore out of scope.

To satisfy the design demands that are imposed on it, a partial set of air vehicle requirements have been identified. These requirements include:

1. The air vehicle shall overfly the operational area.
2. The air vehicle shall take images of the operational area.
3. The air vehicle shall ensure that at least 90% of the operational area is covered by the images.
4. The air vehicle shall capture images with a maximum camera roll angle of 10° relative to the ground.
5. The air vehicle shall capture images with a maximum camera pitch angle of 10° relative to the ground.
6. The air vehicle shall capture images with a maximum distance between camera and ground of 160 m.

These requirements translate system demands in terms of the behaviors of the air vehicle. The design of the air vehicle satisfies these requirements by delegating them to components through architectural demands. The components of the air vehicle include:

- The autopilot/flight control system,
- The airframe,
- The servos,
- Propulsion, and
- The mission sensor.

The design makes critical decisions that influence demands on components. For example, for the mission sensor:

- Camera is fixed at the air vehicle's center of gravity and does not gimbal,
- 90° field of view,
- Image capture rate of 0.2 Hz.

Based on these decisions, several design decisions for the flight control system are made, including:

- Ladder-search flight pattern,
- 10 meter overlap,
- Groundspeed limit of 25 m/s,
- Height-above-ground-level limit of 160 m,
- Roll limit of 10°, and
- Pitch limit of 10°.

4.2.3.3 Flight Control System

To satisfy the design demands that are imposed on it, a partial set of requirements for the flight control system have been identified.

Based on the top level mission requirements, upper bounds on allowable flight control system performance can be established directly. These are intended as bounds on performance under worst-case conditions. Because of the potential for interactions among multiple error sources, tighter bounds may ultimately be required to achieve desired mission success rates.

Requirements include:

1. The FCS shall provide a maximum cross-track error of 10 m in straight and level flight.
2. The FCS shall provide a maximum altitude error of 10 m in straight and level flight.
3. The FCS shall provide a maximum airspeed tracking error of 5 m/s in straight and level flight.
4. The FCS shall provide a maximum roll-command tracking error of 10 deg in straight and level flight.
5. The FCS shall provide a maximum pitch-command tracking error of 10 deg in straight and level flight.

Other requirements for the flight control system can readily be imagined, but do not directly impact the identification of requirements for the measurement subsystem.

The design of the flight control system could decompose each of the control loops into a separate component. However, the integrated nature of the control loops makes compositional reasoning difficult. Instead, we leave the control loops as an integral part of the flight control system, and identify the measurement subsystem as a component.

4.2.3.4 Measurement Subsystem

The requirements on the measurement system include detailed requirements about the maximum allowable error in terms of both standard deviation on noise and bias.

Table 1: Measurement Subsystem Requirements — Individual Signal Limits

Output	Unit	Noise STD	Bias
Roll	deg	12.8	$\pm$2.3
Pitch	deg	3.6	$\pm$10.0
Yaw	deg	3.6	$\pm$4.6
Roll rate	deg/sec	2.0	$\pm$15.0
Pitch rate	deg/sec	2.0	$\pm$15.0
Yaw rate	deg/sec	2.0	$\pm$15.0
Air speed	m/s	5.0	$\pm$5.0
Altitude	m	3.5	$\pm$5.0
GPS Position	m	8.0	$\pm$10.0

These values are chosen because each maximum noise and bias value has approximately the same impact on overall system performance. The values thus reflect, to a certain degree, the sensitivity of the system to errors on each signal.

Each value in the table is acceptable to the system when it occurs in isolation. For example, the system can achieve mission success when the standard deviation of noise for the roll input is

12.8, or if the bias on pitch is ±10 degrees. The system cannot, however, tolerate all of these maximum errors occurring at the same time.

Rather than attempting to explore the tradeoff space fully, the maximum values are uniformly scaled in noise and bias. A mission success rate of 90% is possible with a bias scale factor of ±0.12 and a simultaneous noise scale factor of 0.12.

This results in the following interface:

Table 2: Measurement Subsystem Requirements

Output	Unit	Noise STD	Bias
Roll	deg	1.54	±0.28
Pitch	deg	0.43	±1.20
Yaw	deg	0.43	±0.55
Roll rate	deg/sec	0.24	±1.80
Pitch rate	deg/sec	0.24	±1.80
Yaw rate	deg/sec	0.24	±1.80
Air speed	m/s	0.60	±0.60
Altitude	m	0.42	±0.60
GPS Position	m	0.96	±1.20

The design of the measurement subsystem satisfies these requirements with two components: the sensors and the estimator. The sensors sample environmental phenomena and generate a measurement. The estimator combines measurements with a model of vehicle and control dynamics to filter measurements and generate estimates for phenomena not directly measured.

For this example, two measurement subsystems were developed, allowing us to explore modularity and component replacement. The first measurement subsystem, discussed under Kalman Filter with Full State Feedback, below, assumed sensor measurements providing full state feedback coupled with a traditional, linear Kalman filter. The second measurement subsystem, discussed under Unscented Kalman Filter with Partial State Feedback, below, assumed sensor measurements not providing full state feedback coupled with an unscented Kalman filter.

4.2.3.5 Kalman Filter with Full State Feedback

Initial development of the measurement subsystem focused on a prototype based on a Kalman filter with sensor measurements providing full state feedback. The Kalman filter is an excellent candidate for an initial prototype, as it is easy to implement. The Kalman filter assumes that it is estimating the state of a linear stochastic process with independent, white, normally distributed process and measurement noise having known covariances. When this assumption is met, the filter returns a zero-bias estimate of state with minimum *a posteriori* covariance.

The air vehicle, however, is not a linear system. Since the assumptions of the estimator are violated by the system, we expect that its guarantees will not hold.

Simulation of the air vehicle with a simple inner-loop controller, simple sensor models, and the estimator revealed a significant bias in several elements of state — including pitch attitude. The bias was significant enough to indicate that the nonlinearity was not a minor consideration for the measurement system.

The bias was indicative of a contextual mismatch between the Kalman filter — and hence the measurement system — and the air vehicle, which is recursively a part of the context for the measurement system by way of the flight control system (see Figure 46). Compositional reasoning would lead us to conclude that attempting to use this measurement system would lead to system failures, as a result of the contextual mismatch.

To validate this conclusion of compositional reasoning, the team ran several simulations of the UAV flying a simple racetrack pattern using a waypoint-guidance algorithm based on Dubins curves and an outer-loop controller providing altitude, heading angle, and airspeed tracking. Input to the Kalman filter was provided from the simulation's truth model, without additional noise or bias. This setup enabled the team to characterize the impact of bias introduced by the Kalman filter as a result of the violated assumption of process linearity.

The team elected to validate the integration failure of the Kalman filter through the outer-loop controller because of the possibility that, as a UAV, the performance of the Ultra Stick at the outer loop is more important than the performance of the Ultra Stick at the inner loop. Given the mission described in the problem statement for the Ultra Stick UAS, it is very unlikely that the Ultra Stick would be flown by directly commanding pitch attitude. Instead, the Ultra Stick will be flown through a loop providing waypoint guidance that, in turn, will feed altitude commands to the outer loop.

The outer loop tracks a reference altitude by commanding changes to pitch attitude in response to differences between the altitude estimate and the reference command. The control loop includes an integrator, which, in the presence of a bias on pitch estimate, quickly winds up. The outer loop then commands the inner loop to track a pitch attitude that matches the biased pitch estimate. As a result, the outer loop effectively rejects the bias. For longitudinal control, the biased pitch estimate would therefore not cause an integration failure. Thus, while compositional reasoning would lead us to conclude that the measurement system is incompatible with the Ultra Stick UAS, we did not observe mission failure when we validated the conclusion.

This result seems surprising, but must be understood in the context of two additional observations:

1. Contextual incompatibility need not guarantee a failure. The impact of contextual incompatibility is impossible to predict, using compositional reasoning. Because context represents a global consideration, the impact of contextual incompatibility can only be assessed globally. When contextual incompatibility is found to be acceptable, a goal must be introduced in the argument at the highest level of system decomposition claiming the acceptability and supporting it with evidence from, for example, testing. Effectively, acceptance of contextual incompatibility becomes a question of system-wide compatibility (see Section 3.3.10). This means that should any other element of the system change at any level of the design decomposition, any previously acceptable contextual incompatibility would have to be reviewed. This essential brittleness of "acceptable" contextual incompatibility should be enough for it to be deemed unacceptable by systems engineers.

2. Validation was partial, not complete. While the team is confident in the validation results discussed above, not all flight conditions were explored during validation. It is possible, therefore, that there are flight conditions where the contextual incompatibility would still cause the system to fail to complete its mission.

Ultimately, while the estimator was found to be acceptable for the Ultra Stick UAS in spite of the contextual incompatibility, the assumption of full state feedback was found to be unacceptable. The kinds of sensors typically available for a small UAS like the Ultra Stick do not provide full state feedback.

4.2.3.6 Unscented Kalman Filter with Partial State Feedback

To address the incorrect assumption that sensors providing full state feedback would be available, a second measurement subsystem was developed. This measurement subsystem is based upon the sensors that are provided by the Pixhawk autopilot. These sensors represent partial state feedback, and are presented in Table 3, below.

Table 3: Pixhawk Sensor Characteristics

Sensor	Model	Range	Sensitivity	Noise Density	RMS Noise	Sample Rate (Hz)
Invensense MPU 6000 (primary)	3-axis accelerometer	$\pm$16 g	2048 LSB/g	400 $\mu g/\sqrt{Hz}$	5012 μg	100
	3-axis gyroscope	$\pm$2000 deg/s	16.4 LSB/(deg/s)	0.005 deg/s/$\sqrt{Hz}$	0.0626 deg/s	100
Gyroscope	ST Micro L3GD20H 16 bit gyroscope	$\pm$245 deg/s	8.75 (mdeg/s) / digit	0.011 deg/s/$\sqrt{Hz}$	0.1378 deg/s	100
Accelerometer / magnetometer	ST Micro LSM303D 14 bit accelerometer	$\pm$16g	0.732 mg/LSB			
	magnetometer	$\pm$2 gauss	0.08 mgauss / LSB		5 mgauss/RMS	
Barometer	MEAS MS5611 Barometer Pressure	10-1200 mbar				
	Temperature	-40-85 C				10
GPS Position	ublox NEO-7 series	N/A			3 m (hor), 6 m (vert)	10
GPS Velocity	ublox NEO-7 seris				0.12 m/s	10

To estimate the states not directly provided by the sensor suite, an unscented Kalman filter was developed and integrated into the revised measurement subsystem, as described in Appendix Appendix C,

Once completed, the measurement subsystem was analyzed to determine whether or not it satisfied its requirements, which were derived from the flight control system interface. The result of the analysis is shown in Table 4, below.

Table 4: Estimator Output with Nominal Sensors

Output	Noise STD Limit	Estimate STD	Bias Limit	Estimate Bias
Roll	1.54	1.56	±0.28	0.73
Pitch	0.43	1.78	±1.20	0.54
Yaw	0.43	0.53	±0.55	1.02
Roll rate	0.24	0.11	±1.80	0.08
Pitch rate	0.24	0.11	±1.80	0.06
Yaw rate	0.24	0.08	±1.80	0.02
Air speed	0.60	0.15	±0.60	–0.02
Altitude	0.42	0.04	±0.60	–0.01
GPS Position	0.96	2.88	±1.20	0.32

As is clear from the table, several of the signals do not satisfy the requirements. Two signals, in particular, deviate significantly from their noise limit: pitch and GPS position. The noise limit for pitch is 0.43, but the estimate provided by the measurement subsystem is 1.78 — more than four times the limit. The noise limit for GPS position is 0.96, but the estimate provided by the measurement subsystem is 2.88 — exactly three times the limit.

Compositional reasoning would therefore lead us to conclude that this measurement subsystem is also unacceptable for our system. While there is no contextual compatibility mismatch, since the unscented Kalman filter is appropriate for systems with nonlinear dynamics, the measurement subsystem does not satisfy the design demands imposed upon it by the flight control system. In the argument, this failure would be identified in the contract between the measurement subsystem and the flight control system. The argument is shown in Figure 47 and the claim with failing support is highlighted in red.

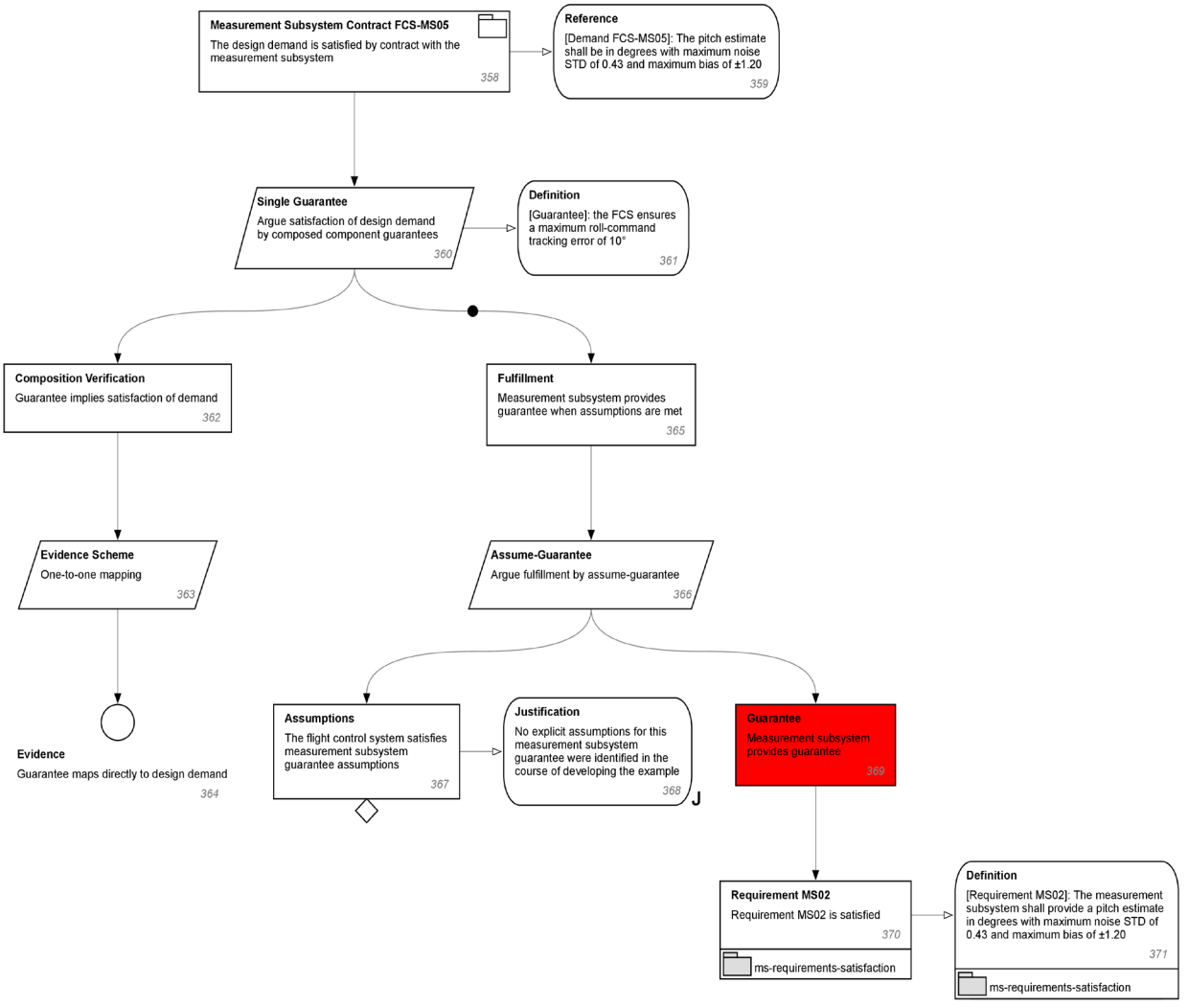


Figure 47: FCS-Measurement Subsystem Contract Failure

To validate this conclusion, the team simulated missions using the developed measurement subsystem. As expected, the mission was only successful about 62% of the time. The conclusion reached by composition reasoning was therefore correct: this measurement subsystem is not acceptable.

To better understand the mission impact resulting from the failure of the measurement subsystem to satisfy its design demands, the team explored measurement subsystems with varying levels of

GPS position error[12]. Position is a very important measurement, as errors in position can cause captured images to not line up, resulting in insufficient coverage of the mission area.

The tables below show output of the estimator using nominal output from all sensors *except* GPS. GPS noise is scaled from 0.0 to 1.0 and the resulting output from the estimator is shown, along with the probability of mission success.

Table 5: Estimator Output with Scaled GPS Noise — Noise

	Roll	Pitch	Yaw	p	q	r	V	HAGL	GPS	Success
Limits	1.51	0.43	0.43	0.24	0.24	0.24	0.60	0.42	0.96	-
0.0	1.32	1.13	0.52	0.08	0.05	0.07	0.13	0.04	0.04	96%
0.4	1.35	1.13	0.52	0.08	0.05	0.07	0.13	0.04	1.19	94%
0.8	1.38	1.13	0.53	0.08	0.05	0.07	0.13	0.04	2.34	76%
1.0	1.33	1.13	0.53	0.08	0.05	0.07	0.13	0.04	2.89	62%

Table 6: Estimator Output with Scaled GPS Noise — Bias

	Roll	Pitch	Yaw	p	q	r	V	HAGL	GPS	Success
Limits	0.28	1.20	0.55	1.80	1.80	1.80	0.60	0.60	1.20	-
0.0	0.31	0.25	0.51	0.04	0.03	0.01	–0.01	–0.01	0.15	96%
0.4	0.36	0.23	0.51	0.04	0.03	0.01	–0.01	–0.01	0.16	94%
0.8	0.34	0.29	0.51	0.04	0.04	0.01	–0.01	–0.01	0.15	76%
1.0	0.33	0.28	0.52	0.04	0.04	0.01	–0.01	–0.01	0.16	62%

These results point to the importance of having a GPS that meets mission requirements, confirming that the design demand from the control system to the measurement subsystem was necessary.

For the purpose of the example, this is an excellent result. There are many approaches that could be taken to address the failure of this measurement system to meet its requirement. For example:

- A better GPS sensor could be sought. In practice, this is difficult due to fundamental limitations of GPS. However, it may be possible to identify a sensor or set of sensors that could yield a better position measurement.
- A better state estimator could be developed. The kalman filter currently used for state estimation has not been aggressively tuned. A more aggressively tuned estimator may be able to better filter GPS position errors.
- Flight-path characteristics could be changed. The sensitivity to GPS position errors arises from the amount of image overlap that results from the selected flight path. At the air-vehicle, a

[12] The team focused on GPS position noise rather than pitch noise because the investigation of the previous measurement subsystem had revealed that the system is not very sensitive to errors in pitch.

design decision was made to target a 10-meter overlap in the area to be imaged on each leg of the flight path. With the selected roll and pitch limits, a 10-meter overlap does not offer very much margin to account for position errors. Increasing the overlap from 10 to 15 meters would likely permit a relaxation in the requirements on the measurement system. This would change the context for the flight control system, making it easier for the flight control system — and hence the measurement subsystem — to satisfy mission requirements.

4.2.4 Discussion

Throughout the construction of the Ultra Stick UAS example problem, we applied system-interface abstraction technology at each level of decomposition. Application of the reference model and processes during development facilitated avoiding the introduction of design decision while requirements were being developed.

For example, in stating requirements at the UAS level, it is tempting to phrase the requirements in terms of roll and pitch angle limitations on the air vehicle — but the use of an air vehicle is a design decision, albeit a design decision strongly implied by the term "UAS". Thinking strictly in terms of problem elements and environmental phenomena led us instead to phrase the requirements in terms of the camera and its relationship to the ground.

An even better approach would be to state the requirements in terms of characteristics of the image. For example, resolution might be stated in terms of pixels per square meter and distortions might be stated in terms of parallel line pairs. Unfortunately, the team does not have sufficient experience with the general problem of surveillance to confidently state such requirements and then reduce them to flight-path characteristics. As such, we stated the requirements more directly in terms of camera angles and height above ground level.

System-interface abstraction technology similarly informed the design architecture of the air vehicle and the flight control system. As is discussed above, there are a variety of architectures that can be envisioned for an air vehicle and a flight control system. The critical driver in determining the architecture is the order in which design decisions will be made. Because the air vehicle and the flight control system exist in a closed loop, design decisions on any component of these two systems influence every other component to some degree.

For the purpose of this example, the team elected to fix design decisions of the air vehicle first, including performance characteristics, control surfaces, and servos. This ordering of design decisions seems particularly well suited for an unmanned aircraft system. Since there are a variety of commercial-off-the-shelf airframes available with integral control surfaces and sometimes servos, these components are likely to be selected first.

The flight control system could be decomposed so that each control loop is a separate component. However, the difficulty of identifying a sufficiently strong interface to allow effective separation of the control loops led the team to treat the control loops as integral to the flight control system.

Instead, we decompose the flight control system so that the measurement subsystem is identified as a component. This decision is perhaps atypical of control system for a small unmanned aircraft system, but is nevertheless informative. The measurement subsystem allows us to demonstrate both the utility of system interface abstraction technology, but also highlights a significant challenge associated with the application of the technology: the identification and exploration of the tradeoff space associated with the controller inputs.

There are nine inputs from the measurement subsystem to the flight control system, counting GPS latitude and longitude as a single input. As discussed above, the team first identified the maximum noise and bias allowable for each input *separately*. Then, the team scaled the vector of maximum noise and bias for all signals simultaneously until the mission succeeded. The resulting vector was used as the input interface for the flight control system and drove the design demands levied on the measurement subsystem.

This approach, however, assumes that the sensitivity of the flight control system to noise and bias on each signal is equal. For the selected mission, this is unlikely to be true. A more robust interface would consider the sensitivity of overall mission success to noise and bias on each signal.

The advantage of a more robust interface is that mission success may be assured under a broader selection of measurement subsystems — and, ultimately, sensors. This flexibility comes at a cost: a much more in-depth analysis is required to identify the more robust interface. Once the design of the flight control system is fixed, this analysis only needs to be performed once, however. If the system is envisioned to be long-running or if a variety of sensors must be supported, the cost of the more in-depth analysis may be merited.

For this example, the team elected not to pursue the more robust interface and instead focused on the development of the measurement subsystem including sensor and estimator selection.

4.2.5 Artifacts

At each level of decomposition, sample artifacts were developed to illustrate the reference model objects. Many of these artifacts are summarized above; additional artifacts are shown here, for illustration.

The repetitive appearance of the artifacts is intentional and is a positive outcome of the application of system-interface abstraction technology. At each level of system decomposition, artifacts are instantiated from patterns. These patterns not only reduce development cost, but promote quality during system development: the patterns ensure that important elements of development are addressed. Additionally, the patterns facilitate review of the system.

4.2.5.1 UAS Successful Development

The top-level argument for the Ultra Stick UAS is an argument for successful development. The argument, shown in Figure 48, contains all of the elements that are identified by the reference model:

- Identification of the problem;
- Identification of the requirements;
- Identification of the context; and
- Satisfaction of requirements.

Assessment of safety is shown in the argument, but is left undeveloped as no safety requirements or specific hazards were identified during the development of the example. Instead, for this example, the focus is on the satisfaction of requirements.

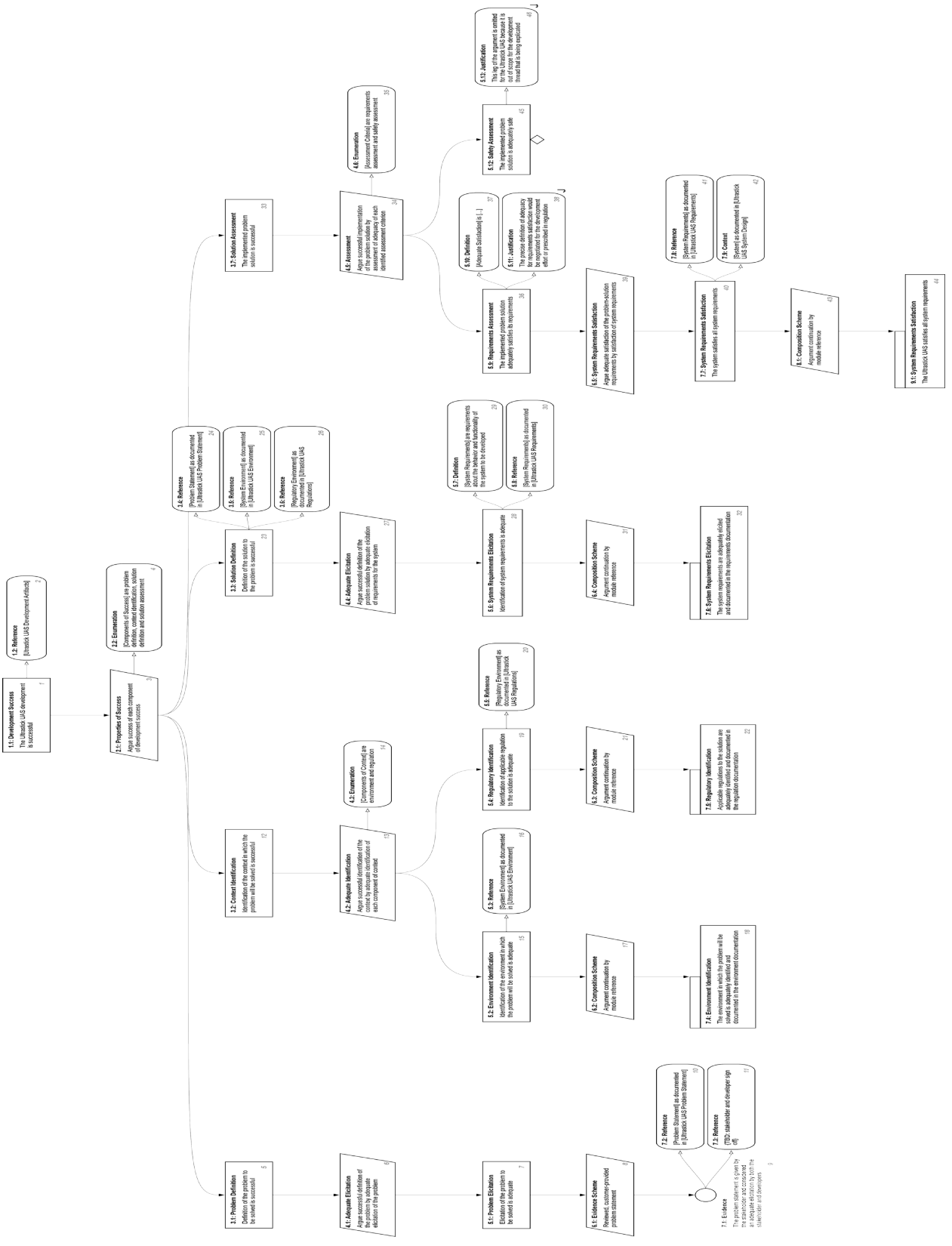


Figure 48: Ultra Stick UAS — Successful Development

4.2.5.2 UAS Requirements Satisfaction

The argument for satisfaction of Ultra Stick UAS requirements takes each requirement and argues that the requirement is satisfied by the design. The argument is shown in Figure 49 Where the design delegates satisfaction of a requirement to a component, the argument additionally argues that the design demand is satisfies by the associated component.

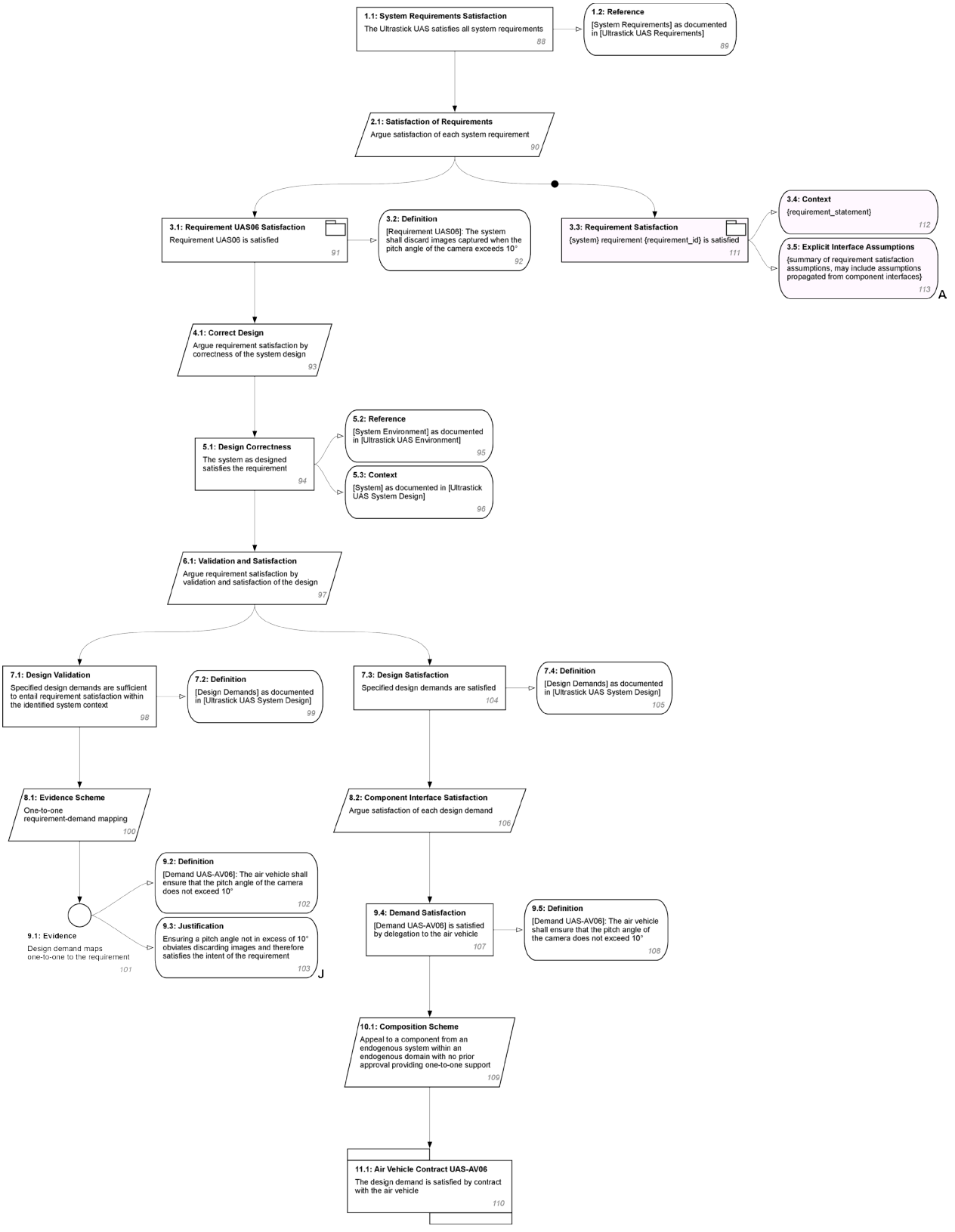


Figure 49: Ultra Stick UAS — Requirements Satisfaction

4.2.5.3 Ultra Stick UAS – Air Vehicle Contract

The contract between the UAS and the air vehicle states that the UAS design demand is met by the guarantee provided by the air vehicle when the UAS meets air-vehicle assumptions. The associated argument is shown in Figure 50. In this example, no explicit air-vehicle assumptions were identified that must be met by the UAS.

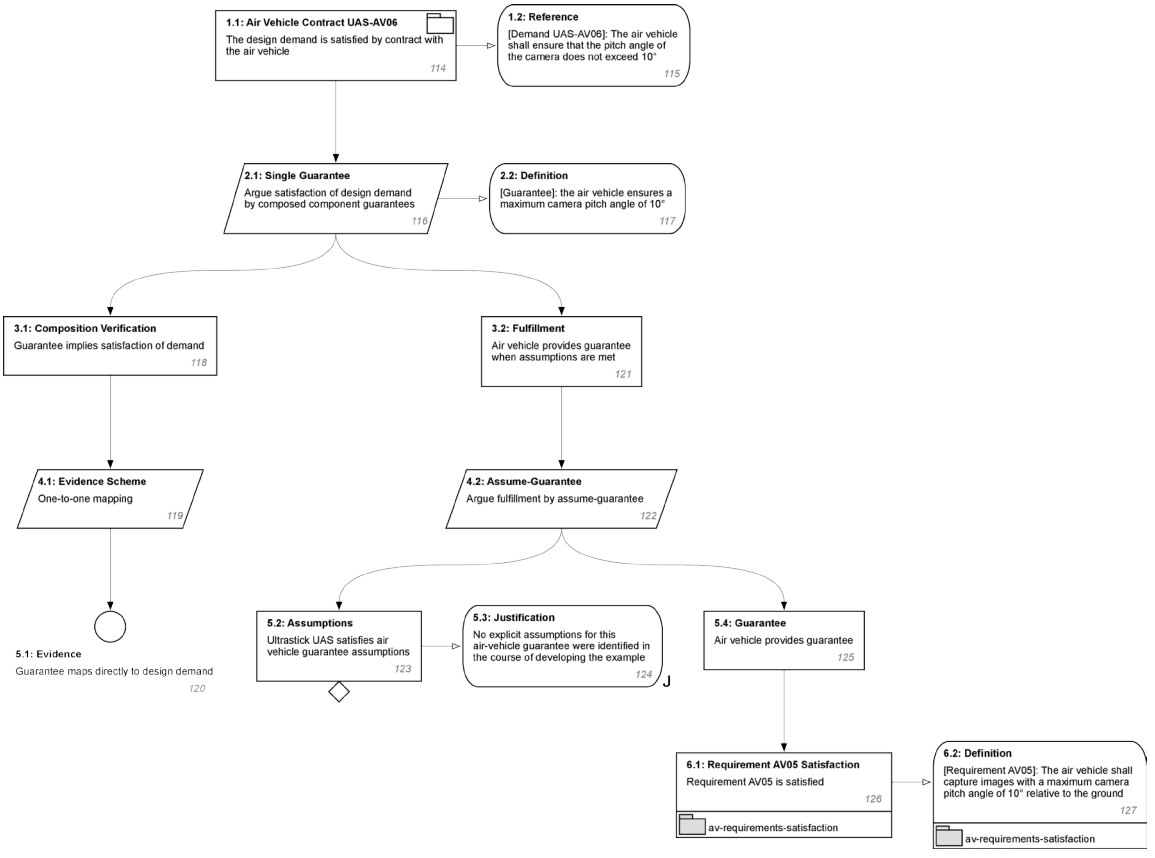


Figure 50: Ultra Stick UAS — Air Vehicle Contract

4.2.5.4 Air Vehicle Successful Development

The top-level argument for the air vehicle is an argument for successful development. The argument, shown in Figure 51, contains all of the elements that are identified by the reference model except for problem identification:

- Identification of the requirements;
- Identification of the context; and
- Satisfaction of requirements.

The air vehicle is viewed as purpose-built for this example, and as such does not have a separate problem that it seeks to solve. As above, for this example, the focus is on the satisfaction of requirements.

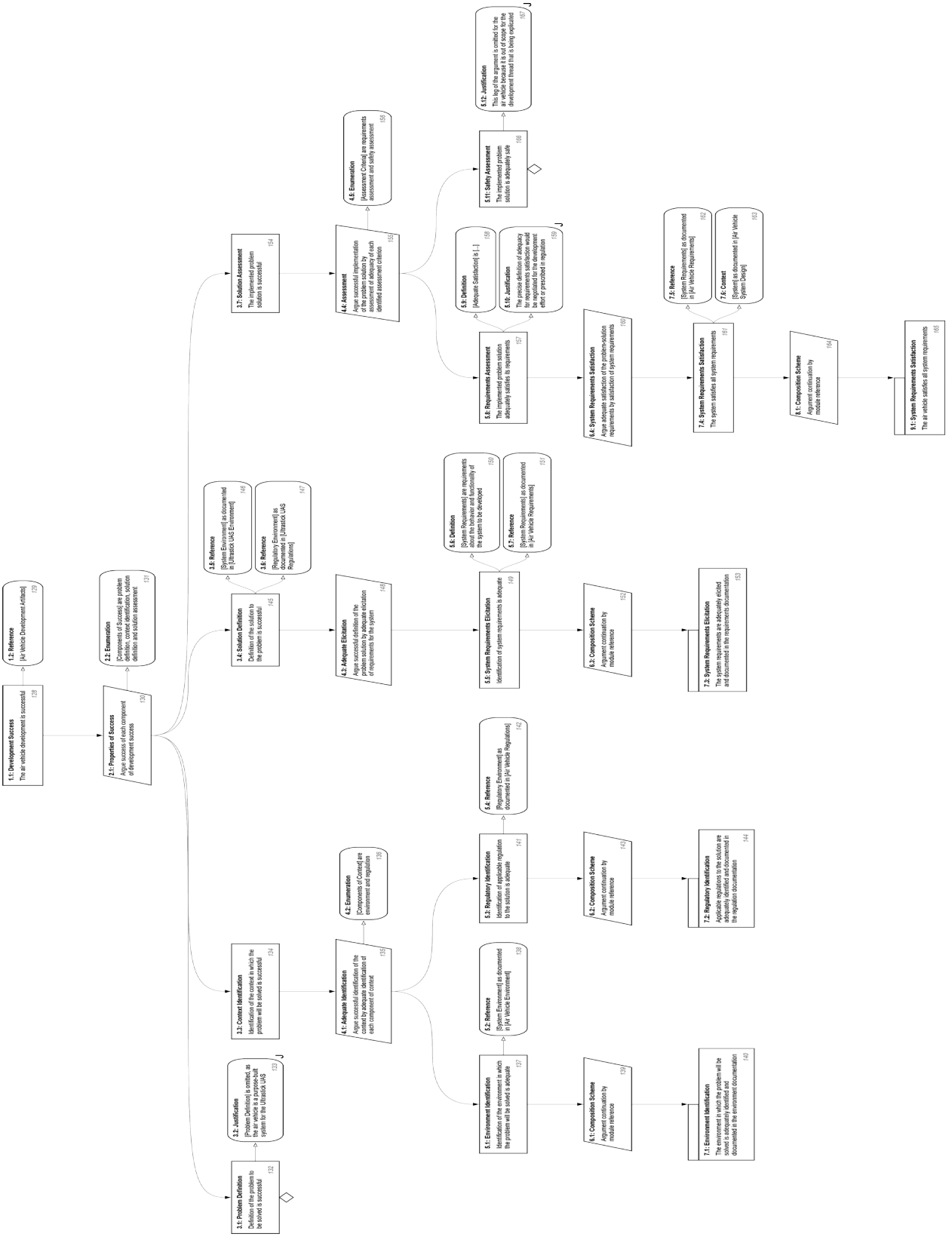


Figure 51: Air Vehicle — Successful Development

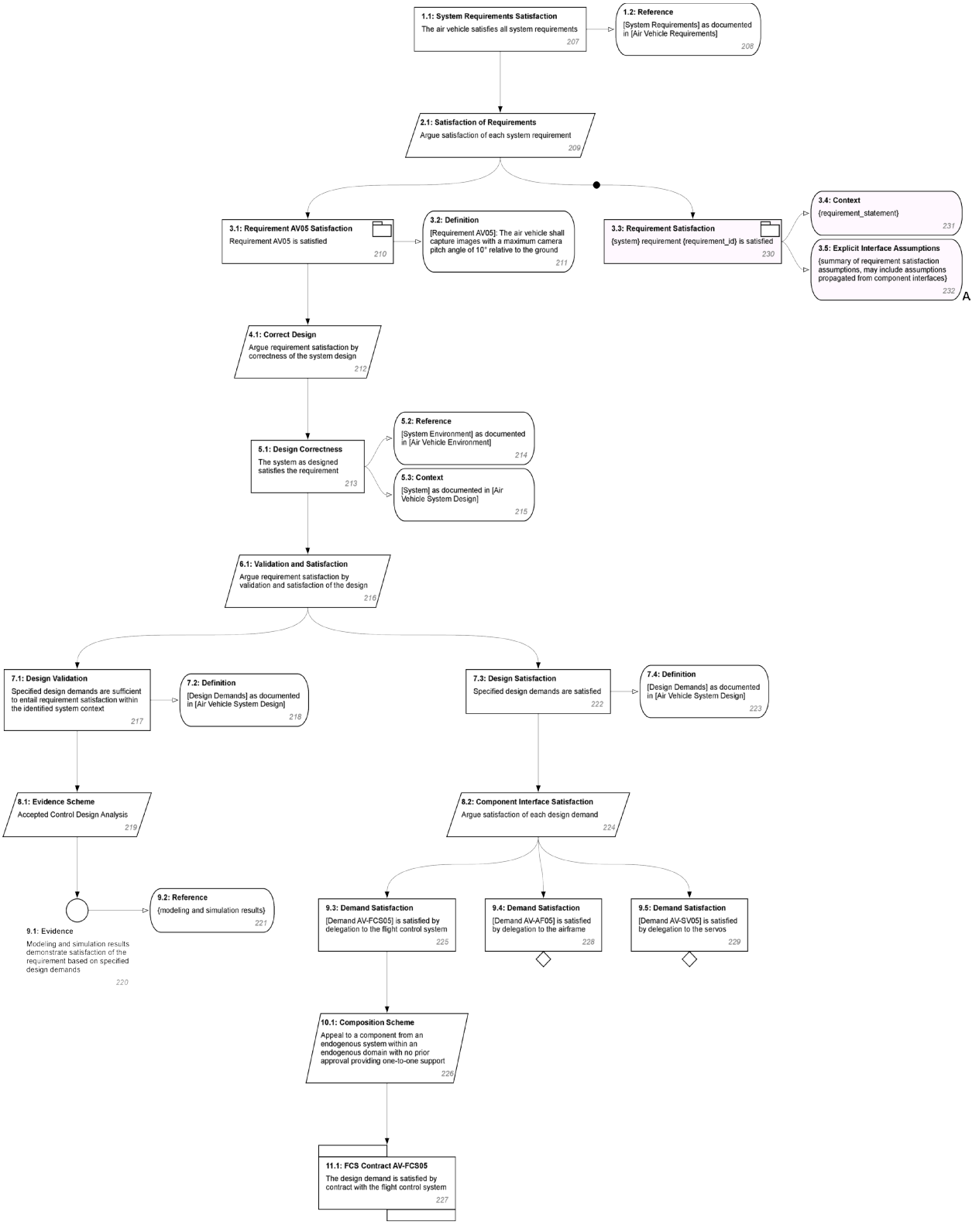


Figure 52: Air Vehicle — Requirements Satisfaction

4.2.5.5 Air Vehicle Requirements Satisfaction

The argument for satisfaction of the air vehicle requirements takes each requirement and argues that the requirement is satisfied by the design. The argument is shown in Figure 52. Where the design delegates satisfaction of a requirement to a component, the argument additionally argues that the design demand is satisfies by the associated component.

4.2.5.6 Air Vehicle – Flight Control System Contract

The contract between the air vehicle and the flight control system states that the air vehicle design demand is met by the guarantee provided by the flight control system when the air vehicle meets flight-control-system assumptions. The associated argument is shown in Figure 53. In this example, no explicit flight-control-system assumptions were identified that must be met by the air vehicle.

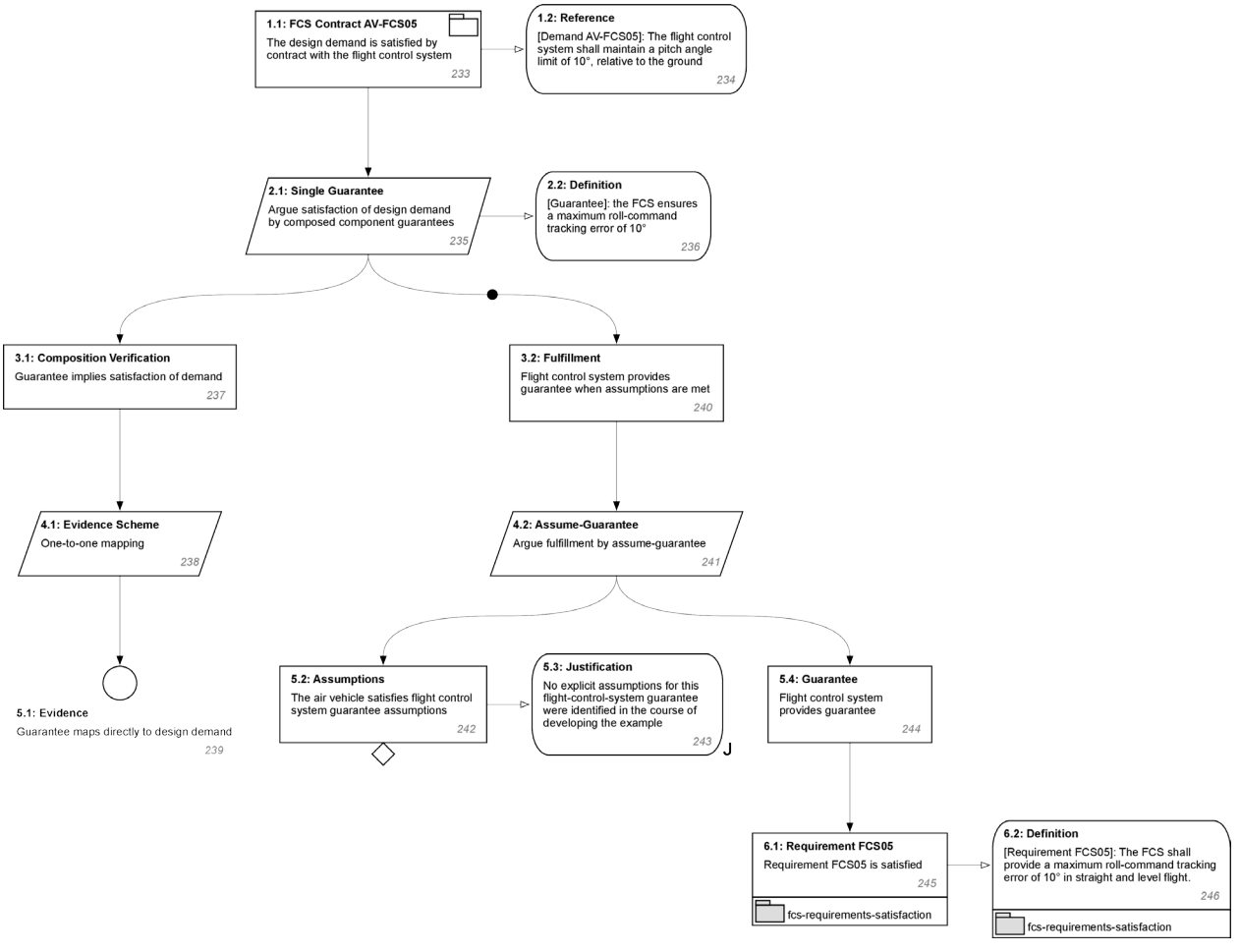


Figure 53: Air Vehicle — FCS Contract

4.2.5.7 Flight Control System Successful Development

The top-level argument for the flight control system is an argument for successful development. The argument, shown in Figure 54, contains all of the elements that are identified by the reference model except for problem identification:

- Identification of the requirements;
- Identification of the context; and
- Satisfaction of requirements.

Like the air vehicle, the flight control system is viewed as purpose-built for this example, and as such does not have a separate problem that it seeks to solve.

As above, for this example, the focus is on the satisfaction of requirements.

4.2.5.8 Flight Control System Requirements Satisfaction

The argument for satisfaction of the flight control system requirements takes each requirement and argues that the requirement is satisfied by the design. The argument is shown in Figure 55. Where the design delegates satisfaction of a requirement to a component, the argument additionally argues that the design demand is satisfies by the associated component.

4.2.5.9 Flight Control System – Measurement Subsystem Contract

The contract between the flight control system and the measurement subsystem states that the flight control system design demand is met by the guarantee provided by the measurement subsystem when the flight control system meets measurement-subsystem assumptions. The associated argument is shown in Figure 56. In this example, no explicit measurement-subsystem assumptions were identified that must be met by the flight control system.

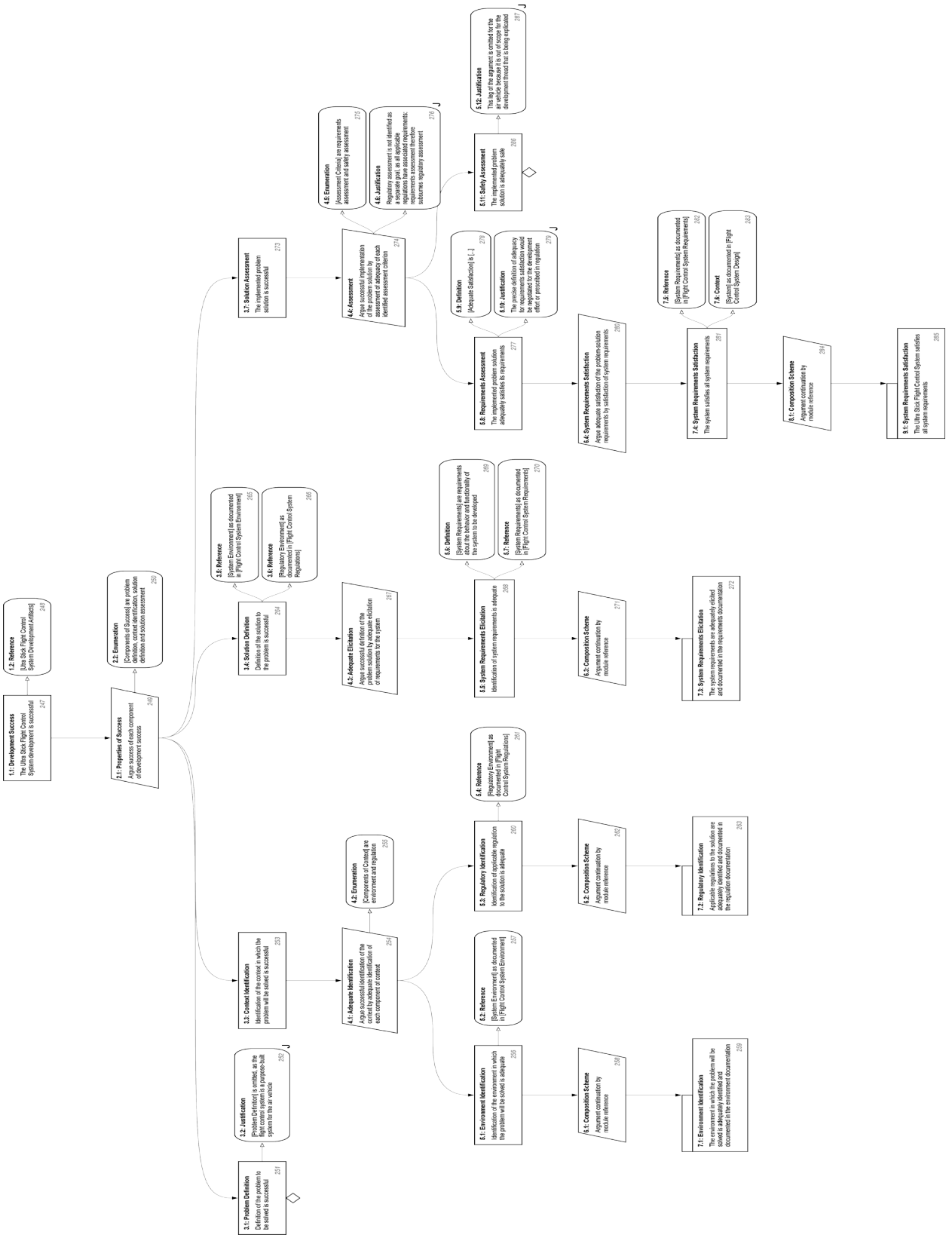


Figure 54: FCS — Successful Development

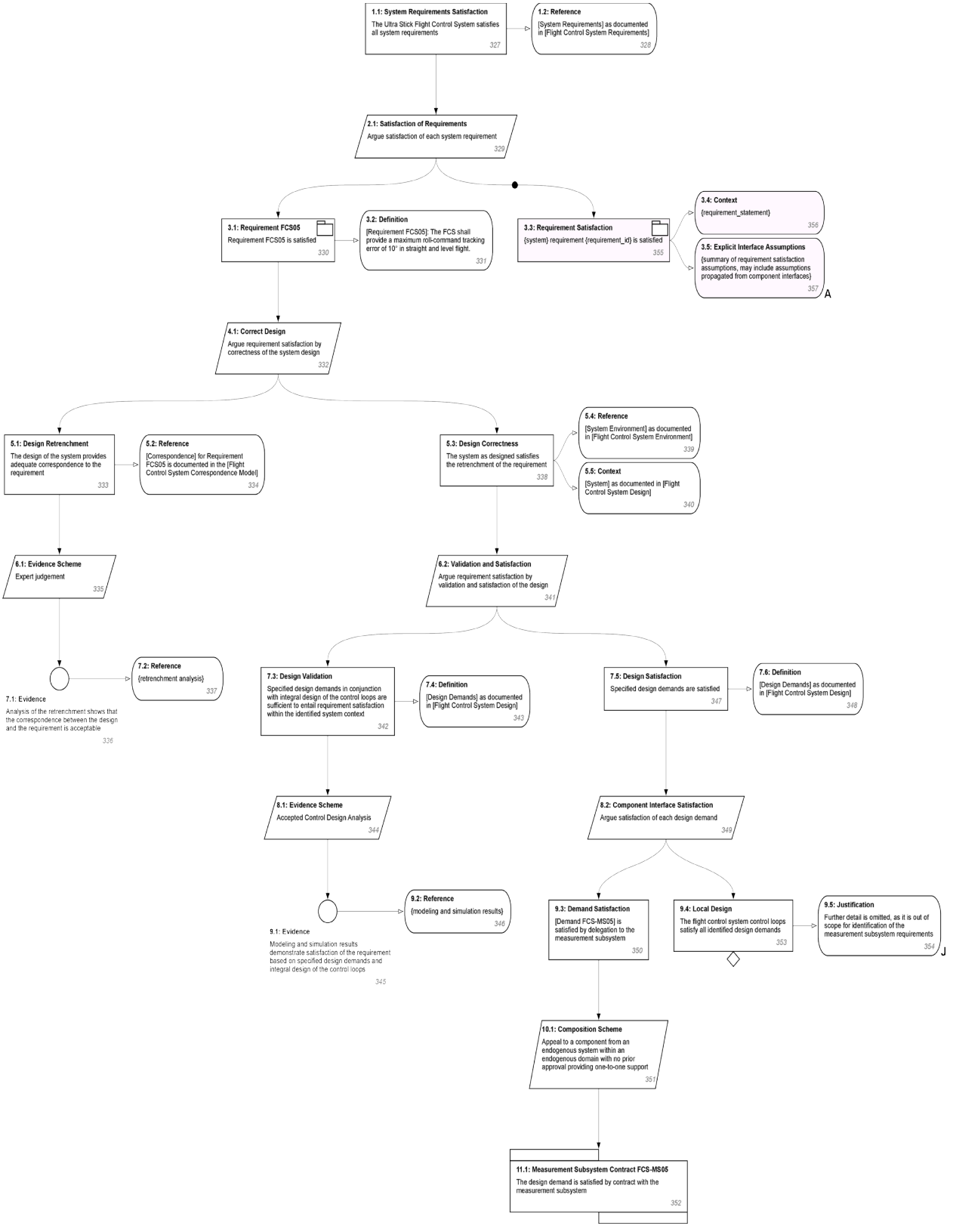


Figure 55: FCS — Requirements Satisfaction

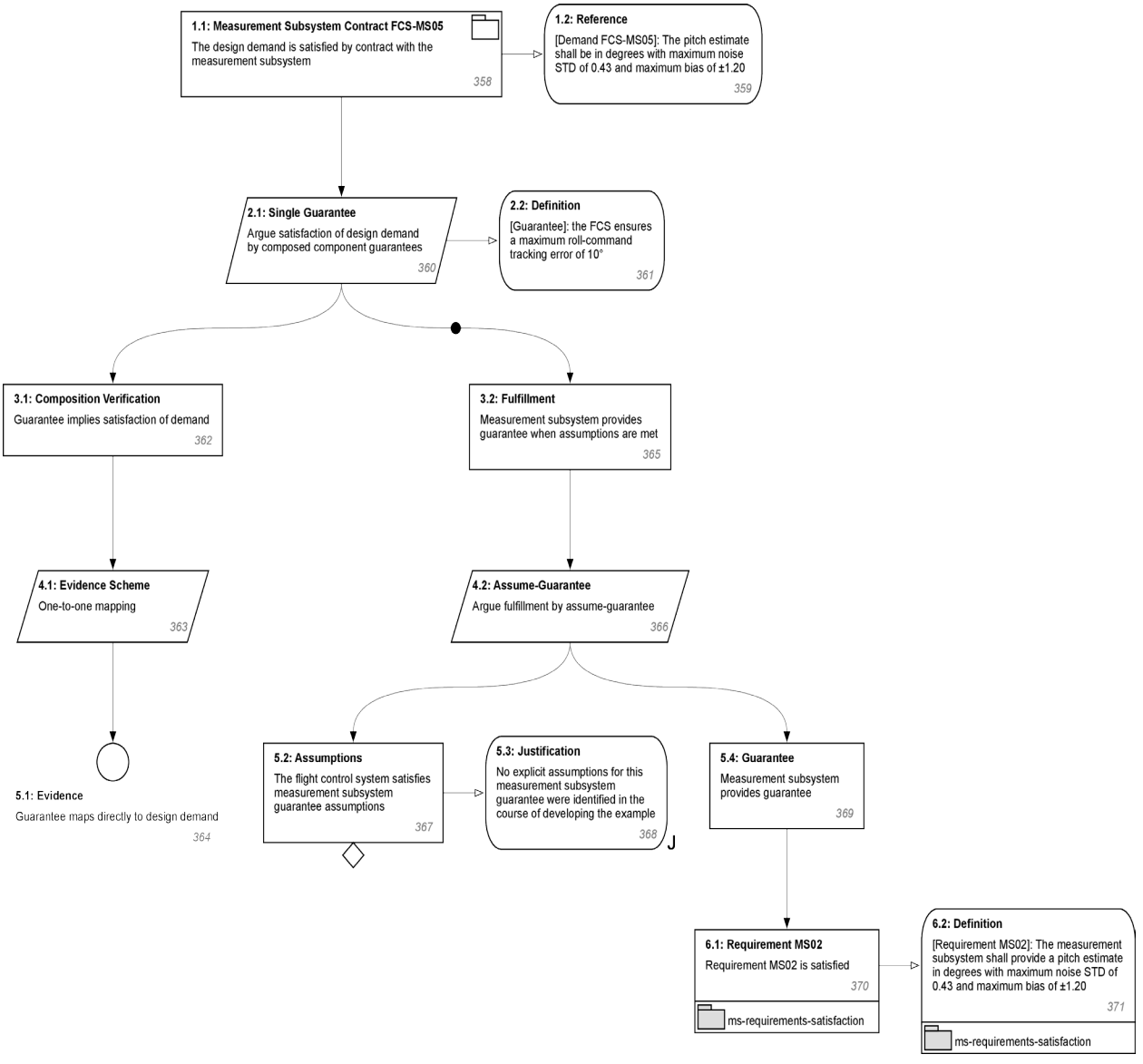


Figure 56: FCS — Measurement Subsystem Contract

4.2.5.10 Measurement Subsystem Successful Development

The top-level argument for the measurement subsystem is an argument for successful development. The argument, shown in Figure 57, contains all of the elements that are identified by the reference model except for problem identification:

- Identification of the requirements;
- Identification of the context; and
- Satisfaction of requirements.

Like the flight control system, the measurement subsystem is viewed as purpose-built for this example, and as such does not have a separate problem that it seeks to solve.

As above, for this example, the focus is on the satisfaction of requirements.

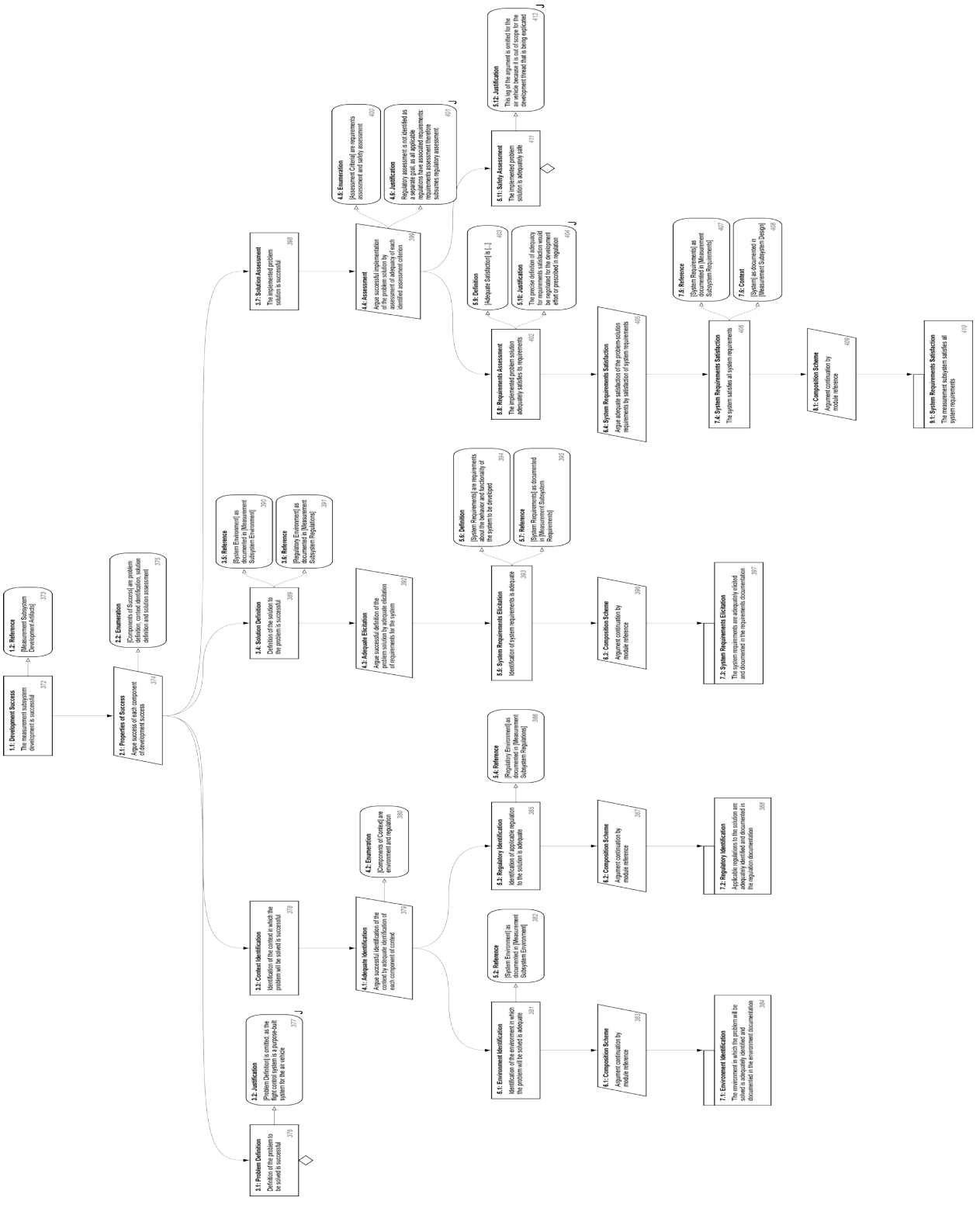


Figure 57: Measurement Subsystem — Successful Development

4.2.5.11 Measurement Subsystem Requirements Satisfaction

The argument for satisfaction of the measurement subsystem requirements takes each requirement and argues that the requirement is satisfied by the design. The argument is shown in Figure 58. The argument terminates a the design demands for the two sibling components of the measurement subsystem: the estimator and the sensor suite. Detailed consideration of these demands and the their satisfaction is out of scope for the example, so the argument is left undeveloped at these claims.

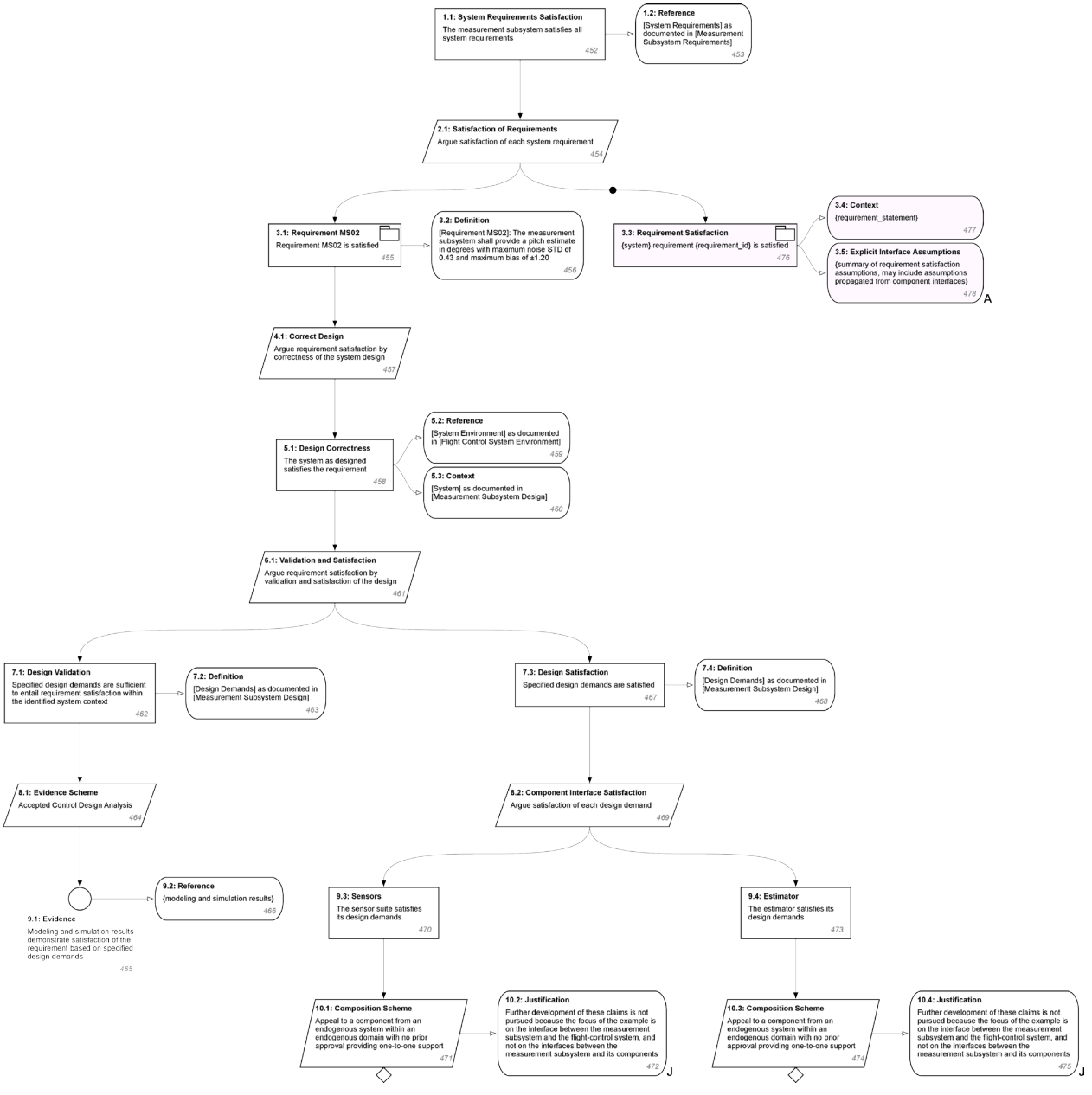


Figure 58: Measurement Subsystem — Requirements Satisfaction

4.2.6 Conclusion

Through the application system-interface abstraction technology, the team developed an example based upon a hypothetical small UAS based on the Ultra Stick platform. Using the example, we explored the design and replacement of a measurement subsystem, and used compositional reasoning to conclude that neither the original nor its replacement were satisfactory. The first measurement subsystem made an unreasonable assumption about what sensors were available and made unsupported assumptions about the system context. The second measurement subsystem addressed both these issues, but failed to satisfy its requirements, leading to a broken contract between the flight control system and the measurement subsystem. For both of the measurement subsystems, we validated the conclusions reached through compositional reasoning by conducting simulations at the system level.

System-interface abstraction technology worked well where it was exercised through the development of the example. The explicit and careful seperation of requirements, context, and design that is called for by the reference model and reference processes led to a set of development artifacts that are easily defended: there is little question that design decisions are appropriately delayed to design or that the context has been clearly identified.

The technology also led to the development of a decomposition that is easily defended. The design authority view, shown in Figure 46, clearly separates the major components of the UAS. The modularity provided by this design facilitates argumentation and supports component upgrade/replacement throughout the life of the UAS.

While the example illustrates many aspects of system-interface abstraction technology well, it is not without limitations. Although modularity was clearly identified, full compositional reasoning and argumentation was not demonstrated for the complete system. Such reasoning and argumentation depends highly on domain knowledge, as the properties of interest and the way in which abstractions may be established and may be argued to satisfy through properties are deeply rooted in the domain. While the team had significant domain expertise, the team was not used to applying this expertise towards modular design or compositional reasoning — instead, the team typically designs small, bespoke, monolithic systems for research purposes, often with an eye towards optimality. As a result, development of the arguments and associated artifacts for the example was largely limited to pattern instantiation.

The observation that *both* domain expertise and experience in modular design and compositional reasoning is required for successful application of system-interface abstraction technology points to parameters for future studies that will demonstrate the efficacy of the approach. A team must be assembled that has both domain expertise and experience in modular design and compositional reasoning, so that the new system developed with system-interface abstraction technology can fully exercise all of the elements of the theory. Alternatively, system-interface abstraction technology could be applied *post hoc* to an existing system that is already modularized and that already leverages compositional reasoning.

Finally, the ultra stick example problem was developed to enable us to refine the theory through its application. Throughout the development of the example, the theory continued to evolve to address identified limitations of the theory. Although we were unable to fully evaluate the technology using this example, the example was nevertheless instrumental in the development of system-interface abstraction technology.

4.3 Examples of Argument Recovery

This section reports on the examination of MIL-HDBK–516C (516C) and a Joint Service Specification Guide (JSSG) for the presence of domain arguments–rationale arguing from the perspective of technical expertise. The remainder of this section discusses the motivation for this activity, the results of analysis, and conclusions about how domain arguments impact the utility of requirements guidance.

Argument recovery enables SIAT to incorporate domain arguments as opposed to imposing strict argument structures. For example recovered arguments might support the success argument goals of regulatory compliance and requirements satisfaction.

4.3.1 Motivation

MIL-HDBK–516C (516C) and the Joint Service Specification Guides (JSSGs) instruct system developers in the kinds of requirements that should be specified for a succesful project. Both documents attempt to communicate expert knowledge about requirements development, in part, because failure to capture the kinds of requirements they describe can lead to project overruns and failures due to poor system specification.

516C focuses on airworthiness requirements. Specifically, page 49 of MIL-HDBK–516C states "The following criteria, standards and methods of compliance apply to all air systems and represent the minimum requirements necessary to establish, verify, and maintain an airworthy design." The structure of criteria, standards, and methods of verification is repeated throughout the document in an attempt to create clarity and insight for the reader.

Joint Service Specification Guides present a "framework to be used by Government-Industry Program Teams in the Aviation Sector for developing program unique requirements documents for Air Systems, Air Vehicles, and major Subsystems." They provide guidance for specification of air force systems surrounding the kinds of functions and capabilities that are expected of an air force system while considering functionality of the system to be specified.

In both both 516C and JSSGs, guidance involves a structured document format. 516C categories criteria, and for each states standards, and methods sections . JSSGs requirements often include a 'Rationale' section. In both cases, the authors are attempting to convey guidance and why the guidance has value. It is this latter element in which recent research conducted for NASA may be of benefit.

Research with NASA under contract NNL13AA08C demonstrated that technical experts have detailed rationale to explain why what they do and build will work. This rationale can be captured and made explicit. From there it can be critiqued, improved, and shared more readily with a community of practice. Research work with NASA demonstrated a technique whereby existing writing could be parsed and processed for any rationale contained therein. Therefore it was hypothesized that rationale could be identified, extracted, and organized should it exist within the writing of 516C and JSSG documents. For example, 516C might describe why assurance methods demonstrate requirements satisfaction. Likewise, the explicit 'Rationale' sections of JSSGs might be parsed and analyzed to help explain the value of their requirement categories.

4.3.2 Domain Arguments

Rationale exist first and foremost in the minds of experts. The 'mind space' of experts is called a *domain* of expertise. Domains can be shared between people in communities of practice. Shared domains of knowledge include both common knowledge as well as common misperceptions and variances in belief and focus. In other words, a domain represents the conceptual space of knowledge in a discipline as it exists in the minds of its members.

Codified knowledge is domain knowledge that has been extracted from the abstract space and presented specifically. For example, text books, research studies, and tutorials represent codified knowledge from a domain.

Arguments are part of domain knowledge. Often, these arguments will be about why techniques work or principles hold. Such *domain arguments* have value to the communities that hold them. In some fields like mathematics, many arguments are proofs in deductive logic. The strength of proof forms the backbone of domain knowledge, and therefore they are explicit, codified and shared as artifacts between domain members. In other domains, where arguments apply inductive logic, argument is secondary knowledge to the accepted facts, techniques, principles, and 'laws' of the domain. Often techniques and principles are written down, but argument is only occasionally communicated [16].

In specialized fields, domain arguments are often implicit. For example, a software company might have engineering principles and rules that employees must read and follow, but argument for the merit of the rules is often discussed verbally within the organization. When domain arguments are written down they are frequently embedded within status reports that explain what and how things were done. Contrastingly, the arguments explain why such products and activities have value.

Recent work demonstrated that *domain arguments* can be recovered from the minds of experts as well as from the documents that experts write to one another. Example arguments were recovered from the reports of control engineers, the mental model of an FAA DER, and the submission forms for flight test ranges [16]. The following section describes the technique of extracting domain argument.

4.3.3 Overview of Technique

Previous work explored three mechanisms for recovery of domain arguments. These were:

1. *Argument Synthesis*: discussions with experts to retrieve argument,
2. *Argument Recording*: writing of argument in Goal Structuring Notation by experts, and
3. *Argument Recovery*: analysis of writing and reports to annotate and extract arguments from written narratives.

The later two techniques are detailed in the flow chart of Figure 59. Given that 516C and JSSG are already written documents, argument recovery is applied.

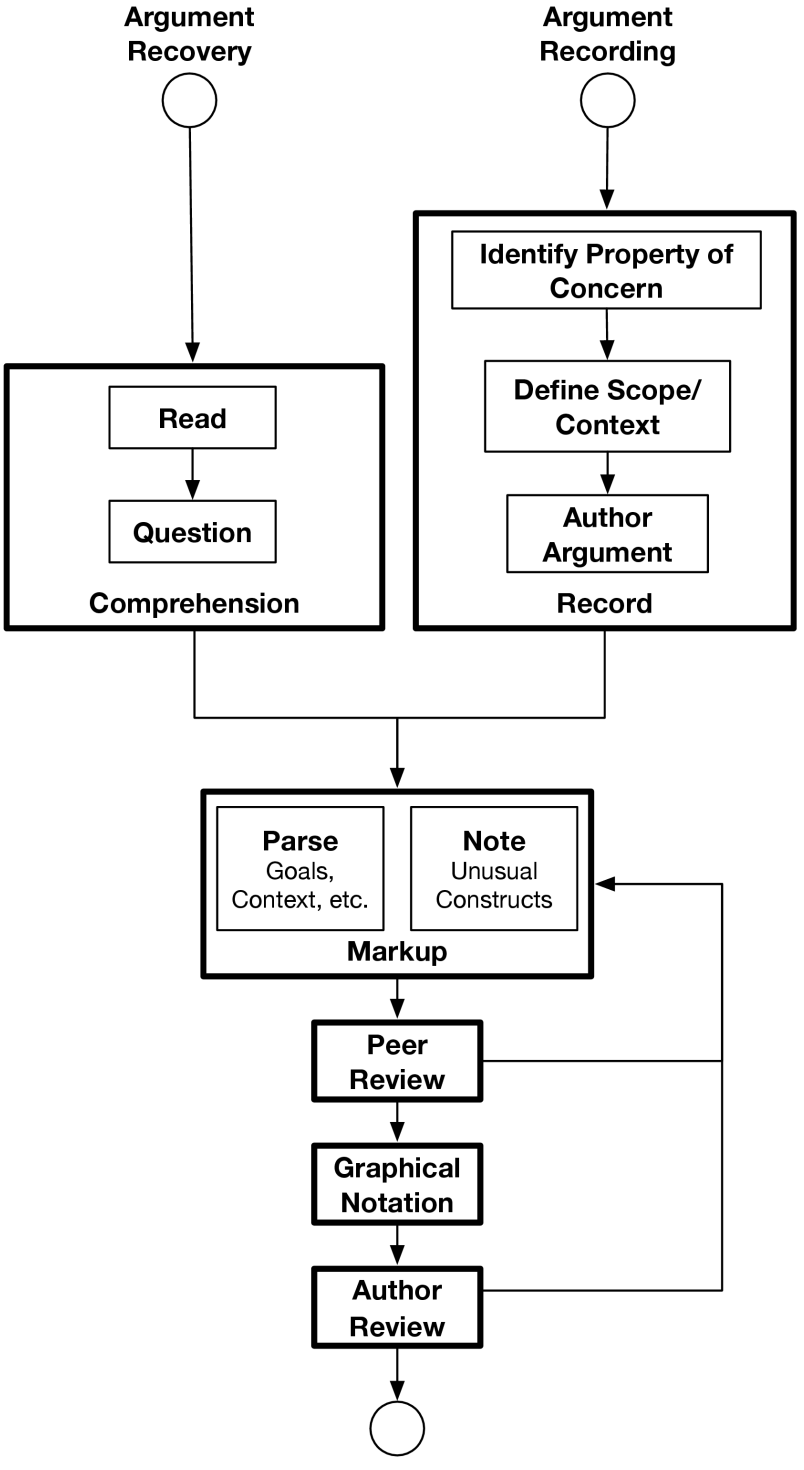


Figure 59: Argument Retrieval and Recovery Processes

There are two key roles in argument recovery. Domain experts write a document in the regular course of their work. Argument experts then work to identify and extract domain argument contained within the document.

There are several steps in the recovery process as shown in the figure. First, domain experts read the produced document. The domain experts provide feedback on its content in response to questions from the readers and follow-on discussions.

After comprehension is sufficient, the document is parsed for elements of argument as defined in Goal Structuring Notation (GSN). (Applying the conventions of GSN structure artificially reduces the narrative representation of the argument [16], but is judged by the researchers to be a reasonable step in this preliminary research.) As a consequence, text from the document is now a textually annotated form of GSN argument. Many argument fragments might be found in the text and might be dependent or independent of one another.

The resulting annotated text is rendered in GSN's graphical notation. The resulting graphical argument form is the initial result of argument extraction. However, it typically contains interpretation errors and must be reviewed by document authors or experts. Only after review by the original document authors is the argument considered to have representative validity.

Given that we do not have access to the authors of 516C or JSSGs, the results of the analysis in this report should be viewed as highly preliminary and likely to be in significant disagreement with the authors' intent. Therefore, the only conclusions that can be drawn from an analysis will be whether argumentative reasoning or rationale is present and the relative complexity of its presentation to the reader. The specific arguments recovered are speculative and represent a layman's interpretation of the text.

4.3.4 Analysis

This work examined specific samples of 516C and a JSSG document. The documents were not examined in the whole, as the purpose of the work was to assess feasibility of argument recovery and explore the initial hypothesis of domain argument presence in air force guidance documents. Given that the documents both had highly regularized and repeating structure, it was believed that a small sample analysis was sufficient to determine the general extent of argument presence.

4.3.4.1 JSSG

The purpose of examining a JSSG document was to determine if "Rationale" sections contained domain argument(s). In particular, because a rationale is often a justification for a position, statement, request, or standard, it was felt that such justification might take the form of argument.

For this work, "JSSG–2009: Air Subsystems", was analyzed due to its public availability and relevance to aviation systems safety. The main finding of the analysis was:

"Rationale" sections vary in what they describe. Some present *argument*, many present *context*. These represent information about 'why', and 'what', respectively.

Context is supporting information. For example, in Goal Structuring Notation, context can be attached to a goal to clarify its terms or conditions, to a strategy to elucidate information applied to creating sub-goals, or to evidence to help explain in detail what the evidence must include.

For example, Section "B.3.4.2 Hydraulic power subsystem", on page B–4 of JSSG–2009, states a "Requirement Rationale" section as follows:

The function of the hydraulic power subsystem is to deliver fluid at sufficient flow rates and pressure to the actuating devices in all modes of flight or ground operation. The speed of actuation is a function of fluid flow-rate whereas the actuating force is a function of pressure.

Hydraulic fluid power has been found to be the lightest and most efficient method to transmit high horsepower in air vehicles.

The above text provides context for the purpose of hydraulic fluid power systems. But it also presents a short argument about why hydraulic power is important (it is the lightest and most efficient power transmission method).

Section “G.3.4.7.30 Control systems integrity assurance provisions”, on page G–75, states the rationale for control systems integrity assurance provisions as:

This requirement defines the features of the systems intended to be used by the crew during preflight checks to determine the operational condition of the circuits and components of the control systems.

This provides context about what the requirements represent in terms of providing safety through crew preflight checks. It does not present argument. While one might infer reasons for the importance of this type of requirement from the required context, that reason is not explicitly stated nor directly implied. This is an example of how argument can be implied by with insufficient information to be recovered.

A final example of JSSG “Rationale” containing argument can be found on page A–109, “A.4.4.1.11.1.1 Air vehicle tire performance”. This section deals with the functional requirements of tire performance. The rationale provided is:

The use of a laboratory dynamometer to evaluate the tire performance characteristics permits evaluation to the limits of the tire capability with risk. The design conditions are carefully controlled and are repeatable. The Industry has always utilized this method of evaluation prior to installation on an air vehicle to determine performance limits and to establish safety of flight. It is significantly more economic than any other verification method. The tire will also be observed and evaluated during the routine flight test program.

The above rationale seems to be a statement about why the given analysis technique should be applied in the analysis of air vehicle tires. The argument for why the analysis technique (dynamometer in the lab) should be used is presented in Figure 60.

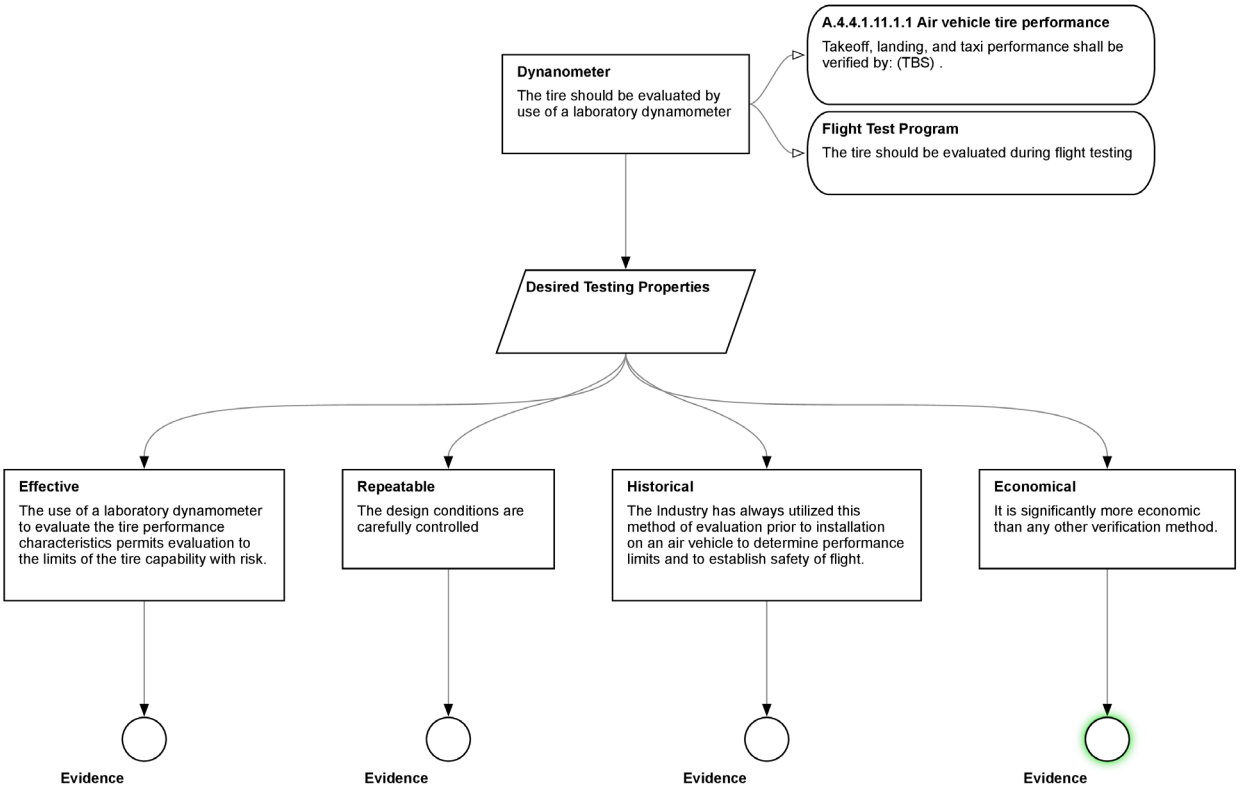


Figure 60: JSSG_2009 Rationale in GSN Form

The argument consists of properties of the "dynamometer" laboratory approach that make it valuable in testing of tire performance requirements.

In conclusion, it appears that the "Rationale" defined in JSSG–2009 consists of a mixture of context and argument. In some cases, the text is entirely context from which the reader is left to infer the reasoning of the rationale. In other cases, context is mixed with cursory attributes stated as matter-of-fact, without explication. In the final example, the rationale is an explicit domain argument regarding the value of a particular tire testing technique.

4.3.4.2 516C

Two specific criteria from MIL-HDBK–516C where chosen for analysis:

1. 8.3.2 Qualification Tests: Verify that all components, either individually or as part of a subsystem, have passed all safety-related qualification tests as required for airworthy performance.
2. 8.3.8 Fuel Transfer Rates: Verify that fuel transfer flow rates meet the operational ground and flight envelope requirements.

Argument analysis markup was applied to the text of 516C section 8.3.2 in Figure 61. The top goal of the argument is verification of passing of all qualification testing for fuel system components and subsystems. Context is provided to the scope of qualification testing. It is constrained to tests "as required for airworthy performance." The strategy applied is to demonstrate this is true of all fuel system components. Thus fuel system components are

enumerated under this strategy. For each component X, the goal must be that component X has been "subjected to qualification testing commendurate with [its] intended operational usage." Additional context is provided about the airworthiness "standards" to which components must be subject. This list could make up additional layers of the argument with appropriate standards applied for testing of each component.

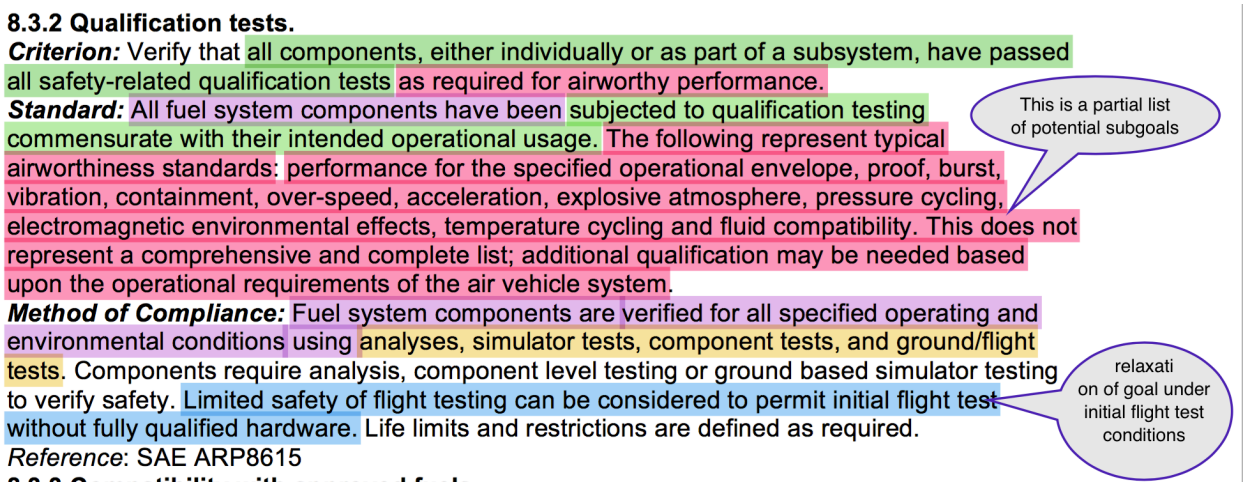


Figure 61: Argument Markup Example — Qualification Tests

Forms of evidence are presented under "Method of Compliance". Evidential approaches include "analyses, simulator tests, component tests, and ground/flight tests." These evidential approaches are applied under the strategy that components are "verified for all specified operating and environmental conditions". Under this strategy, either an enumerated space of condition combinations or some more complex data structure would represent the space of conditions. Evidential approaches would be applied as appropriate to each component/subsystem. There is an additional contextual note about flight testing that permits use of hardware that is not fully qualified.

The above interpretation of the text as argument is shown with GSN in Figure 62. This argument forms a pattern with multiple levels of decomposition. The decomposition is in the order of component/subsystems, standards of airworthiness, and environmental and operation conditions. Five types of testing results are given as potential evidence of having passed the qualification tests.

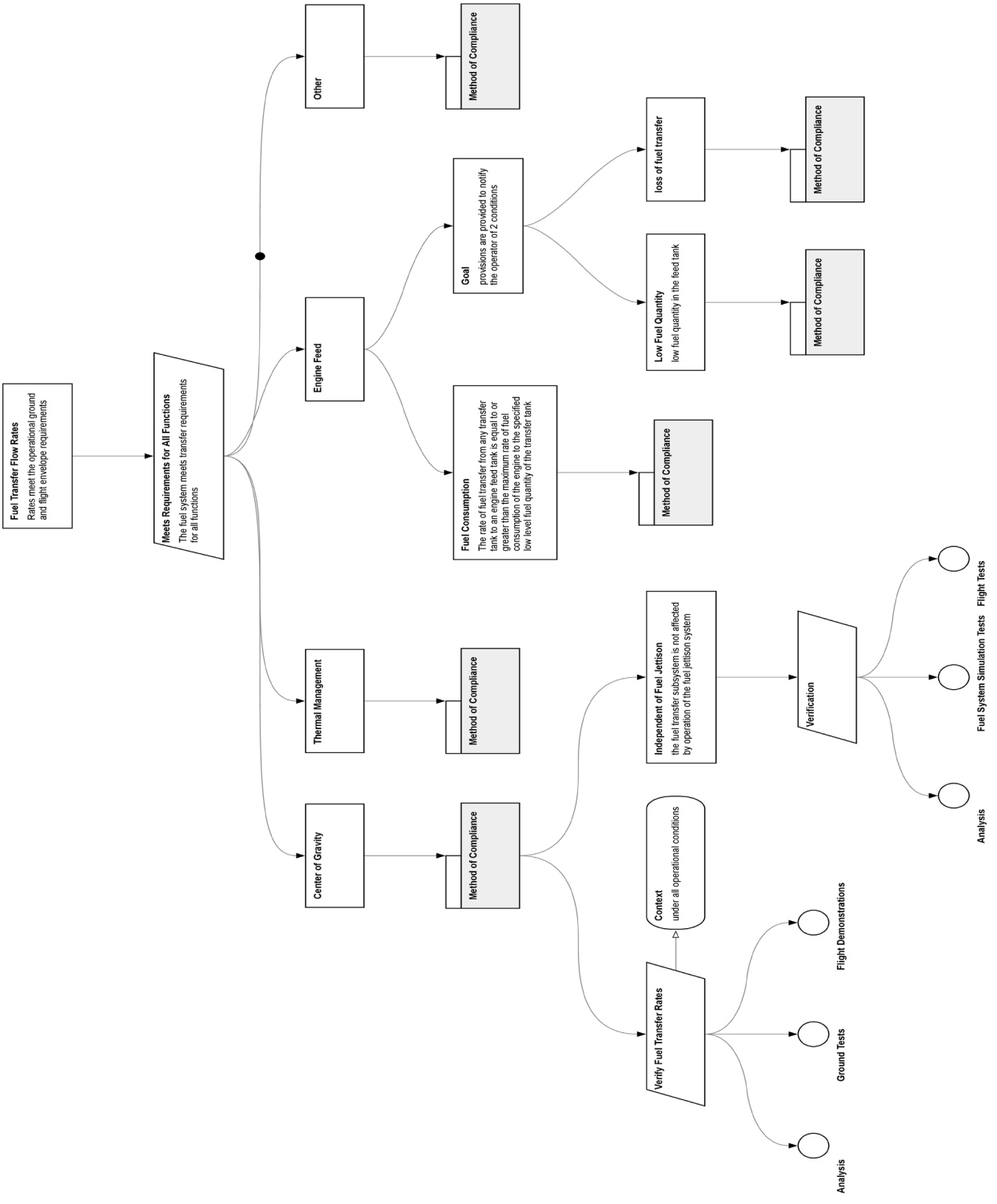


Figure 62: GSN for Qualification Tests

Argument analysis markup was applied to the text of 516C section 8.3.8 in Figure 63. The top goal of the argument is verification of fuel transfer rates. This must be asserted for ground and

flight envelope requirements. The strategy applied is to show that the system meets transfer requirements for "all functions". This is enumerated into a list of functions including, but not exclusive to "center of gravity management," "thermal management," and "engine feed". Our interpretation of the next two sentences were that they decomposed fuel transfer requirements into two sub-requirements: a constraint on the rate of fuel transfer, and "provisions are provided" to notify the operator of low two conditions: low fuel quantity or loss of fuel transfer.

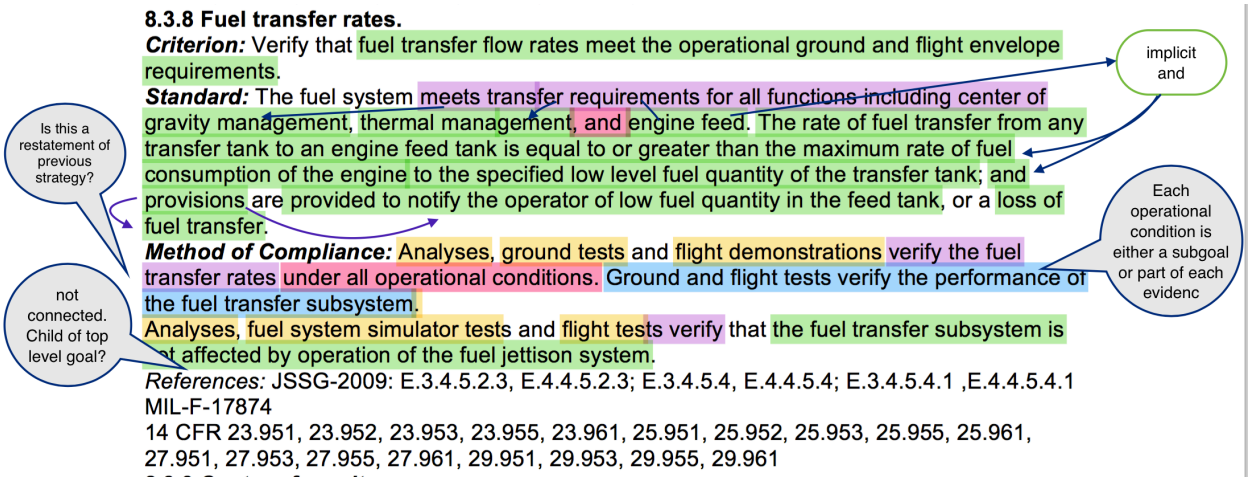


Figure 63: Argument Markup Example — Fuel Transfer Rates

The "Methods of Compliance" describe evidence that can be applied to the above argument structure. "Analyses, ground tests and flight demonstrations" are enumerated. These verify fuel transfer rates. Both ground and flight tests verify the system's performance. In addition, it must be shown that "the fuel transfer subsystem is not affected by operation of the fuel jettison system." Evidential approaches to this goal are enumerated in the text.

The above interpretation of the text as argument is shown with GSN in Figure 64. Note that the argument forms a pattern where additional functions (not specified but noted by section 8.3.8) must be analyzed.

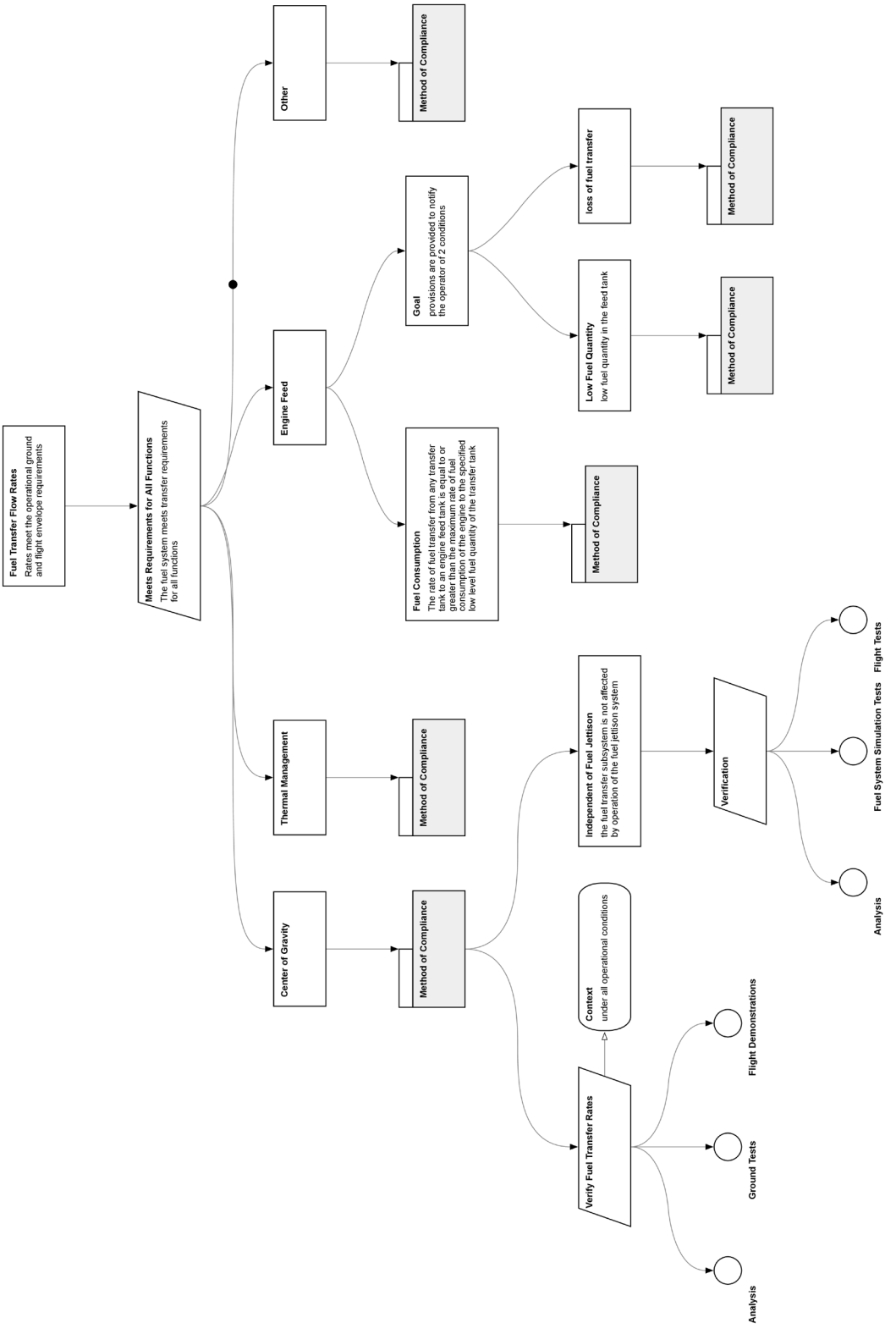


Figure 64: GSN for Fuel Transfer Rates

In conclusion, the sections of MIL-HDBK–516C that were analyzed contain substantial argument for sufficient verification of fuel systems. It is likely that this same pattern of inductive verification computation, from the minds of experts, is encoded throughout the document. However, we cannot determine the accuracy of the specific recovered arguments without access to the original document authors.

Due to the number of embedded arguments, 516C could be adapted to an argument-driven verification handbook, in which explicit arguments take the place of the existing "text buckets" in the document. An argument form, either as text or GSN, would replace the existing categorization of "criterion", "standard", and "methods of compliance" with their equivalent argument structure. The benefits of this approach would include:

1. more rigorous recording of expert rationale,
2. more homogeneous interpretation of the text, and
3. potential for incremental improvement of 516C arguments as lessons are learned.

4.3.5 Conclusions

JSSG–2009 and MIL-HDBK–516C were analyzed for the presence of domain arguments. In particular it was hypothesized that the "Rationale"s present in JSSG documents would be or contain strong arguments. It was hypothesized that the "Criterion", "Standards", and "Methods of Compliance" of the 516C handbook would argue for the value of requirements criteria.

The JSSG–2009 rationales sometimes contain explicit argument. However, this is not universally the case and many arguments are either implied without structure or propeties are asserted without supported reasoning. This is despite the structured documents use of the term "Rationale". By in large, the term *rationale* in JSSG documents appears to refer to *context*. Argument and inductive reasoning occurs in brief statements if at all. Much of the logic behind the context is left for the reader to infer. Where argument is present, its main purpose appears to be adjunctive.

In contrast, the 516C guidebook contains many domain arguments in which the veracity of a criterion is decomposed through sub-goals and means of evidence collection. Because of the "bucketed" nature of 516C, it is not always clear how the lower-level, "methods of compliance" directly relate to the criterion or supporting standards. It is possible that an argument-oriented approach to 516C handbook could elucidate these connections for the reader.

The use of strong argument patterns could further strengthen the claims of experts. For example, Section 8.3.2 of 516C suggests testing under environmental and operating conditions, but does not explicitly ask how one knows that the proper environmental and operating conditions have been identified. While other documents might cover this material, its relation to the arguments presented is critical knowledge that could be codified to the reader's benefit.

There is considerable potential to extract and clarify the domain arguments present in 516C. However, the task of properly extracting domain argument from 516C would require direct access to experts responsible for writing and/or interpreting the 516C document.

In conclusion, both JSSG–2009 and MIL-HDB–516C could improve their presentation of "why". Rigorous argument, either in text or graphical notation, could lead to a more comprehensive capture of rationale as codified knowledge. This form might lead to better support for the community in capturing a more complete set of what, how, and why various criteria and requirements are required and should be present in a system specification.

Rigorous, codified arguments allow:

1. more rigorous recording of expert rationale to explain “why” to the reader,
2. more homogenous interpretation of text as “why” helps elucidate “how” and “what”, and
3. incremental improvement of documents with improved knowledge over time, as explicit representation of “why” allows “how” and “what” to be updated as circumstances change.

All of the above characteristics would benefit the guidance potential of 516C and JSSGs.

5 CONCLUSIONS

We developed a reference model and reference processes for systems engineering, based on well-regarded prior work from the community, that promotes careful identification of problem, requirements, context and design — yielding a set of artifacts that supports modularity by clearly identifying assumptions, guarantees, and critical context that might otherwise be overlooked. Based on this foundation, we developed a set of argument patterns that enable practitioners to quickly and authoritatively argue success of their development efforts — the most fundamental, highest goal in any development effort. Expanding on these patterns and building on prior work from the community, we advanced a practical approach to argument modularity that brings assume-guarantee reasoning into the argument and provides guidance for considering and addressing contextual compatibility at all levels of argument composition. Finally, we developed a set of theories and tools to enable precise formal analysis of component interfaces and contracts between components, moving beyond syntax-only considerations and including well-formed semantics encoded as real-world types.

Throughout the development of system-interface abstraction technology, we developed two examples that enabled us to both demonstrate and refine the technology.

The first example is based on the premise that system-interface abstraction technology, including the argument patterns described above, could be used not only to guide the development of the system but also to provide a defensible response to a request for proposals from a customer, such as the USAF. While necessarily incomplete, the example highlights the explicative power of applying argumentation at the very beginning of a development effort — even before a contract is in place.

The second example is based upon the development of a hypothetical small UAS, and is used to illustrate the identification of interfaces for components during development as well as to demonstrate the efficacy of compositional reasoning, once solid interfaces have been identified. The example additionally answers a critical question: can system-interface abstraction technology be successfully applied to control systems? In spite of the apparent tight coupling of control-system design, we nevertheless answered this question in the affirmative.

In addition to these two examples, we also applied domain-argument recovery, a technique first developed in our prior work on CLASS, to regulatory standards common in USAF acquisition efforts. Our experience underscored the importance of having domain experts available whenever regulations are in force, to ensure that the intent of the regulation is well understood by all parties.

The development of the examples used to demonstrate system-interface abstraction technology highlighted two important points:

1. the difficulty of building good challenge or example problems; and
2. the importance of having good challenge or example problems.

A well-motivated challenge problem will be rooted in an interesting and important domain, such as controls engineering or, in the case of a system of systems, in many domains, such as the collection of domains that must come together to build a UAS. For small research teams, finding the required depth and breadth of experience is very difficult. Moreover building the challenge problem is time-consuming, as there are many detailed aspects of the problem that must be demonstrated.

Having good challenge or example problems is very important, to motivate conclusions about new theoretical approaches to systems engineering. If the challenge problems do not represent sufficiently realistic systems, the conclusions reached are unlikely to be compelling. Worse, if the challenge problems are too simple, they may not demonstrate the kind of engineering challenges that the new approaches are designed to address.

System-interface abstraction technology, since it addresses challenges arising in modular system development, requires complex and realistic challenge problems. As noted above, while significant elements of the technology were successfully demonstrated using the example problems we developed, there was insufficient detail and domain experience and experience in key areas. As a result, we were unable to demonstrate some of the features of compositional reasoning in modular argumentation that are particularly important in system-interface abstraction technology. Nevertheless, the development of these example problems enabled us to identify limitations in the technology and address those limitations through further theoretical development. The resulting technology presented in this report therefore represents a culmination of a rational compositional reasoning infrastructure based our observations about the limitations of compositional reasoning discovered both during initial development of the approach and as discovered during assessment. Ideally, rational infrastructure, including both theory and mechanics, would be developed a priori and applied as prescribed. In practice, however, complex development infrastructures and methodologies provide guidance but are expected to evolve as needed when applied or in light of new discoveries [49]. Later additions to underlying theory and mechanics were not the subject of experimentation and remain to be assessed in future work. Additional research and development is therefore required, to more fully demonstrate system-interface abstraction technology.

Ideally, we would join a large-scale development effort with system and domain engineers familiar with modularity and compositional reasoning, and apply system-interface abstraction technology from problem identification through system design. Such a development effort would illustrate the approach that was presented in Section 4.1 and represent the most comprehensive demonstration possible. Our role would be to support the systems and domain engineers in the application of system-interface abstraction technology and to mediate integration of development artifacts and evidence into the arguments. As part of the activity, data would be collected so that a report on the efficacy of the approach could be produced, citing the development effort as a comprehensive case study.

As an alternative, system-interface abstraction technology could be applied *post hoc* to an existing system or an existing challenge problem that demonstrates sufficient modularity, complexity, and compositional reasoning. While such an application would be necessarily limited in the extent to which the reference model and reference processes could be applied, it would nevertheless allow a significant demonstration of the argument patterns and the development and assessment of argument modularity.

6 REFERENCES

[1] J. S. Dahmann and K. J. Baldwin, "Understanding the current state of us defense systems of systems and the implications for systems engineering," in *Systems Conference, 2008 2nd Annual IEEE*. IEEE, 2008, pp. 1–7.

[2] M. W. Maier, "Architecting principles for systems-of-systems," in *INCOSE International Symposium*, vol. 6, no. 1. Wiley Online Library, 1996, pp. 565–573.

[3] R. W. Butler and G. B. Finelli, "The infeasibility of experimental quantification of life-critical software reliability," *SIGSOFT Softw. Eng. Notes*, vol. 16, no. 5, pp. 66–76, Sep. 1991. [Online]. Available: http://doi.acm.org/10.1145/123041.123054

[4] M. Whalen, A. Gacek, D. Cofer, A. Murugesan, M. Heimdahl, and S. Rayadurgam, "Your "what" is my "how": Iteration and hierarchy in system design," *Software, IEEE*, vol. 30, no. 2, pp. 54–60, March 2013.

[5] J. Rushby, "New challenges in certification for aircraft software," in *Proceedings of the Ninth ACM International Conference on Embedded Software*, ser. EMSOFT '11. plus 0.5em minus 0.4emNew York, NY, USA: ACM, 2011, pp. 211–218. [Online]. Available: http://doi.acm.org/10.1145/2038642.2038675

[6] D. Cofer, "Compositional analysis of avionics architectures in AADL," 2012.

[7] E. A. Strunk and J. C. Knight, "The essential synthesis of problem frames and assurance cases," *Expert Systems*, vol. 25, no. 1, pp. 9–27, 2008.

[8] M. Jackson, *Problem Frames: Analyzing and Structuring Software Development Problems*. plus 0.5em minus 0.4emBoston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[9] E. Strunk, C. Furia, M. Rossi, J. Knight, and D. Madrioli, "The engineering roles of requirements and specification," University of Virginia, Department of Computer Science, Tech. Rep. CS-2006-21, 2006.

[10] P. J. Graydon, "Assurance based development," Ph.D. dissertation, Charlottesville, VA, USA, 2010.

[11] S. E. Toulmin, *The uses of argument*. Cambridge University Press, 2003.

[12] R. Hawkins, T. Kelly, J. Knight, and P. Graydon, "A new approach to creating clear safety arguments," in *Advances in Systems Safety*, C. Dale and T. Anderson, Eds. Springer, 2011, pp. 3–23.

[13] C. Y. Baldwin and K. B. Clark, *Modularity in the design of complex engineering systems*. Springer, 2006.

[14] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[15] J. Knight, J. Rowanhill, M. A. Aiello, and K. Wasson, "A comprehensive safety lifecycle," in *International Conference on Computer Safety, Reliability, and Security*. Springer, 2015, pp. 38–49.

[16] J. Knight. J. Rowanhill, "Domain arguments in safety critical software development," in *27th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, October

2016.

[17] J. Rowanhill, “Class server toolset: Design and implementation,” Dependable Computing, Tech. Rep. TR-2016-01, May 2016.

[18] J. Rowanhill and J. C. Knight, “Class assurance knowledge ecology,” Dependable Computing, Tech. Rep. TR-2015-1, May 2016.

[19] J. R. John C. Knight, “The indispensable role of rationale in safety standards,” in *International Conference on Computer Safety, Reliability and Security (SAFECOMP)*, September 2016.

[20] J. Knight, J. Rowanhill, U. Ferrell, A. Bateman, and N. Gandhi, “Integrating an assurance case into do-178b compliant software development,” in *2015 IEEE/AIAA 34th Digital Avionics Systems Conference (DASC)*. IEEE, 2015, pp. 1–22.

[21] J. Knight, J. Rowanhill, and J. Xiang, “A safety condition monitoring system,” in *International Conference on Computer Safety, Reliability, and Security*. Springer, 2015, pp. 83–94.

[22] Ministry of Defence, “Defence standard 00-56 issue 4: Safety management requirements for defence systems,” 2007.

[23] K. Attwood, P. Chinneck, M. Clarke, G. Cleland, M. Coates, T. Cockram, G. Despotou, L. Emmet, J. Fenn, B. Gorry *et al.*, “GSN community standard version 1,” *Origin Consulting Limited, York*, November 2011.

[24] P. J. Graydon, J. C. Knight, and E. A. Strunk, “Assurance based development of critical systems,” in *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN’07)*, June 2007, pp. 347–357.

[25] P. J. Graydon and J. C. Knight, “Software process synthesis in assurance based development of dependable systems,” in *European Dependable Computing Conference (EDCC)*. IEEE, 2010, pp. 75–84.

[26] IEEE Computer Society Software Engineering Standards Committee and IEEE-SA Standards Board, “Ieee recommended practice for software requirements specifications.” Institute of Electrical and Electronics Engineers, 1998.

[27] IEEE Computer Society Software Engineering Standards Committee and IEEE-SA Standards Board, “Iso/iec/ieee international standard - systems and software engineering – life cycle processes –requirements engineering.” Institute of Electrical and Electronics Engineers, 2011.

[28] A. Ayoub, B. Kim, I. Lee, and O. Sokolsky, “A systematic approach to justifying sufficient confidence in software safety arguments,” in *Computer Safety, Reliability, and Security*. Springer, 2012, pp. 305–316.

[29] R. Hawkins, K. Clegg, R. Alexander, and T. Kelly, “Using a software safety argument pattern catalogue: Two case studies,” in *Proceedings of the 30th International Conference on Computer Safety, Reliability, and Security*, ser. SAFECOMP’11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 185–198. [Online]. Available: http://dl.acm.org/citation.cfm?id=2041619.2041640

[30] R. Hawkins and T. Kelly, “A systematic approach for developing software safety

arguments," *Hazard prevention*, vol. 46, no. 4, p. 25, 2010.

[31] B. D. Rodes, J. C. Knight, and K. S. Wasson, "A security metric based on security arguments," in *Proceedings of the 5th International Workshop on Emerging Trends in Software Metrics*. ACM, 2014, pp. 66–72.

[32] B. D. Rodes and J. C. Knight, "Speculative software modification and its use in securing soup," in *Dependable Computing Conference (EDCC), 2014 Tenth European*. IEEE, 2014, pp. 210–221.

[33] B. Rodes and J. Knight, "Reasoning about software security enhancements using security cases," in *The First International Workshop on Assurance for Argument and Agreement (AAA)*, 2013.

[34] B. D. Rodes, "Speculative software modification," Ph.D. dissertation, University of Virginia, 2015.

[35] T. P. Kelly, "Concepts and principles of compositional safety case construction," *Contract Research Report for QinetiQ COMSA/2001/1/1*, 2001.

[36] J. Fenn, R. Hawkins, P. Williams, T. Kelly, M. Banner, and Y. Oakshott, "The who, where, how, why and when of modular and incremental certification," in *System Safety, 2007 2nd Institution of Engineering and Technology International Conference on*, Oct 2007, pp. 135–140.

[37] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, *ECOOP'97 — Object-Oriented Programming: 11th European Conference Jyväskylä, Finland, June 9–13, 1997 Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, ch. Aspect-oriented programming, pp. 220–242. [Online]. Available: http://dx.doi.org/10.1007/BFb0053381

[38] Industrial Avionics Working Group (IAWG). (2012) Modular software safety case process description. [Online]. Available: https://www.amsderisc.com/related-programmes/?doing\s\do5(w)p\s\do5(c)ron=1459350756.341255903244018554687 5

[39] Object Management Group (OMG). (2011) Business process model notation (bpmn) version 2.0 (2011). [Online]. Available: http://www. omg. org/spec/BPMN/2.0

[40] J. Rushby, " Logic and epistemology in safety cases," in *Computer Safety, Reliability, and Security*, ser. Lecture Notes in Computer Science, F. Bitsch, J. Guiochet, and M. Kaâniche, Eds. Springer Berlin Heidelberg, 2013, vol. 8153, pp. 1–7. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-40793-2\s\do5(1) t

[41] Safety certification of software-intensive systems with reusable components. [Online]. Available: http://www.safecer.eu

[42] Modular software safety cases. [Online]. Available: http://capability-agility.co.uk

[43] J. Rushby, *Modular certification*. National Aeronautics and Space Administration, Langley Research Center, 2002.

[44] P. Graydon and I. Bate, "The nature and content of safety contracts: Challenges and suggestions for a way forward," in *Dependable Computing (PRDC), 2014 IEEE 20th Pacific Rim International Symposium on*, Nov 2014, pp. 135–144.

[45] J. Fenn, R. Hawkins, P. Williams, and T. Kelly, “Safety case composition using contracts-refinements based on feedback from an industrial case study,” in *The Safety of Systems*. Springer, 2007, pp. 133–146.

[46] P. Graydon and C. Holloway, ‘‘‘‘evidence” under a magnifying glass: Thoughts on safety argument epistemology,” 2015.

[47] R. Banach and M. Poppleton, “Retrenchment: An engineering variation on refinement,” in *B’98: Recent Advances in the Development and Use of the B Method*. Springer, 1998, pp. 129–147.

[48] A. B. Hocking, J. Knight, M. Aiello, and S. Shiraishi, “Proving model equivalence in model based design,” in *Software Reliability Engineering Workshops (ISSREW), 2014 IEEE International Symposium on*, Nov 2014, pp. 18–21.

[49] D. L. Parnas and P. C. Clements, “A rational design process: How and why to fake it,” *IEEE Transactions on Software Engineering*, vol. SE-12, no. 2, pp. 251–257, Feb 1986.

[50] T. P. Kelly, “Arguing safety — a systematic approach to safety case management,” Ph.D. dissertation, University of York, 1999.

[51] MoD, “Defence standard 00-56 issue 4. safety management requirements for defence systems: Part 1 requirements,” UK Ministry of Defence, Tech. Rep., 2007.

[52] T. Kelly and R. Weaver, “The goal structuring notation – a safety argument notation,” in *Proc. of Dependable Systems and Networks 2004 Workshop on Assurance Cases*, 2004.

[53] D. B. Owens, D. E. Cox, and E. A. Morelli, “Development of a low-cost sub-scale aircraft for flight research: The faser project,” in *25th AIAA Aerodynamic Measurement Technology and Ground Testing Conference*, no. 2006-3306, 2006.

[54] R. J. Meinhold and N. D. Singpurwalla, “Understanding the Kalman filter,” *The American Statistician*, vol. 32, no. 2, pp. 123–127, May 1983.

[55] S. J. Julier and J. K. Uhlmann, “Unscented filtering and nonlinear estimation,” *Proceedings of the IEEE*, vol. 92, no. 3, pp. 401–422, Mar. 2004.

List of Acronyms

ACP	Assurance Claim Point
ABD	Assurance Based Development
BPMN2	Business Process and Model Notation 2
CLASS	Comprehensive Lifecycle Assurance for System Safety
CONOPS	Concept of Operations
CTS	Cooling Tank System
GSN	Goal Structuring Notation
IAWG	Industrial Avionics Working Group
I/O	Input/Output
IMU	Inertial Measurement Unit
JSSG	Joint Service Specification Guide
LTI	Linear Time Invariant
MSSC	Modular Software Safety Case
NED	North, East, Down
OAT	Outside Air Temperature
RFP	Request for Proposals
SIAT	System Interface Abstraction Technology
SoS	System of Systems
SUAS	Small Unmanned Aircraft System
TCC	Type Correctness Condition
UAS	Unmanned Aircraft System
UKF	Unscented Kalman Filter

APPENDIX A ASSURANCE-CASE TECHNOLOGY

A.1 Background

When engineering software systems, developers have the burden of demonstrating assurance that the software establishes properties and characteristics desired by the system stakeholders. For example, stakeholders might require assurance that the software is adequately safe, secure, reliable, etc. for its use (i.e., for its operating context). A software system is said to be acceptable for its operating context if adequate assurance is demonstrated that the system has the stakeholder-desired property or properties.

The complexity and size of modern software, however, makes providing definitive, complete, and irrefutable proof that any given software system is acceptable impractical for all but the most basic and trivial systems. In practice, developers rely on any available evidence to demonstrate that the existence of desired properties is "highly probable", although quantification of such probabilities is often impossible. Consequently, interpretation of what constitutes highly probable assurance is left to intuition.

Evidence alone does not provide a justification that a given software system is acceptable for its specific operating context. "Argument without supporting evidence is unfounded, and therefore unconvincing. Evidence without argument is unexplained – it can be unclear that (or how) safety objectives have been satisfied" [50]. Rather, evidence must be explained and interpreted to demonstrate assurance. While a prescribed standard can be used to explain evidence, if a standard is available, standards are often:

- inflexible to the specific needs of stakeholders, i.e., standards do not take into consideration the characteristics of a specific software system and its operating context,
- rarely contain explicit rationales explaining why they demonstrate acceptable assurance, and therefore prohibit deeper understanding of the assurance any given standard is meant to provide, and
- rely upon the assumption that adherence to a given standard results in an acceptable system in all uses of the standard.

A goal-based approach to explaining evidence allows developers to overcome the limitations of prescriptive standards. In a goal-based approach, claims about the properties of a software system are made based on the needs specific to the system stakeholders. Evidence is then used to support the specified claims. The key benefit of this approach is that assurance is tailored to a specific system. Consequently, developers must take a more active role in defining what evidence should be collected and rationalizing how that evidence supports claims.

A *safety case* organizes a goal structure as an argument, supported by a body of evidence that provides a compelling, comprehensible and valid case that a system is safe for a given application in a given operating environment [51]. The safety case concept is generalizable and applicable for the purposes of generating an explicit rationale for belief in any system characteristic of interest (e.g., system functionality, performance, security, or reliability). The generalized concept is referred to as an *assurance case*.

A.1.1 Elements of an Assurance Case

The general elements of an assurance case (see Figure A-1) are:

- A set of **goals** in which a top-level goal documents the main assurance claim and other sub-goals help to structure and elaborate the argument. Goal statements have the form of a proposition, *e.g.*, "The system is safe," in order that they may be evaluated as true or false.
- A definition of the **context** within which the top-level goal is expected to hold. Context includes everything that might be needed to constrain the applicability of the argument to a given set of circumstances, including assumptions.
- A collection of supporting **evidence** that includes results from inspections, analyses, testing, and simulation estimating fundamental system properties, as well as process-based information such as standards compliance and maturity level of the development organization. This evidence forms the basis from which assurance can be argued.
- An explicit **argument** that shows how the overall claim (goal) can be reasonably inferred from the supporting evidence. In practice, multiple different argument strategies are used in conjunction to argue assurance in a given case.

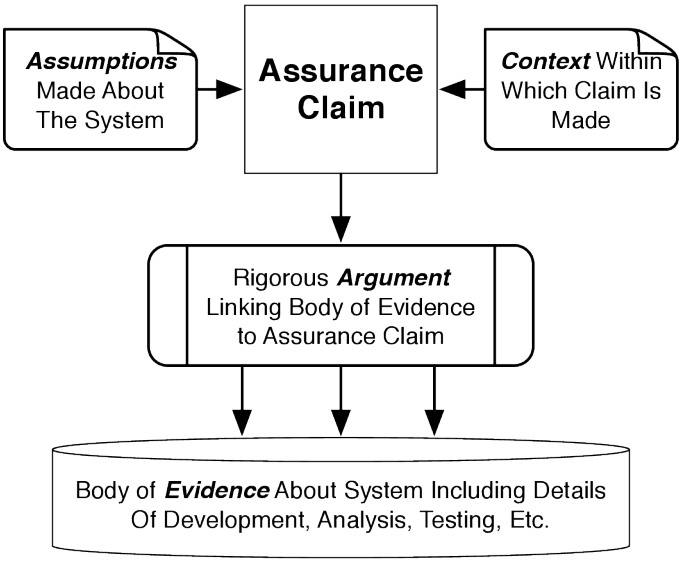


Figure A-1: Major Elements of an Assurance Case

A.1.2 The Goal Structuring Notation

While in principle an assurance case can be documented in any form, including natural language and tables [52], an increasingly popular and effective documentation method is to structure the argument using a graphical notation. In a graphical notation, components of the argument, such as claims and evidence, are represented as nodes in a graph. The connections/relationships between these nodes illustrate how evidence supports claims and thereby forms an assurance argument.

The Goal Structuring Notation (GSN) [23] is a graphical language that provides a rich set of syntax and semantics for documenting assurance arguments. It enables representation of the logical relationships among the basic elements of assurance cases, as well as the documentation of supporting information to contextualize this logic. It is accessible to readers of a wide variety of backgrounds and expertise, enabling a common communication mechanism for safety

argumentation and audit. It is also supported by editing tools, and is amenable to some automated analysis.

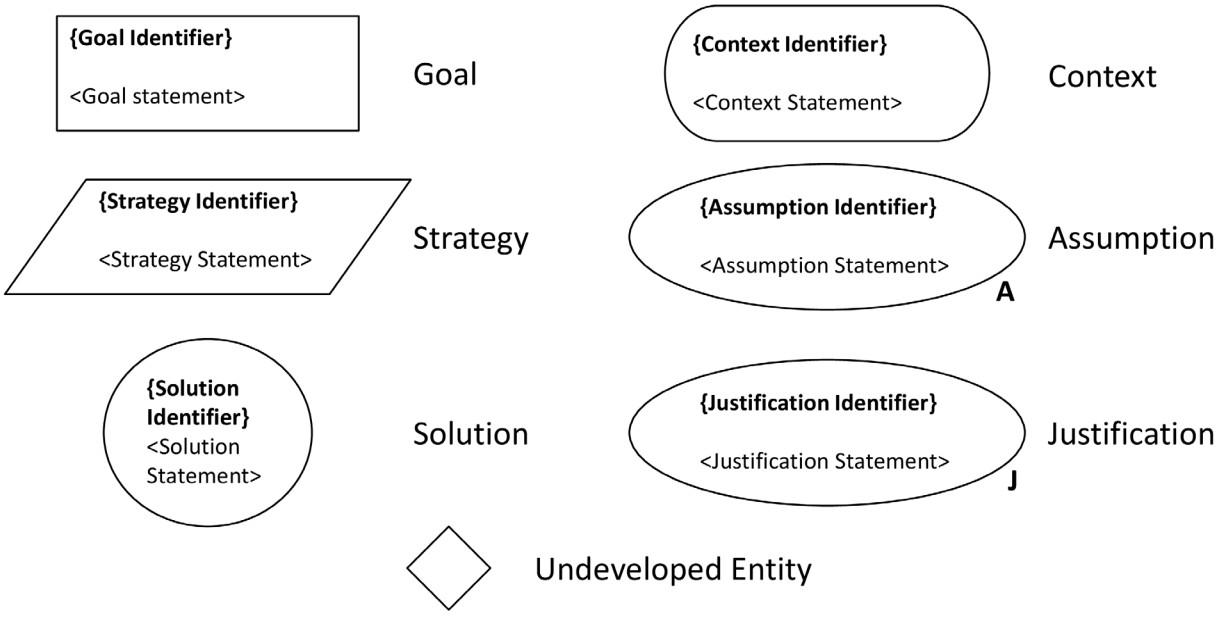


Figure A-2: GSN Elements

The core GSN elements, shown in Figure A-2, are:

- **Goals**: Depicted as a rectangle, a goal documents a claim about a property or a characteristic that a software system is said to have. Each assurance argument contains a top-level goal, which is the conclusion the argument is meant to support. The top-level goal is subdivided hierarchically into sub-goals. Sub-goals are refinements/simplifications of higher-level goals, and represent claims about a more specific sub-system or property of the larger software system. Goals are also referred to as claims.
- **Contexts**: Depicted as an oval, a context provides a reference to contextualizing information and documentation. For example, a context can refer to limitations about the scope of a goal. A context is linked to the argument element requiring contextualization, and all sub-arguments from that element inherit the context.
- **Assumptions**: Depicted as an oval with the letter 'A' at the bottom-right, an assumption is a specialized context element used to present intentionally unsubstantiated statements.
- **Justifications**: Depicted as an oval with the letter 'J' at the bottom-right, a justification is a specialized context element used to provide a rationale for a component of the argument.
- **Strategies**: Depicted as a parallelogram, an argument strategy describes the inference between a goal and its sub-goals.
- **Solutions**: Depicted as a circle, a solution is used to reference evidence in direct support of a goal. Solutions are also referred to as evidence.
- **Undeveloped**: Entities depicted as a hollow diamond, the undeveloped entity symbol is directly placed on any of the above argument elements to indicate that the element is

intentionally left undeveloped by argument engineers. Undeveloped argument entities signify further review and examination of the undeveloped element is necessary.

GSN also provides modular extensions to represent interrelated modules of argument, as well as support for representing patterns that abstract argument form and content into reusable structures. A complete description of GSN can be found in the GSN community standard [23].

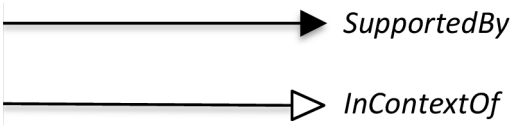


Figure A-3: GSN Element Relationships

GSN elements are connected by one of two types of relationships, represented by arrows, shown in Figure A-3:

- **SupportedBy**: Depicted as a solid (closed) arrow, a SupportedBy relationship indicates inferential and evidential support, e.g., relationships between strategies, goals, and evidence. The arrow points to the supporting argument element.
- **InContextOf**: Depicted as a hollow (open) arrow, an InContextOf relationship indicates a contextualizing relationships where the arrow points to the argument element providing the contextualization, e.g., a context, a justification, or an assumption.

In addition to these core entities and relationships, GSN also provides notational support for expressing argument modularity (including module and contract entities) and argument patterns. For further details of these notational concepts, readers are referred to the GSN community standard [23]. Figure A-4 provides an illustration of the application of the GSN.

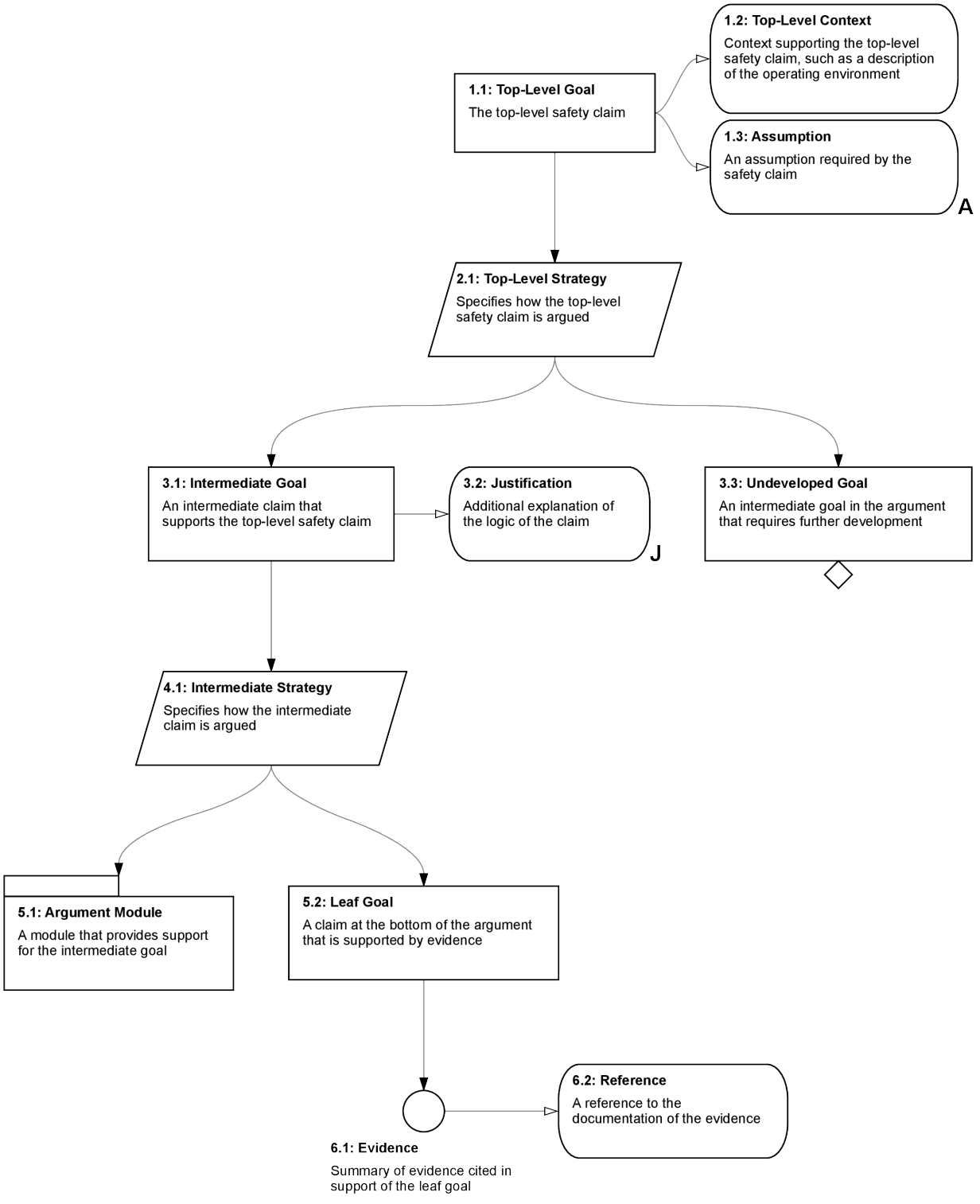


Figure A-4: Example Argument in GSN

A.1.3 Confidence

In general, belief in the top-level goal (claim) of an argument, whether for safety, security or another property is important. The premises upon which an argument is accepted are:

1. The top-level goal defines the stakeholders' needs adequately.
2. The belief in the truth of the top-level goal is justified by the argument.

Although belief in the top-level goal rests on confidence in the associated argument, often there are repeating underlying arguments. Hawkins et al. have introduced the notion of confidence arguments to supplement safety and security arguments in order to capture and separate confidence assessment within assurance arguments [12]. A confidence argument supplements a traditional argument and supplies the rationale for belief in the quality of each of the argument's items of evidence, context definitions, and inferences. When these elements are added to an argument, there is an assertion that the element is valid and correct, i.e., the element serves the intended purpose to support a claim. Assertions in the argument are linked via an Assurance Claim Point (ACP) to a separate confidence argument where confidence in the assertion is argued. Since these concerns repeat any time these entities are added to the argument, separating these confidence arguments has the claimed benefit of simplify arguments and providing clarity of purpose.

For this effort, we adopt a custom ACP notation, illustrated in Figure A-5.

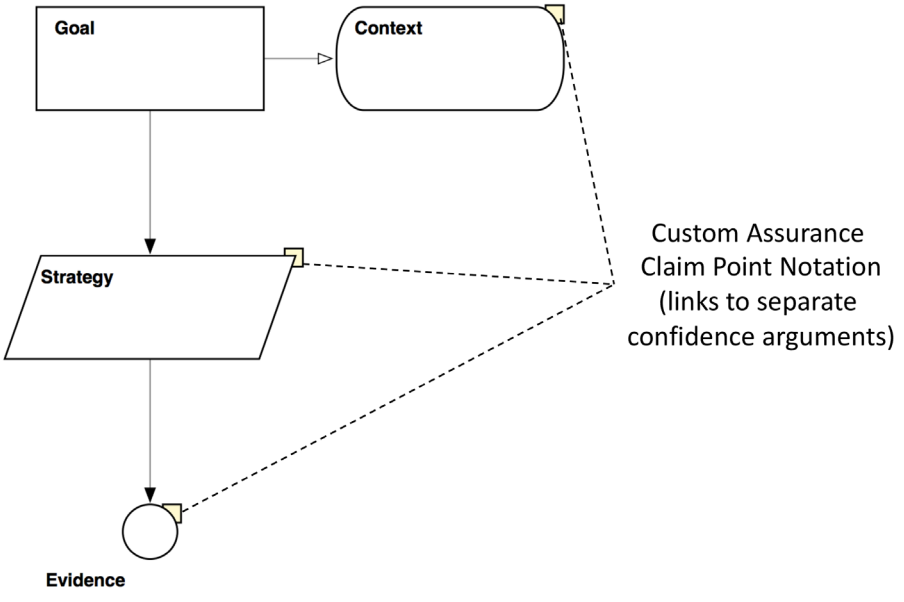


Figure A-5: Custom Assurance Claim Point Notation

A.1.4 Understanding Argument

Using GSN, an argument is constructed as a graphical and hierarchical structure, where a top-level goal is subdivided recursively until a goal can be directly justified by available evidence. When a goal is subdivided, justified by evidence, or contextualized, an inference is made about the relationship between argument elements. Ideally, all inferences within an argument would be

based on deductive reasoning. In deductive logic, if the premises are true, then the conclusion is necessarily true. If an argument is completely deductive, the argument could serve as a proof supporting the top-level goal. In practice, however, application of deductive logic in support of claims about real-world systems is not always possible. Consequently, arguments about software system properties rely primarily on *inductive reasoning*.

In inductive logic, if the premises are true, the conclusion is "*likely*" true. The argument does not offer irrefutable proof that a top-level goal is valid. The likelihood that the top-level goal is valid is based on a careful and systematic examination of all risks and available evidence. Because arguments must be constructed carefully and then thoroughly examined, these arguments are often referred to as *rigorous arguments*.

Since assurance arguments are inductive, they are also provisional and subject to revision as new information becomes available. An assurance argument is said to be *defeasible*: future evidence could refute a claim (such evidence is typically referred to as a *defeater*). The defeasibility of an assurance argument stems from two primary sources of doubt [40]:

- **Inferential doubt**: Doubts about the accuracy of the reasoning used in the argument, i.e., doubt that each step of logical inference follows to justify a top-level goal. These are doubts about the validity of relationships between argument elements.
- **Epistemic doubt**: Doubts about the completeness and accuracy of the knowledge about the system. This knowledge takes the form of evidence, supporting documentation, and generally any information referenced within the argument.

Although an assurance argument cannot typically be used as proof about a top-level claim about a software system, the primary benefit of assurance arguments is the explicit documentation of reasoning and rationale for why a system is considered acceptable. All systems determined to be acceptable for use rely on some form of argument, even if that argument is implicit. By making the argument explicit, assurance arguments facilitate active scrutiny and criticism. The argument can be challenged and reviewed exhaustively, supporting a more structured approach for finding flaws/weaknesses in the systems. As the system is updated, either in response to a found flaw or new functional needs, the argument is also updated, allowing developers and stakeholders to understand the impact of alterations and to determine if further changes to the software are necessary. The adequacy of the argument is context specific and ultimately determined by the needs and opinions of the system stakeholders.

APPENDIX B COOLING TANK(S) CHALLENGE PROBLEM

B.1 CONOPS

An industrial facility has a need to cool a liquid as one stage in a process. The facility would like to have one or more cooling tanks added in the middle of the production line. Prior to the cooling tank(s), the liquid is held in a reservoir that is large enough to be considered to always have liquid available to be moved into the cooling tanks. The liquid will need to be pumped from the reservoir into the cooling tank system and the pump will be considered part of the cooling tank system. After the liquid has been cooled to the appropriate temperature range, it must be sent to the next stage of the process via a production line. There is an emergency dumping line that liquid may be sent to in the case of emergencies.

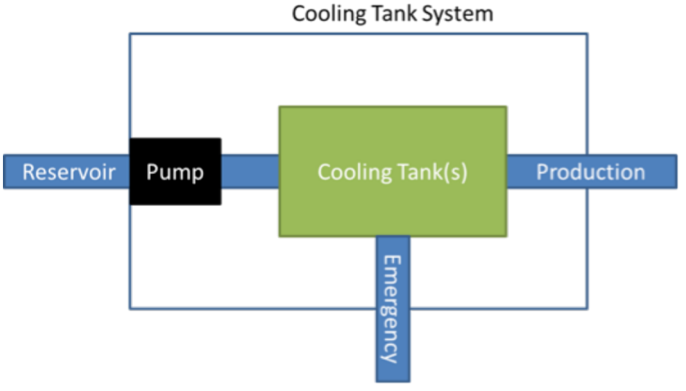


Figure B-1: CONOPS High-Level Cooling Tank System

High-level requirements:

1. The cooling tank system (CTS) shall contain a pump to transfer liquid from a reservoir to the CTS.

a. It may be assumed that the reservoir has an unlimited supply of liquid.

2. The CTS shall contain at least one tank where the liquid will be cooled.

a. NOTE: Temperature is being abstracted away in this challenge problem. A large assumption is being made that the liquid gets "appropriately cooled" by being inside of the cooling tank(s) of the CTS.

3. The CTS shall use a production liquid line to send cooled liquid to the next stage of the process.

a. The production line may accept up to 0.2 m3/s of cooled liquid.

b. It may be assumed that the production line never gets backed up (i.e. cooled liquid may always be passed on to the next stage).

4. The CTS shall dump liquid into an emergency line in the case of a safety or other emergency.

a. The emergency line may accept up to 0.5 m3/s of liquid.

b. It may be assumed that the emergency line never gets backed up (i.e. liquid may always be dumped into the emergency line).

5. The CTS shall not cause any unsafe situations for the workers in the facility.

a. No liquid may leave the CTS except through the production or emergency lines.

APPENDIX C ULTRA STICK

C.1 Introduction

This report describes the development of a simulation environment to support research related to system of systems interactions. The environment includes a simulation model of a small unmanned air vehicle, sensor models, a state estimator, and a control system capable of executing a representative SUAS mission. Section C.2 describes the selected mission scenario and Section C.3 provides an overview of the demonstration environment. Section C.4 provides a brief description of the physical vehicle modeled by the simulation, Sections C.5 and C.6 describe the inner- and outer-loop control system, respectively, and Section C.7 describes the path planner and guidance components. The sensor package on which sensor models is based is described in Section C.8, and the filtering and state estimation approaches that were investigated are described in Section C.9 . Section C.10 presents performance results for the system and results on how changing sensor performance characteristics impact system performance.

C.2 Mission Scenario

The demonstration environment is built around a common UAS mission of gathering imagery of a region on the ground. A low-cost UAV configuration with a fixed imaging sensor on a fixed wing UAV is assumed. The camera is assumed to have a square 90° field-of-view, so that when the aircraft is level over flat ground the imaging sensor captures a square region on the ground that extends toward the nose, tail and each wingtip a distance equal to the altitude of the aircraft. The camera captures images every five seconds, and the nominal flight pattern provides 20m of overlap in the lateral direction between images. At the nominal cruise speed of 25m/s, the nominal overlap between images in the longitudinal direction is approximately 160 m. Distortion of the photographs is considered unacceptable if the role or pitch angle exceeds 10° when the image is taken, and such images are discarded. Resolution of photographs taken from more than 15m above the desired altitude is considered unacceptable, and such images are also discarded. Coverage of the region being imaged is assessed by establishing a 1 m square grid over the field and testing whether each grid point is captured in at least one image. Figure C-1 shows the specific mission used in the demonstration environment, which captures a ground region that is a 1km by 2km rectangle. The mission begins from the center of the region being captured, the vehicle climbs to altitude and then executes four passes over the field to provide the desired coverage and overlap.

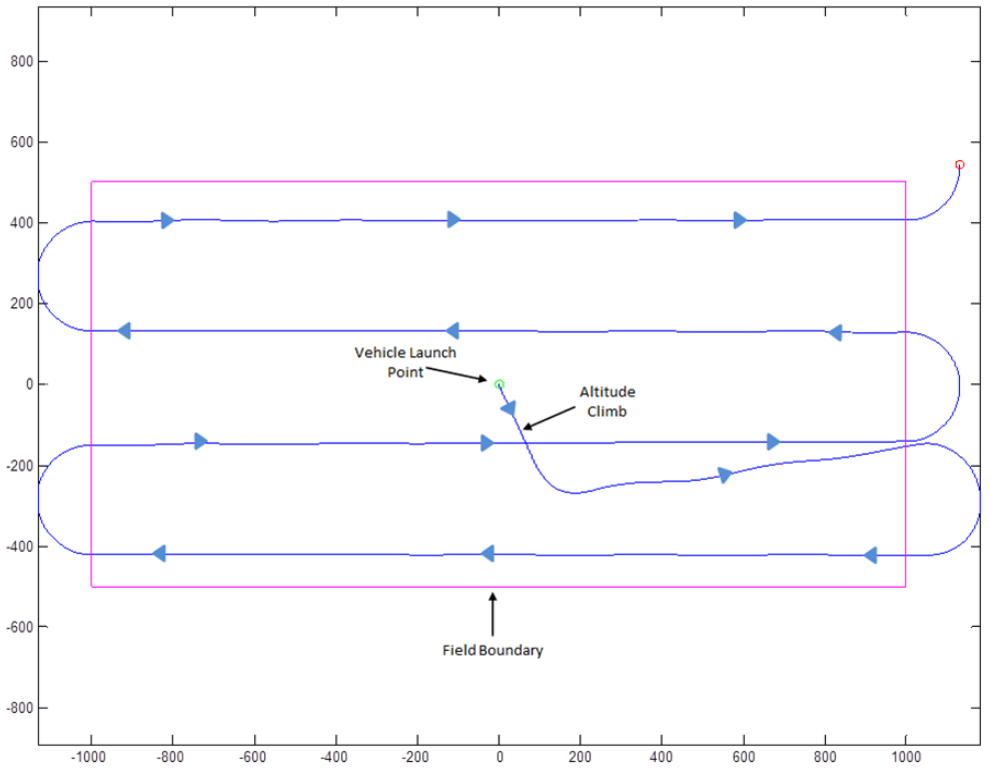


Figure C-1: Example Mission Ground Track

C.3 Overview of Demonstration Environment

An overview of the demonstration environment is shown in Figure C-2. Main components in the demonstration environment are

- Ultrastick simulation: air vehicle simulation that includes rigid body dynamics, actuator models, and sensor models.
- State estimation: estimates vehicle states based on sensor inputs.
- Inner loop control: generates aerodynamic surface commands to achieve commanded Euler angles.
- Outer loop control: generates inner loop control commands to track desired altitude, heading, and velocity.
- Path planner: generates altitude, heading, and velocity commands to achieve a path specified by a set of waypoints.
- Mission planner: in the current implementation, the mission planner is designed only to generate a set of waypoints to achieve the imaging mission shown in Figure C-1.

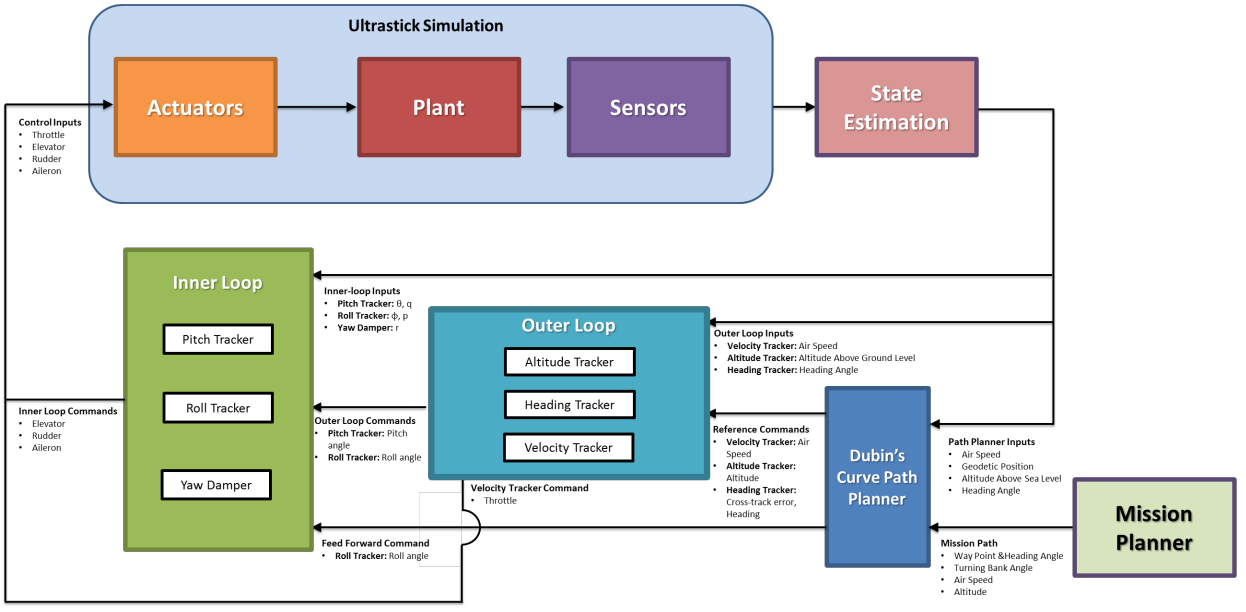


Figure C-2: High-level View of Demonstration Environment

C.4 Air Vehicle

The simulation testbed is built around a simulation model of an Ultrastick fixed wing unmanned air vehicle. Initial development of the simulation model was performed by NASA Langley [53], and work has been continued by the UAV laboratory at the University of Minnesota. Both wind tunnel and flight test experiments have been conducted as part of prior research activities to support work including validation of a nonlinear simulation model of the vehicle. A picture of the flight vehicle used by NASA in developing this simulation is shown in Figure C-3. The "stick" aircraft are a series of low-cost, commercially-available hobbyist aircraft. While the Ultrastick 120 used in the original modeling effort is out of production, very similar aircraft remain on the market. The selected vehicle thus offers a unique combination of a validated simulation model that has been made publicly available, and low-cost airframes that can be readily obtained if flight testing is desired in future phases of the work.

Figure C-3: NASA Flight Vehicle (Image Reproduced from [53])

C.5 Inner-loop Control

The outer loop Θ_{com} and φ_{com} commands are fed into the inner loop control along with a feed forward φ command.

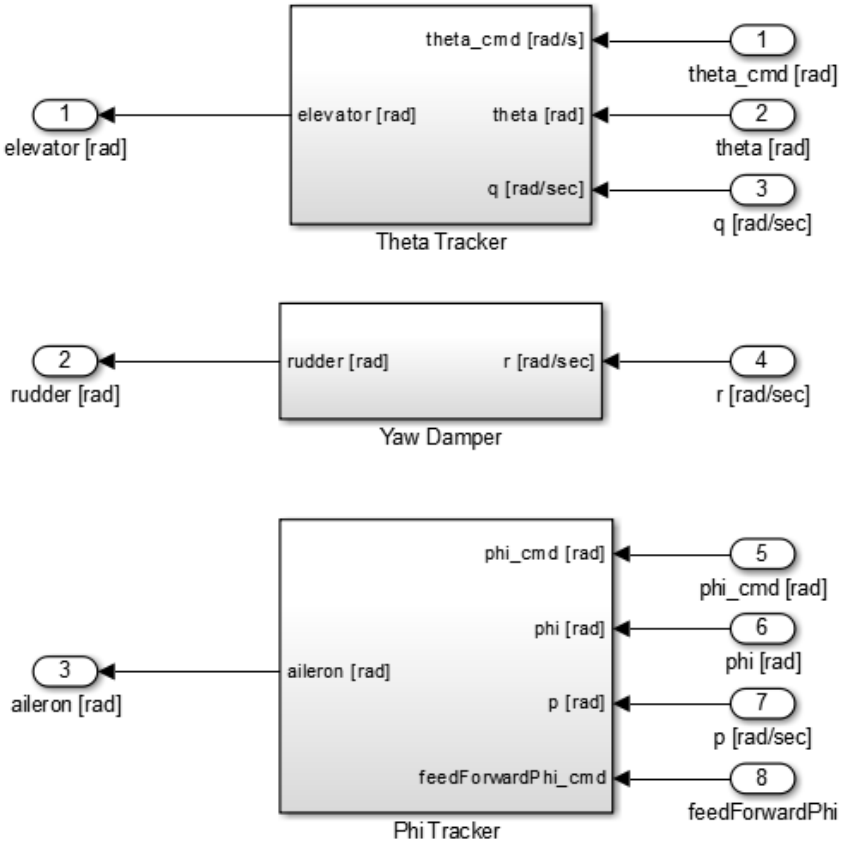


Figure C-4: Inner Loop

C.5.1 Pitch Tracker

The pitch tracker generates an elevator command based on the error between the desired and actual pitch. It accomplishes this via a PID controller with an integrator anti-windup. A system response to a 15° increase in pitch angle can be seen in Figure C-6.

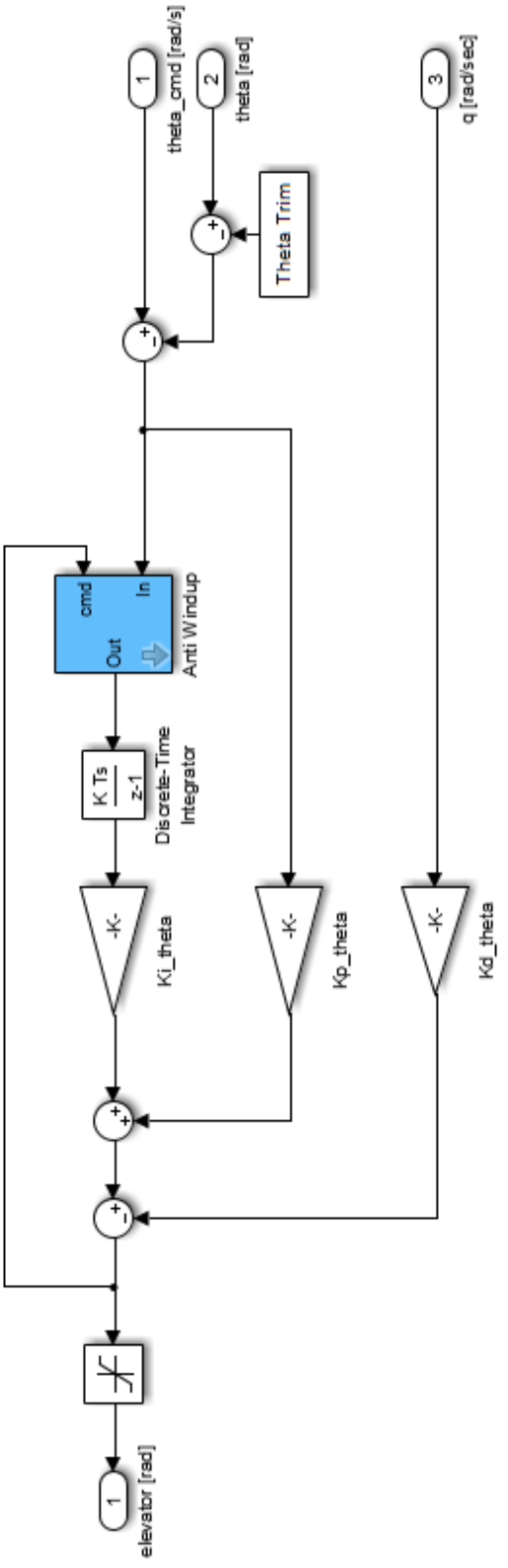


Figure C-5: Pitch Tracker

Table C-1: Inputs of the Pitch Tracker

Inputs	Units	Variable Name	Value Origin
θ	rad	theta	Sensor
θ_{com}	rad	theta_cmd	Outer Loop Control
Angular Rate about body y-axis	rad/s	q	Sensor

Table C-2: Outputs of the Pitch Tracker

Outputs	Units	Variable Name	Destination
Elevator Command	rad	elevator	Actuators

Table C-3: Control Parameters for Pitch Tracker

Control Parameter	Variable Name	Value
Proportional Gain	Kp_theta	-1.0
Integrator Gain	Ki_theta	-1.0
Derivative Gain	Kd_theta	-0.1

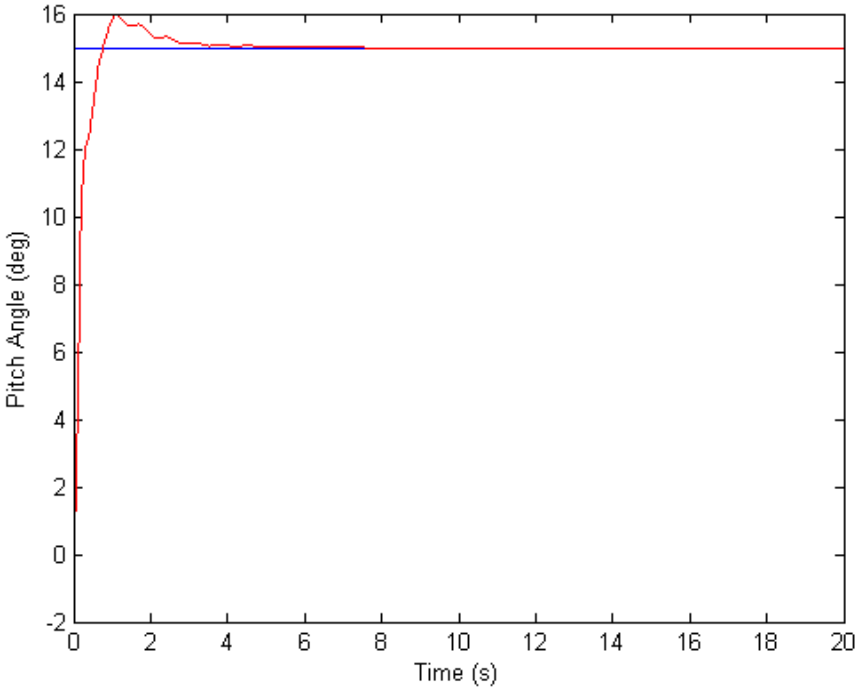


Figure C-6: Theta Response

C.5.2 Roll Tracker

The roll tracker modulates the vehicle's ailerons to track the desired roll angle. The commanded φ comes from the yaw tracker in the outer loop. A PD controller is used to control the feedback portion of this control loop. The feed forward φ command is the reference roll angle from the path planner and is used here in lieu of integral compensation to reduce steady state error and provide lead. Figure C-8 demonstrates the benefit of the feed forward command. Figures C-8(a) and C-8(c) do not have a feed forward term, while Figures C-8(b) and C-8(d) do. The blue line represents the reference value, and the red represents the actual value. As demonstrated, the vehicle is able to track better with a feed forward term.

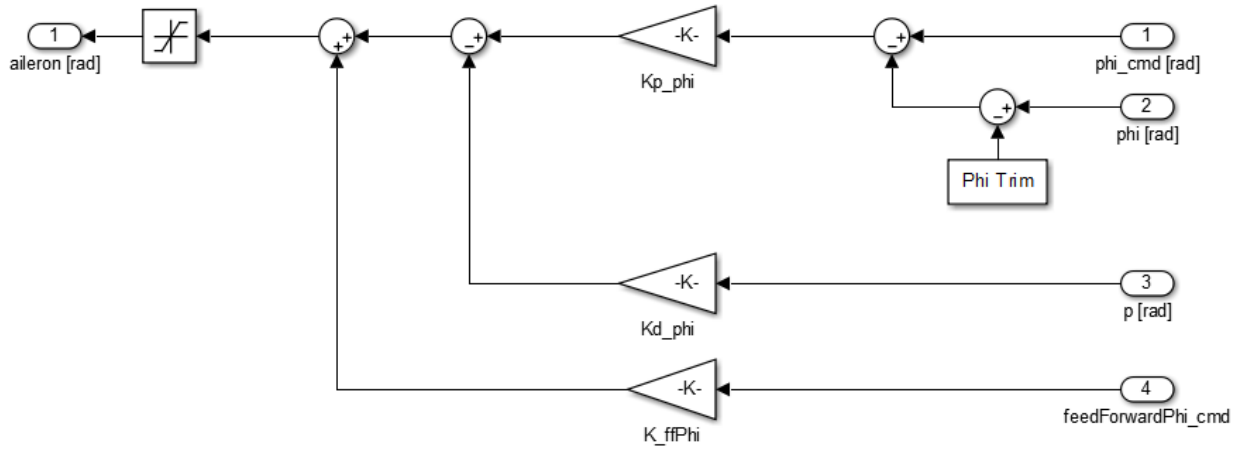


Figure C-7: Roll Tracker

Table C-4: Inputs of the Roll Tracker

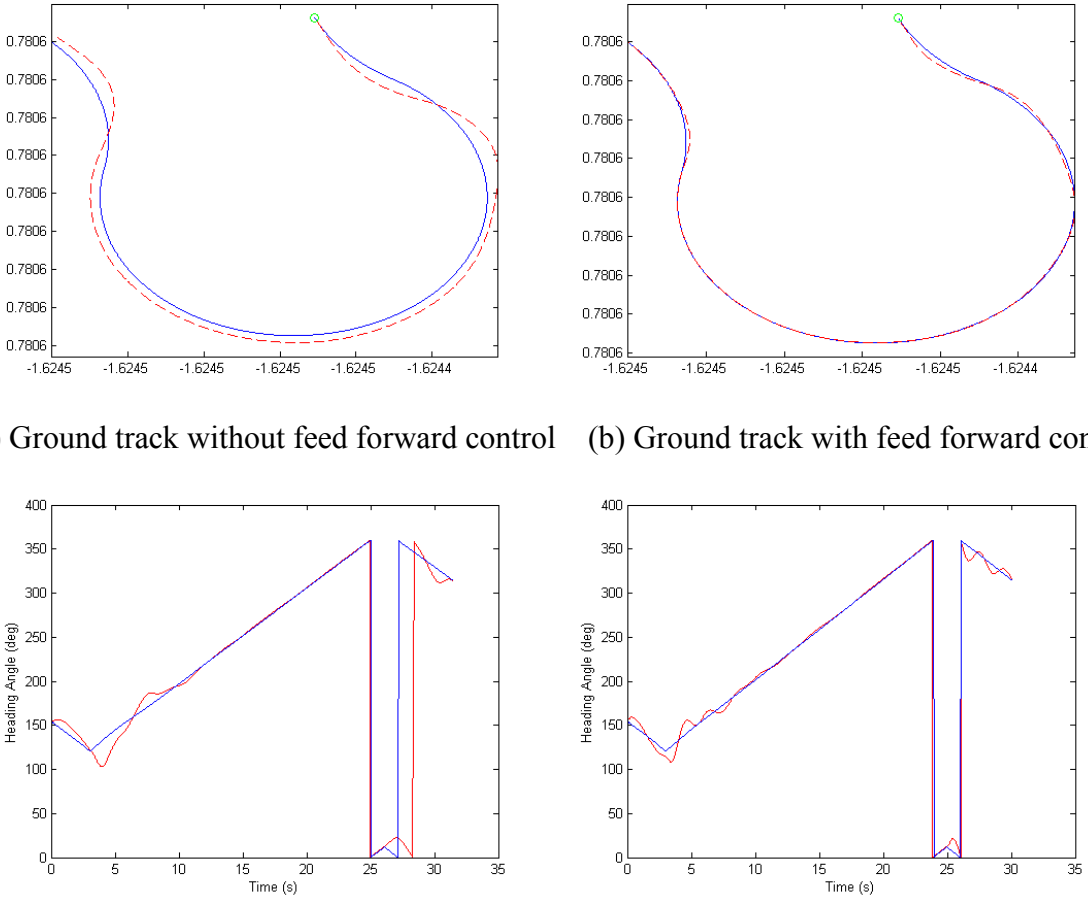
Inputs	Units	Variable Name	Value Origin
φ	rad	phi	Sensor
φ_{com}	rad	phi_cmd	Outer Loop Control
Angular Rate about body x-axis	rad/s	p	Sensor
$\varphi_{feedforward}$	rad	feedForwardPhi_cmd	Dubin's Car

Table C-5: Outputs of the Roll Tracker

Outputs	Units	Variable Name	Destination
Aileron Command	rad	aileron	Actuators

Table C-6: Control Parameters for Roll Tracker

Control Parameter	Variable Name	Value
Proportional Gain	Kp_phi	-0.7
Derivative Gain	Kd_phi	-0.1
Feed Forward Gain	K_ffPhi	-0.7



(a) Ground track without feed forward control (b) Ground track with feed forward control

(c) Heading angle without feed forward control (d) Heading angle with feed forward control

Figure C-8: Cross Track Correction Example

C.5.3 Yaw Damper

The yaw damper employs the first order discreet transfer function in Equation C-1 to dampen the yaw rate via the rudder.

$$\frac{Y(z)}{G(z)} \frac{0.065z-0.065}{z-.96079} \tag{C-1}$$

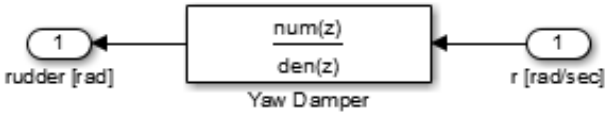


Figure C-9: Yaw Damper

Table C-7: Inputs of the Yaw Damper

Inputs	Units	Variable Name	Value Origin
Angular Rate about body z-axis	rad/s	r	Sensor

Table C-8: Outputs of the Yaw Damper

Outputs	Units	Variable Name	Destination
Rudder Command	rad	rudder	Actuators

C.6 Outer-loop Control

The outer loop control tracks the velocity, altitude, and yaw angle of the vehicle. Note that for small angle of attack and angle of sideslip, yaw angle and heading angle are approximately equal. The altitude and heading tracker generate command inputs for the inner loop, while the velocity tracker is independent of it.

C.6.1 Velocity Tracker

The velocity tracker modulates throttle to track the user defined airspeed. It implements a PID control with a discreet low pass filter and integrator anti-wind up. Equation C-2 represents the discreet low pass filter used for both the velocity tracker and altitude tracker. Figure C-10 is a block diagram of the tracker implemented in the simulation. Note that all diagrams flow from right to left. This is to match the pre-existing convention of the Ultrastick simulation. Figure C-11 shows the system response to a velocity increase of 5m/s.

$$\frac{Y(z)}{G(z)} = \frac{0.0392}{z - .9608} \tag{C-2}$$

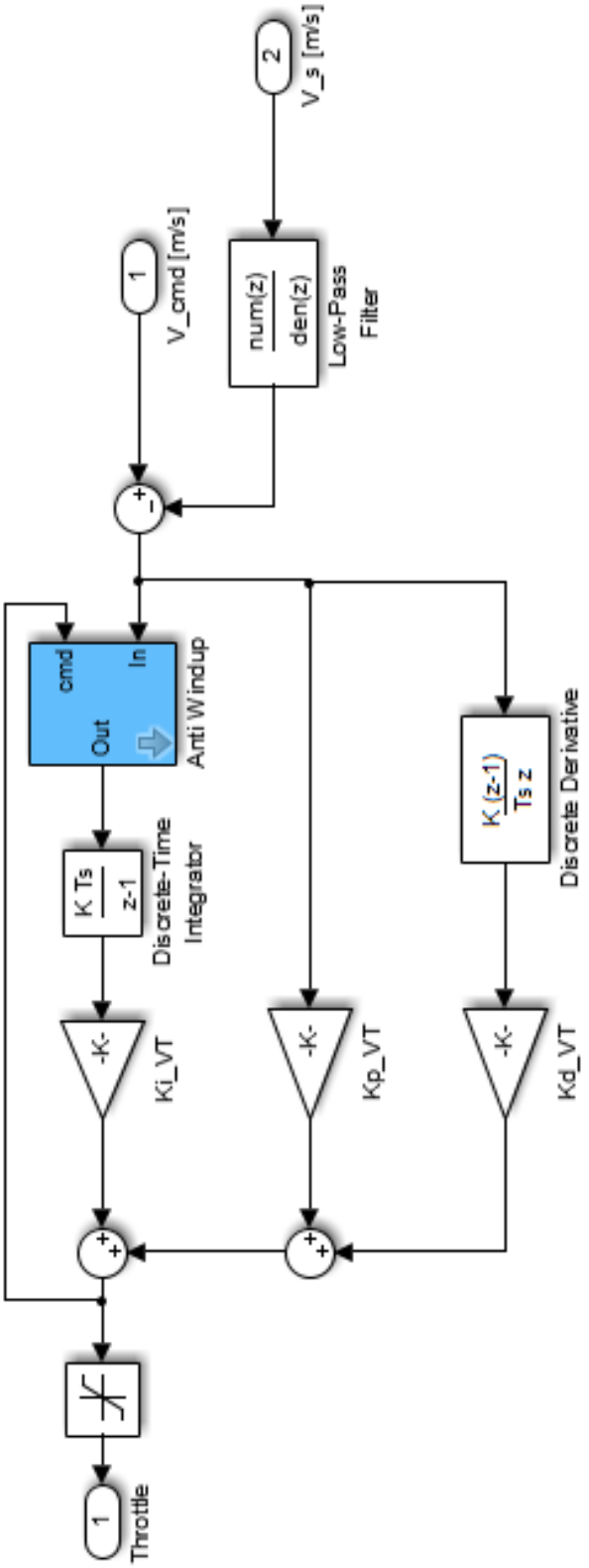


Figure C-10: Velocity Tracker

Table C-9: Inputs to the Velocity Tracker

Inputs	Units	Variable Name	Value Origin
Air Speed	m/s	V_s	Sensor
Commanded Air Speed	m/s	V_cmd	User Defined

Table C-10: Outputs of the Velocity Tracker

Outputs	Units	Variable Name	Destination
Throttle	N/A	throttle	Actuators

Table C-11: Control Parameters for Velocity Tracker

Control Parameter	Variable Name	Value
Proportional Gain	Kp_VT	1.0
Integrator Gain	Ki_VT	0.7
Derivative Gain	Kd_VT	1.0

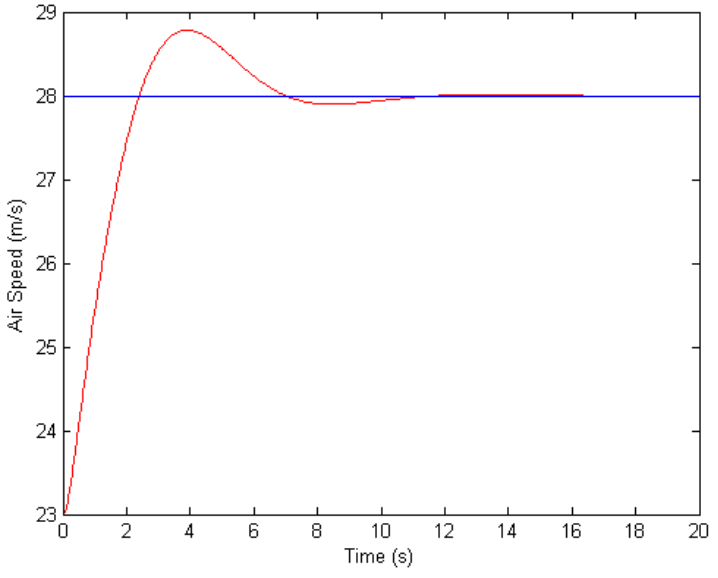


Figure C-11: Velocity Response

C.6.2 Altitude Tracker

The altitude tracker generates a commanded Θ that is fed into the inner loop. It implements a PI control loop with a discreet low pass filter and integrator anti-wind up. The altitude tracker uses the same discreet low pass filter employed by the velocity tracker represented in Equation C-2. Figure C-13 shows the vehicle response when the target altitude is increased by 25m.

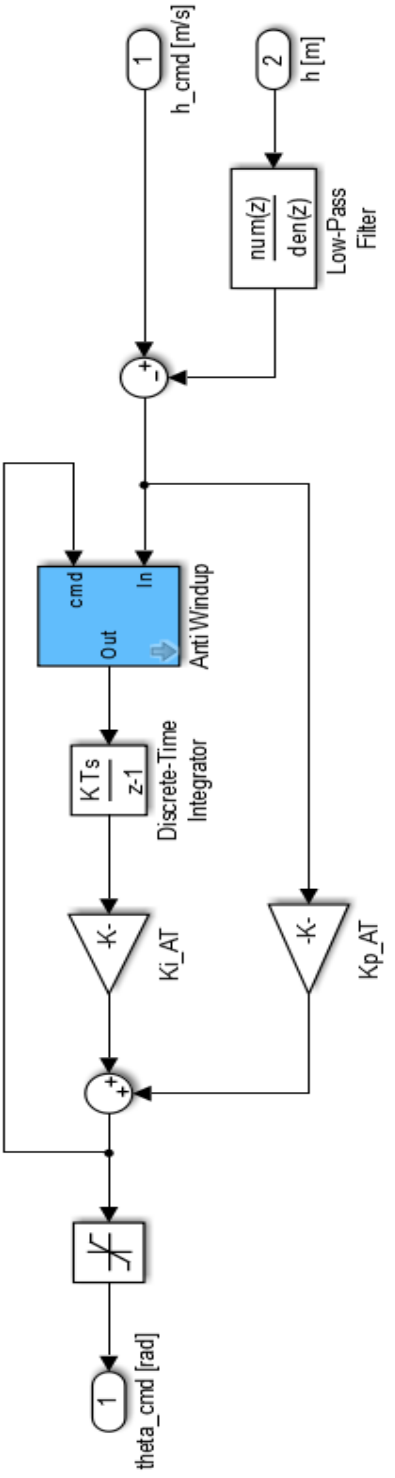


Figure C-12: Altitude Tracker

Table C-12: Inputs of the Altitude Tracker

Inputs	Units	Variable Name	Value Origin
Altitude	m	h	Sensor
Commanded Altitude	m	h_cmd	User Defined

Table C-13: Outputs of the Altitude Tracker

Outputs	Units	Variable Name	Destination
θ_{com}	rad	theta_cmd	Inner Loop Control

Table C-14: Control Parameters for Altitude Tracker

Control Parameter	Variable Name	Value
Proportional Gain	Kp_AT	0.04
Integrator Gain	Ki_AT	0.009

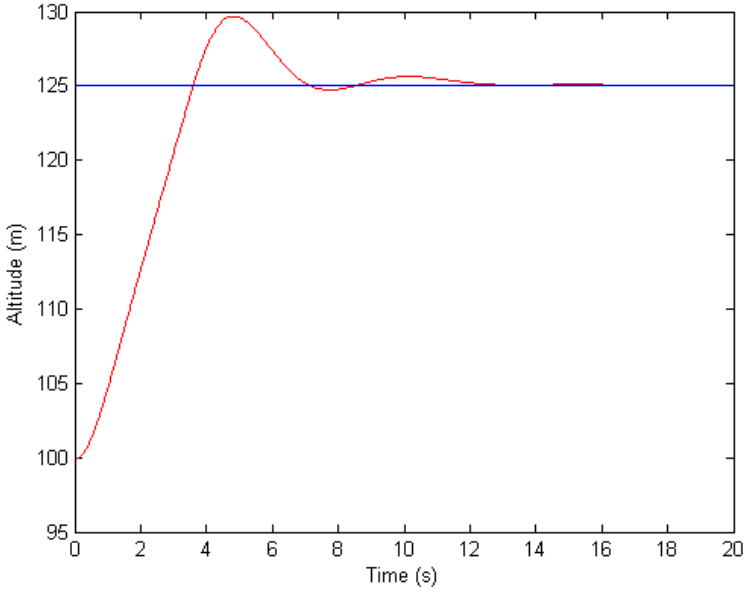


Figure C-13: Altitude Response

C.6.3 Heading Tracker

The heading tracker generates a φ command that is fed into the inner loop. While the heading tracker actually tracks ψ, for small angles of attack and sideslip, ψ is approximately χ. The commanded heading angle and an increment based on cross-track error are added together to calculate the total command. The vehicle's heading is controlled by a simple proportional control loop. The heading tracker also contains logic to resolve angle wrapping issues, which is especially important when the vehicle is headed in a northernly direction.

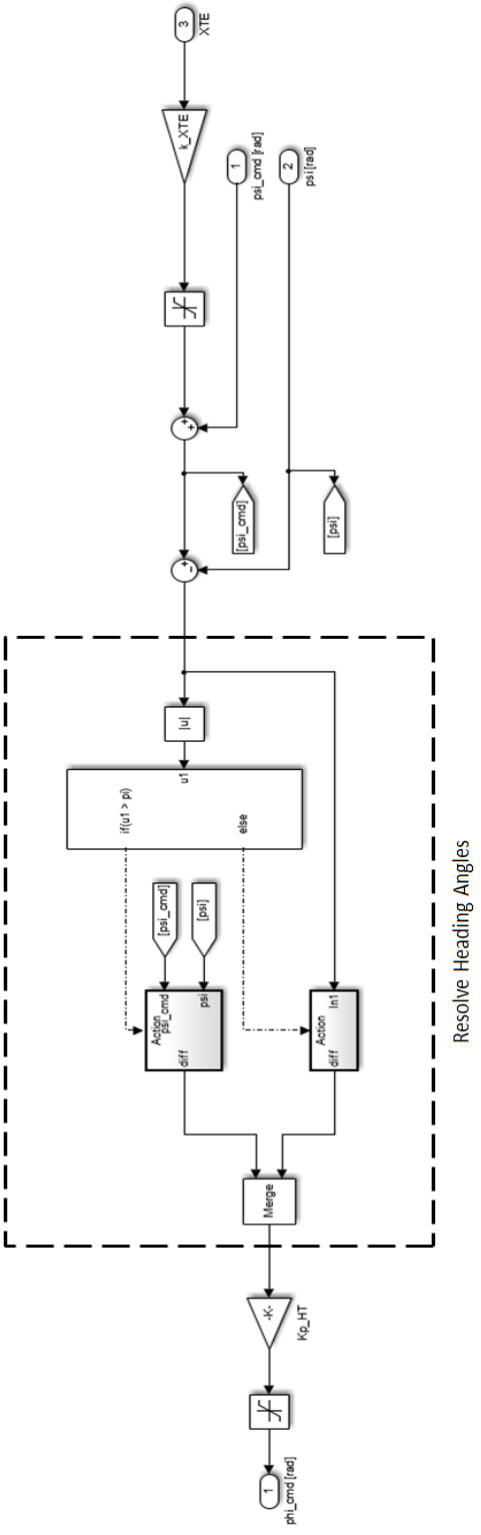


Figure C-14: Yaw Tracker

Table C-15: Inputs of the Heading Tracker

Inputs	Units	Variable Name	Value Origin
ψ	rad	psi	Sensor
ψ_{com}	rad	psi_cmd	Dubin's Car
Cross-Track Error	m	XTE	Dubin's Car

Table C-16: Outputs of the Heading Tracker

Outputs	Units	Variable Name	Destination
φ_{com}	rad	phi_cmd	Inner Loop Control

Table C-17: Control Parameters for Heading Tracker

Control Parameter	Variable Name	Value
Proportional Gain	Kp_HT	1.2
Cross-track Error Gain	Ki_VT	-0.06

C.7 Path Planner/Guidance

The Dubin's Car algorithm is a simple 2D path planning solution that calculates the shortest curve between two points with specified initial and terminal orientations and with constant speed and upper limit on path curvature. The optimal solution is a bang-bang type of solution consisting of at most three path segments and takes either the form CCC or CSC, where C represents circular arcs of maximum curvature, and S represents straight lines. Inputs are defined in Table C-18. Equation C-3 is used to calculate the maximum radius (R) based on the airspeed (V_s), gravity (g), and the user defined turning bank angle (φ). The optimum path is defined by the states in Table C-19. The reference commands fed to guidance are interpolated from the optimized path based on the vehicle's current location. The reference vertical position and velocity are user defined, and controlled independently.

$$R = \frac{V_s^2}{g * \tan(\varphi)} \tag{C-3}$$

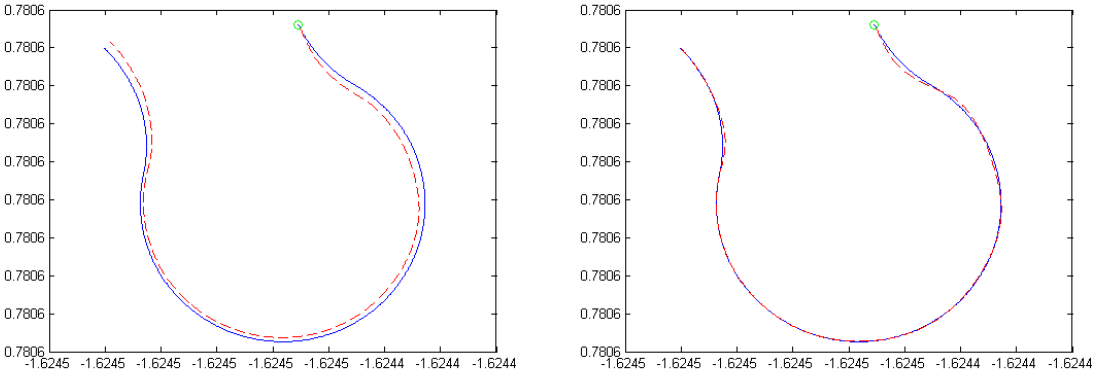
Table C-18: Inputs to Dubin's Car Path Planner

Input Commands	Units	Value Origin
Air Speed	m/s	Sensor
Geodetic Latitude	rad	Sensor
Geodetic Longitude	rad	Sensor
Altitude above sea level	m	Sensor
Yaw angle	rad	Sensor
Target Geodetic Latitude	rad	User Defined
Target Geodetic Longitude	rad	User Defined
Target altitude above sea level	m	User Defined
Target yaw angle at waypoint	rad	User Defined
Turning bank angle	rad	User Defined

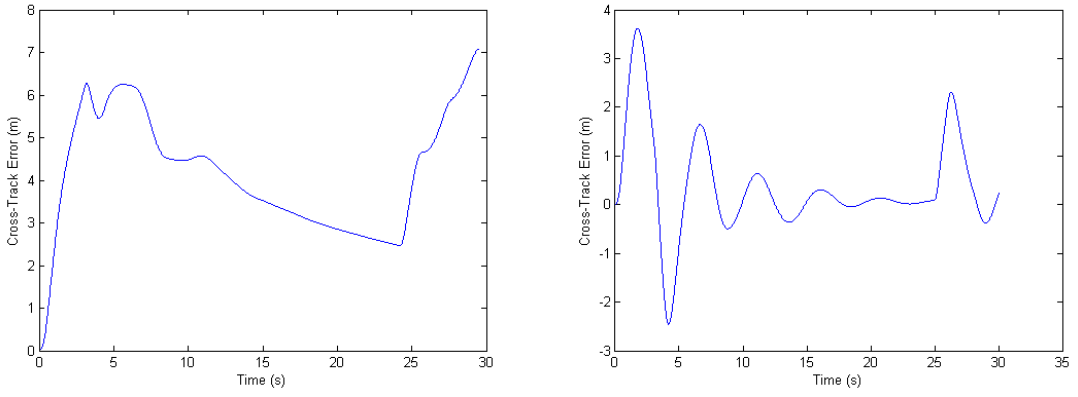
Table C-19: Outputs of Dubin's Car Path Planner

Reference Path Commands	Units
Geodetic Latitude	rad
Geodetic Longitude	rad
Altitude above sea level	m
Yaw	rad
Roll	rad

The vehicle is controlled in the lateral and longitudinal planes by controlling the vehicle's roll and yaw. These reference commands are interpolated based on the vehicles position along the planned path. The Dubin's Car algorithm assumes the maximum turn rate can be achieved instantly. Though the vehicle dynamics are reasonably fast, it of course cannot achieve the turned rate instantly. A cross-track error component in the path tracker corrects for errors due to the assumption, as well as errors arising from other sources including disturbances. An example of how cross-track correction aids the vehicle can be seen in Figure C-15. In Figure C-15(a), the planned path of the vehicle can be seen in blue and the actual path of the vehicle in red. While the vehicle follows a similar path to that planned,it ultimately does not reach the target location. In Figure C-15(b) the vehicle has cross-track correction and can adjust its yaw command to turn the vehicle back onto the planned path. In order to have this capability, there must be enough margin left between the maximum bank angle and the turning bank angle fed to the path planner. A bank angle of 25° was selected for the turning bank angle. The maximum bank angle allowed by the inner-loop of the vehicle is 45°.



(a) Ground track without cross-track correction (b) Ground track with cross-track correction



(c) Cross-track error without correction (d) Cross-track error with correction

Figure C-15: Cross Track correction example

C.7.1 Guidance

The guidance system is designed to track the reference commands from the path planner, as well as the user defined target velocity and altitude. It does this by implementing a cascade control structure. The interfaces of the simulation and the guidance system are shown in in Figure C-2. Each system will be further examined in the following sections. The interface for the guidance system is defined in Tables C-20, C-21, and C-22.

Table C-20: Reference Commands for Waypoint Tracking

Reference commands	Units	Variable Name
Feed Forward Roll angle	rad	phi_cmd
Commanded Yaw angle	rad	psi_cmd
Cross-track error	m	XTE
Commanded Airspeed	m/s	V_cmd
Commanded Altitude above ground level	m	h_cmd

Table C-21: Sensor Outputs Relevant to Waypoint Tracking

Sensor Outputs	Units	Variable Name	Sensor Model
Roll	rad	phi	N/A
Pitch	rad	theta	N/A
Yaw	rad	psi	N/A
Angular Rate about body x-axis	rad/s	p	bias, scale factor, noise
Angular Rate about body y-axis	rad/s	q	bias, scale factor, noise
Angular Rate about body z-axis	rad/s	r	bias, scale factor, noise
Airspeed	m/s	V_s	bias, scale factor, noise
Altitude above ground level	m	h	bias, scale factor, noise

Table C-22: Controller Outputs Relevant to Waypoint Tracking

Control Outputs	Units	Variable Name	Actuator Model
Throttle command	non-dim	Throttle	Input limited between aircraft settings
Elevator Command	rad	elevator	First order model with rate and position limits
Rudder Command	rad	rudder	First order model with rate and position limits
Aileron Command	rad	Aileron	First order model with rate and position limits

C.8 Pixhawk Autopilot Sensor Package

The Pixhawk autopilot was identified as representative of autopilot hardware for SUAS and the sensors on this autopilot motivated the configuration of the state estimator as well as the sensor error characteristics used in testing. The Pixhawk has four sensor chips on board and an attached GPS. The onboard sensor chips consist of an Invensense MPU 6000 3-axis accelerometer/gyro, ST Micro L3GD20H 16 bit gyroscope, ST Micro LSM303D 14 bit accelerometer/magnetometer, and an MEAS MS5611 barometer. The GPS selected for this application is a 3DR u-blox unit with a NEO-7 series GPS module. The available sensor characteristics have been outlined in Table C-23.

Table C-23: Sensor Characteristics

Sensor	Model	Range	Sensitivity	Noise Density	RMS Noise	Sample Rate (Hz)
Invensense MPU 6000 (primary)	3-axis accelerometer	±16 g	2048 LSB/g	400 $\mu g/\sqrt{Hz}$	5012 μg	100
	3-axis gyroscope	±2000 deg/s	16.4 LSB/(deg/s)	0.005 deg/s/$\sqrt{Hz}$	0.0626 deg/s	100
Gyroscope	ST Micro L3GD20H 16 bit gyroscope	±245 deg/s	8.75 (mdeg/s) / digit	0.011 deg/s/$\sqrt{Hz}$	0.1378 deg/s	100
Accelerometer / magnetometer	ST Micro LSM303D 14 bit accelerometer	±16g	0.732 mg/LSB			
	magnetometer	±2 gauss	0.08 mgauss / LSB		5 mgauss/RMS	
Barometer	MEAS MS5611 Barometer Pressure	10-1200 mbar				
	Temperature	-40-85 C				10
GPS Position	ublox NEO-7 series	N/A			3 m (hor), 6 m (vert)	10
GPS Velocity	ublox NEO-7 seris				0.12 m/s	10

The Range and Sample Rate of the MPU 6000 gyroscope and accelerometers are user-selectable. Digital low-pass filters can also be specified by the user which could determine the sample rate. The values indicated in Table C-23 are reasonable values. To calculate the RMS noise, σ, from noise density (ND), the following formula is used:

$$\sigma = ND\sqrt{BW_{eff}} \tag{C-4}$$

$$BW_{eff} = \kappa BW$$

where BW_{eff} is the effective noise bandwidth of the output filter. This is found by scaling the bandwidth of the output filter by a constant, κ . The scaling constant is dependent on the order of the filter and is 1.57 for a first-order filter. For an ideal *brick wall* filter the scaling is 1.0.

C.9 Filtering/State Estimator

The sensors on a small unmanned air system (SUAS) typically exhibit sufficient error that direct use of the raw sensor values in the control system is not desirable, so some filtering is typically

included in the system. Also, the full state vector is typically not measured directly, and state estimation is often employed to provide estimates of the full state vector that can be used in a control system. A Kalman filter was selected for the current implementation because it provides flexibility to interface with a variety of sets of measured quantities. A configuration with full state measurement was first investigated using a basic Kalman filter, which employs a linear model of the system. Section C.9.3 describes experiments that confirm that nonlinearities in the true system lead to significant errors in the state estimates produced by the Kalman filter under certain conditions. Section C.9.4 describes that in this case the Kalman filter served as a smoothing filters could be used in the current configuration that includes full state feedback, this approach would not extend directly to cases in which the desired controller inputs are not a subset of the measured quantities.

C.9.1 Principles of a Kalman Filter

The Kalman filter (cf. [54]) is one of the most well known recursive state estimation algorithms. It computes exact quantities for the conditional mean vector

$$\hat{\mathbf{x}}_{k|k} = E\left[x_k \middle| y_k, y_{k-1}, \ldots, y_1; u_{k-1}, u_{k-2}, \ldots, u_1 \right]$$

for linear systems with additive Gaussian noise governed by the dynamic equations

$$\begin{aligned} \mathbf{x}_{k+1} &= A_k \mathbf{x}_k + B_k \mathbf{u}_k + \mathbf{w}_k \\ \mathbf{y}_k &= C_k \mathbf{x}_k + \mathbf{v}_k \end{aligned} \tag{C-5}$$

The quantities A_k, B_k, and C_k are matrices that can change over time. The process noise $\mathbf{w}_k$ is assumed to have zero mean and covariance Qk, and the measurement noise is assumed to have zero mean and covariance R_k. Because these equations are linear, the conditional mean vector and covariance matrix can be computed recursively in closed form. The Kalman filter computations are typically described as having prediction and update or prediction and correction stages, as described below.

C.9.1.1 Predict

The prediction stage produces the a priori estimate of the state:

$$\hat{\mathbf{x}}_{k|k-1} = A_k \hat{\mathbf{x}}_{k-1|k-1} + B_k \mathbf{u}_{k-1} \tag{C-6}$$

based on the system dynamics in Eq. C-5, and the a priori estimate of the covariance:

$$P_{k|k-1} = A_k P_{k-1|k-1} A_k^T + Q_k \tag{C-7}$$

C.9.1.2 Update

The update phase produces the Kalman gain K_k, which is a function of both the a priori state covariance matrix ($P_{k|k-1}$) and the measurement error covariance matrix (R_k):

$$K_k = P_{k|k-1} C_k^T (C_k P_{k|k-1} C_k^T + R_k)^{-1} \tag{C-8}$$

the a posteriori state estimate $\hat{\mathbf{x}}_{k|k}$, which is updated based on the difference between the predicted and measured state values:

$$\hat{\mathbf{x}}_{k|k} = \hat{\mathbf{x}}_{k|k-1} + K_k(\mathbf{y}_k - C_k\hat{\mathbf{x}}_{k|k-1}) \quad \text{(C-9)}$$

and the a posteriori covariance matrix estimate:

$$P_{k|k} = (I - K_k C_k)P_{k|k-1} \quad \text{(C-10)}$$

The algorithm above is exact only for linear systems. For nonlinear systems, the matrices A_k, B_k, and C_k are often formed by linearizing the system around the mean vector μ_k and the input vector u_k. The resulting algorithm is referred to as an extended Kalman filter. The initial implementation described below assumes linear time-invariant (LTI) system dynamics, so A_k, B_k, and C_k are constant. The air vehicle model used in simulation evaluations is nonlinear, and the simulation results shown below reflect estimation errors that are likely due in large part to the simplifying assumption of an LTI system made within the estimator.

C.9.2 Implementation of Kalman Filter

In order to implement a Kalman Filter with the Ultrastick simulation, a discrete linear model of the plant needed to be generated. The model was linearized about the states listed in Table C-24. The outputs of the linear model are listed in Table C-25. For this test case, it is assumed that measurements of the roll, pitch, and yaw angles are available. These angles are often directly measured with gyroscopic instruments on manned aircraft, but for small unmanned systems it is more common to estimate these states based on other sensors including rate gyroscopes, linear accelerometers, and magnetometers. Measurements of angle of attack and sideslip are often available on research aircraft, including small vehicles such as the Ultrastick, but are less commonly available on production SUAS. All other measurements are commonly available on SUAS. As discussed above, the Kalman filter can readily be used for state estimation, allowing future experiments to be easily conducted with different assumptions about the outputs of the sensor subsystem.

Table C-24: Kalman Filter States

State	Units	Variable Name
Roll angle	rad	ϕ
Pitch angle	rad	θ
Yaw angle	rad	ψ
Angular rate about body x-axis	rad/s	p
Angular rate about body y-axis	rad/s	q
Angular rate about body z-axis	rad/s	r
Velocity in body x-axis	m/s	u
Velocity in body y-axis	m/s	v
Velocity in body z-axis	m/s	w
Flat Earth x-position	m	Xe
Flat Earth y-position	m	Ye
Flat Earth z-position	m	Ze

Table C-25: Sensor Observations

State	Units	Variable Name	Physical Sensor
Roll angle	rad	ϕ	None - Simulation only
Pitch angle	rad	θ	None - Simulation only
Yaw angle	rad	ψ	None - Simulation only
Angular rate about body x-axis	rad/s	p	Gyroscopes
Angular rate about body y-axis	rad/s	q	Gyroscopes
Angular rate about body z-axis	rad/s	r	Gyroscopes
Accelerations in body x-axis less gravity	m/s^2	Ax	Accelerometers
Accelerations in body y-axis less gravity	m/s^2	Ay	Accelerometers
Accelerations in body z-axis less gravity	m/s^2	Az	Accelerometers
Air Speed	m/s	V_s	Pitot Tube
Angle of Side Slip	rad	beta	Wind Vane
Angle of Attack	rad	alpha	Wind Vane
Above ground altitude	m	h	Barometer
Flight path angle	rad	gamma	GPS
Flat Earth x-position	m	Xe	GPS
Flat Earth y-position	m	Ye	GPS
Flat Earth z-position	m	Ze	GPS

The linear model used in the Kalman filter is generated at a straight and level trim condition[13], and the Kalman filter estimates states that represent perturbations from this trim condition. Trim values of the states, outputs, and inputs are denoted $\mathbf{x}_{t0}$, $\mathbf{y}_{t0}$, and $\mathbf{u}_{t0}$, respectively, and the corresponding perturbation quantities are defined as

$$\Delta\mathbf{x}=\mathbf{x}-\mathbf{x}_{t0} \tag{C-11}$$

$$\Delta\mathbf{y}=\mathbf{y}-\mathbf{y}_{t0}$$

$$\Delta\mathbf{u}=\mathbf{u}-\mathbf{u}_{t0}$$

The Kalman filter thus operates on the linear model

$$\Delta\mathbf{x}_{k+1} = A_k\Delta\mathbf{x}_k+B_k\Delta\mathbf{u}_k$$

$$\Delta\mathbf{y}_k = C_k\Delta\mathbf{x}_k$$

with appropriate trim increments added and subtracted at the Kalman filter interface in the simulation.

C.9.2.1 Simulink Model

The Kalman Filter Simulink Model is shown in Figure C-16. The *a priori* state and error covariance estimates are initialized in the memory blocks xHat and P-, respectively. Since the

[13] The linearization routine in theUltrastick simulation environment produces a continuous time model, so a continuous to discrete conversion step is needed to generate the discrete model used in the Kalman filter. The is done using the Matlab default zero order hold method.

linear model operates on the perturbation quantities ($\Delta\mathbf{x}$, $\Delta\mathbf{y}$, $\Delta\mathbf{u}$), the trim measurements must be subtracted from the real time measurements. The control inputs are commanded deflections from the trim settings, so no modification is needed. Because the current control system is designed to accept the measured quantities as inputs, the state estimates are multiplied by the output matrix C_k to compute the estimated outputs of the system. Given the current control system, the Kalman filter could be replaced with simple filtering of the measured quantities, but using the Kalman filter provides greater flexibility to replace both the sensor and control subsystems without requiring architectural changes in the filtering/state estimation subsystem.

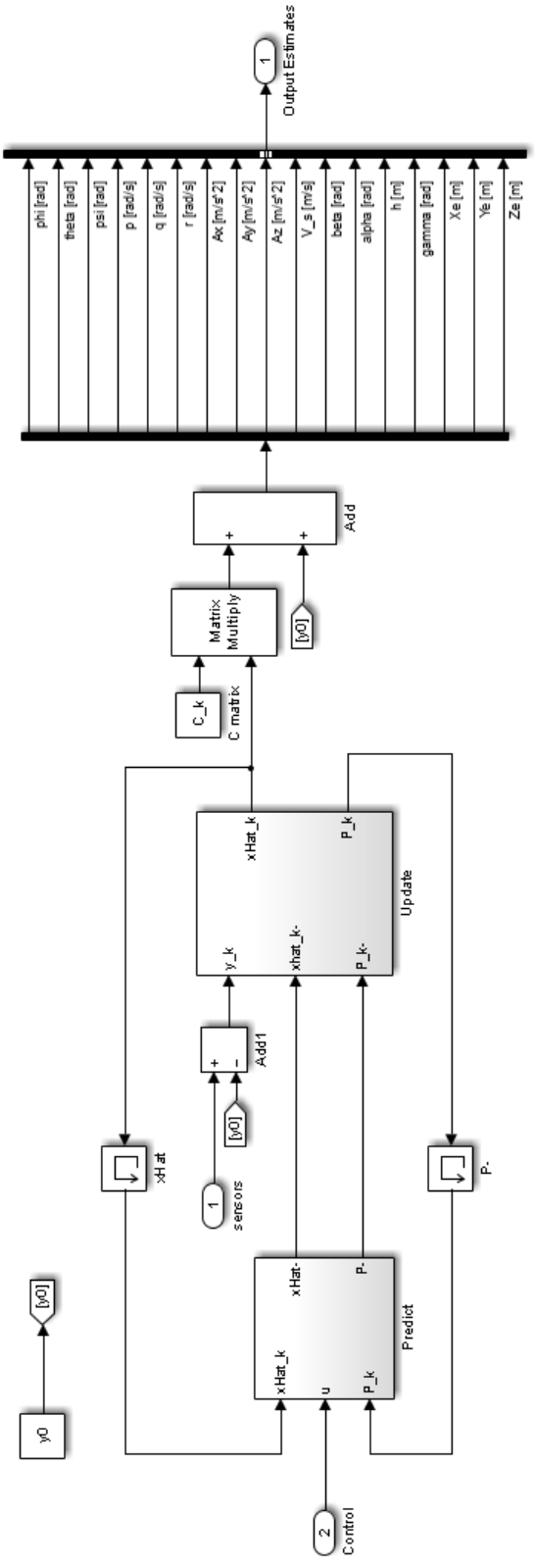


Figure C-16: Kalman Filter

The prediction step is captured in Figure C-17. This model corresponds to Eqs. C-6 and C-7.

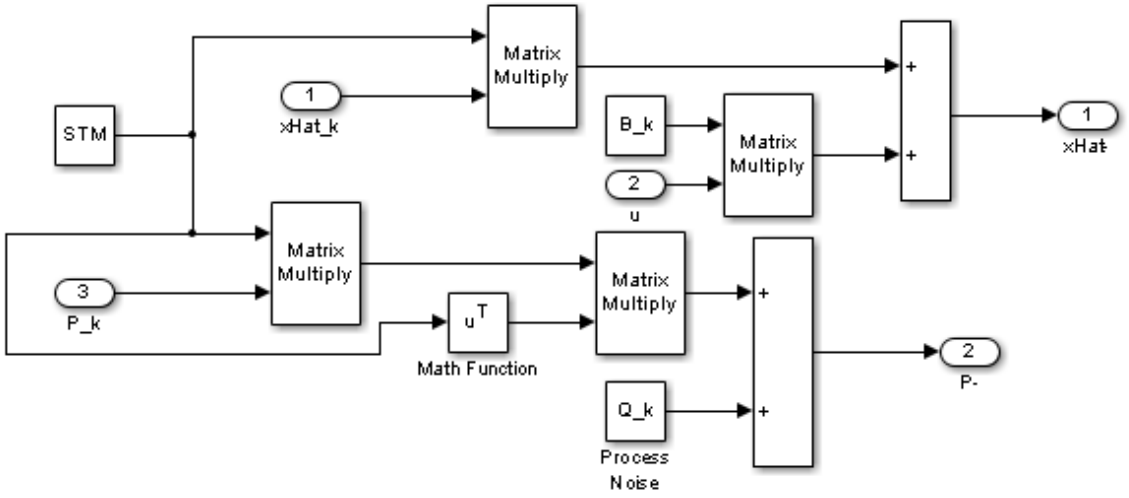


Figure C-17: Kalman Filter: Predict

The update step is broken into multiple subsystems: compute Kalman gain, update states, and update error covariance, as shown in Figure C-18.

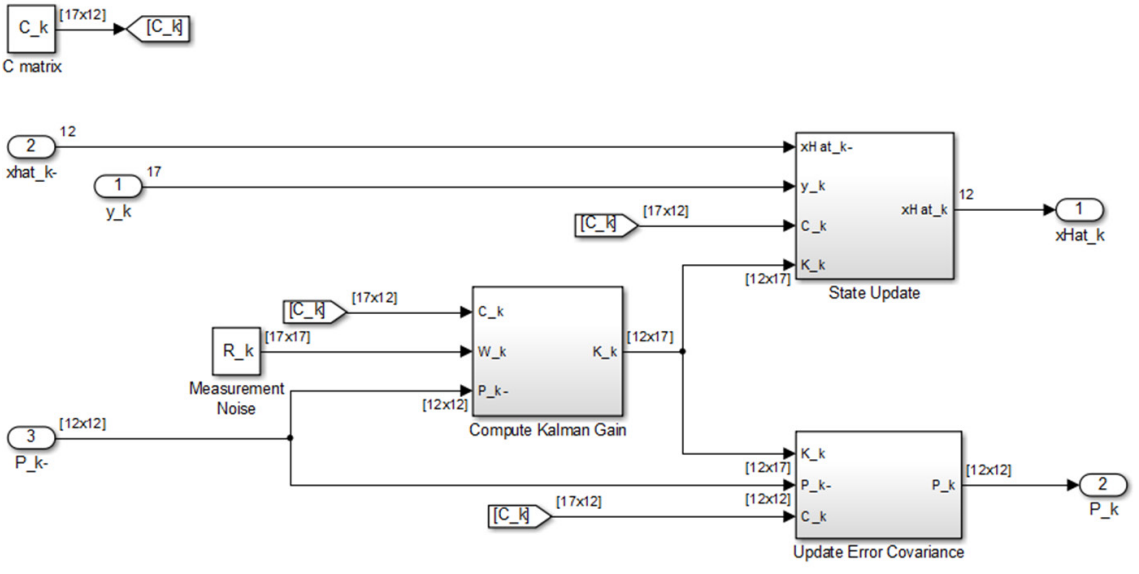


Figure C-18: Kalman Filter: Update

The *Compute Kalman Gain* subsystem implements Eq. C-8 to find the Kalman Gain, as shown in Figure C-19.

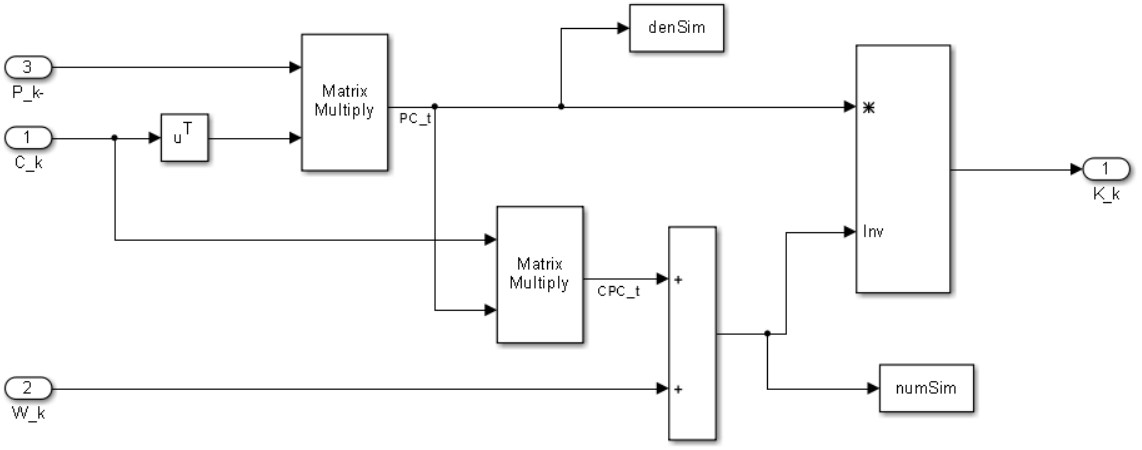


Figure C-19: Kalman Filter: Compute Kalman Gain

The *State Update* subsystem shown in Figure C-20 uses the measurements from the sensors to compute the a posteriori state estimate, implementing Eq. C-9.

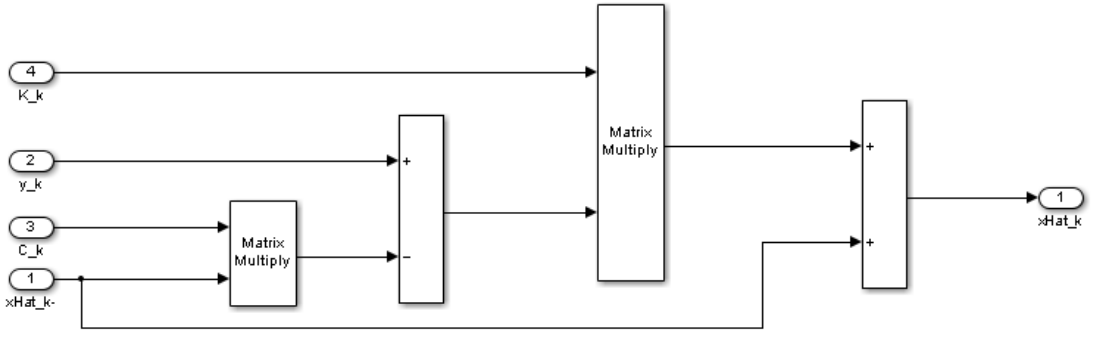


Figure C-20: Kalman Filter: State Update

In the *Update Error Covariance* subsystem, Eq. C-10 is used to update the estimate of the error covariance

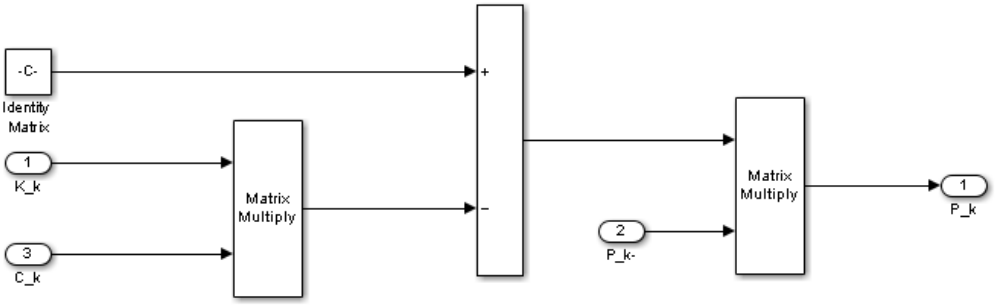
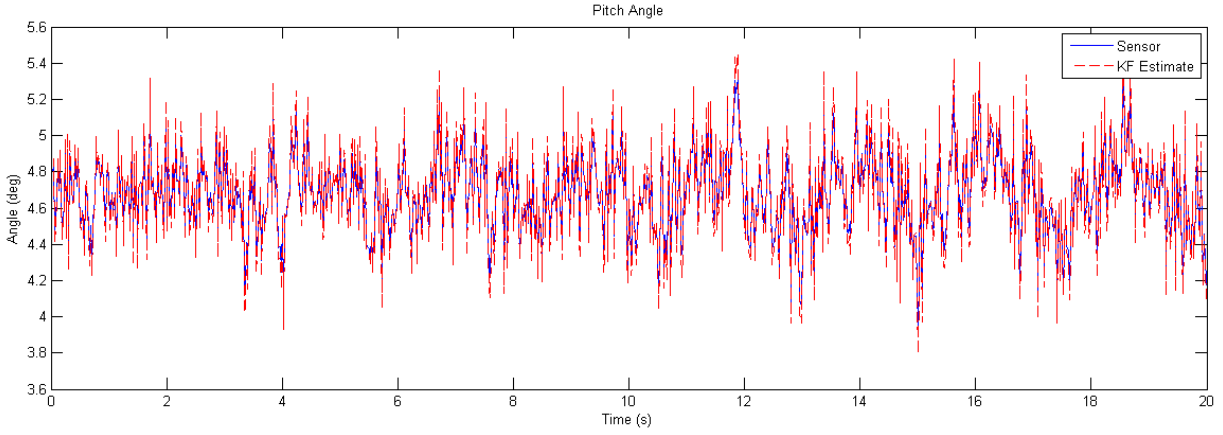


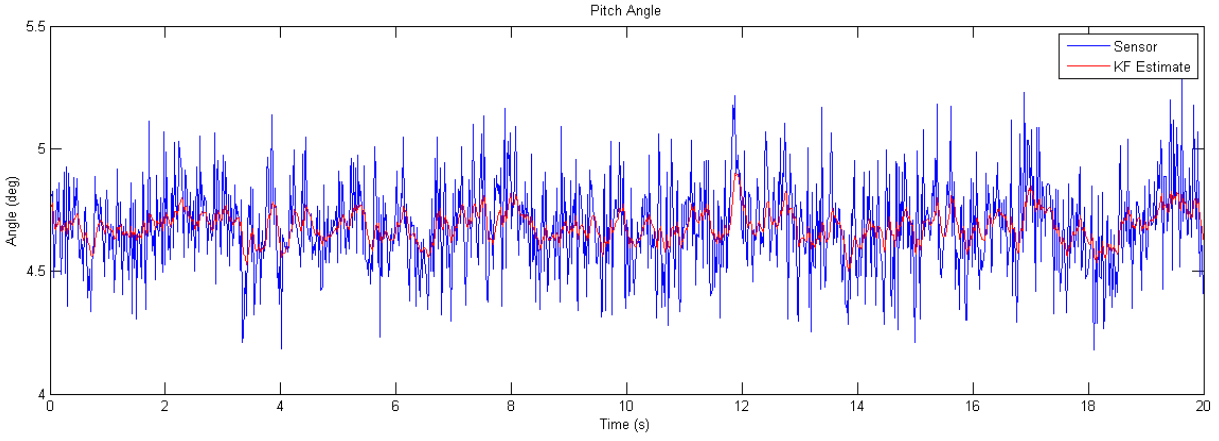
Figure C-21: Kalman Filter: Error Covariance Update

C.9.2.2 Tuning the Kalman Filter

The process noise covariance matrix (Q_k) and the measurement noise covariance matrix (R_k) are Kalman filter parameters that must be chosen carefully to achieve the desired performance of a Kalman Filter. The process noise should capture disturbances and errors introduced by the linear model assumption. An initial estimate of the process noise focused on modeling error was generated by executing the full nonlinear simulation model and the linear system model (Eq. C-5 with no noise) and computing the difference in the state increments at each step. The measurement covariance matrix (R_k) was initially populated with the variance of the sensor noise along the diagonals. From this initial point, the parameters can be tuned to obtain the desired performance from the filter. An example of tuning impact is shown in Figure C-22. In Figure C-22(a), the measurement covariance matrix is defined by the sensor noise. As can be seen, it retains much of the noise from the sensor. By scaling R_k, we are able to effectively reduce the weighting on the measurements and obtain a much smoother estimate of the pitch angle that removes the high frequency noise of the sensor.



(a) Pitch Angle without tuned measurement matrix



(b) Pitch Angle with tuned measurement matrix

Figure C-22: Cross Track Correction Example

C.9.3 Multi-Model Kalman Filter

Testing with the basic Kalman filter with a single linear model of the system showed that a bias existed between the Kalman filter estimated states and the true states of the system in some cases when system states deviated significantly from the trim point at which the linear model used by the Kalman filter was generated. Figure C-23 shows an example in which pitch angle deviates significantly from the trim point and biases in state estimates are present. To test the hypothesis that the bias is due to nonlinear system dynamics, a Kalman Filter was created that employed two separate linear models. The first was linearized about the initial states, and the second was linearized with a pitch angle of 15°, the commanded step input for pitch in the example in Figure C-23. The filter then switches between the two models based on which pitch angle it is closest to. As can be seen in Figure C-24, introduction of the second model nearly eliminates the steady state biases in the Kalman filter estimates, supporting the hypothesis that the biases are due to nonlinearities in the system dynamics. Because the focus of the current effort is on requirements and not on development of a high-performance system, the use of a single linear model in the Kalman filter has been deemed adequate. The results shown here are intended primarily to

demonstrate that the observed errors are due to system nonlinearities and not an error in the Kalman filter implementation.

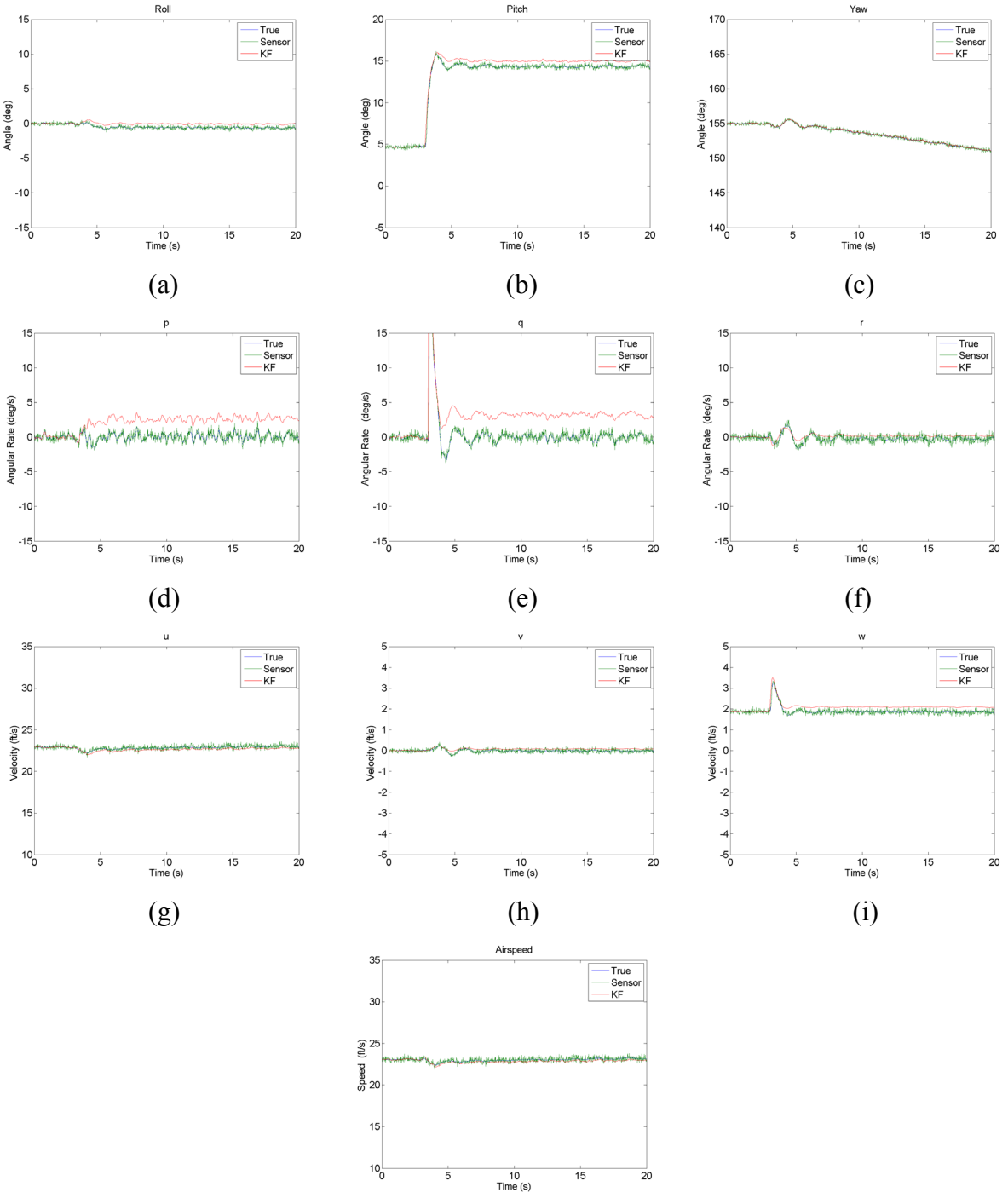


Figure C-23: Kalman Filter Pitch Angle Response — One Linear Model

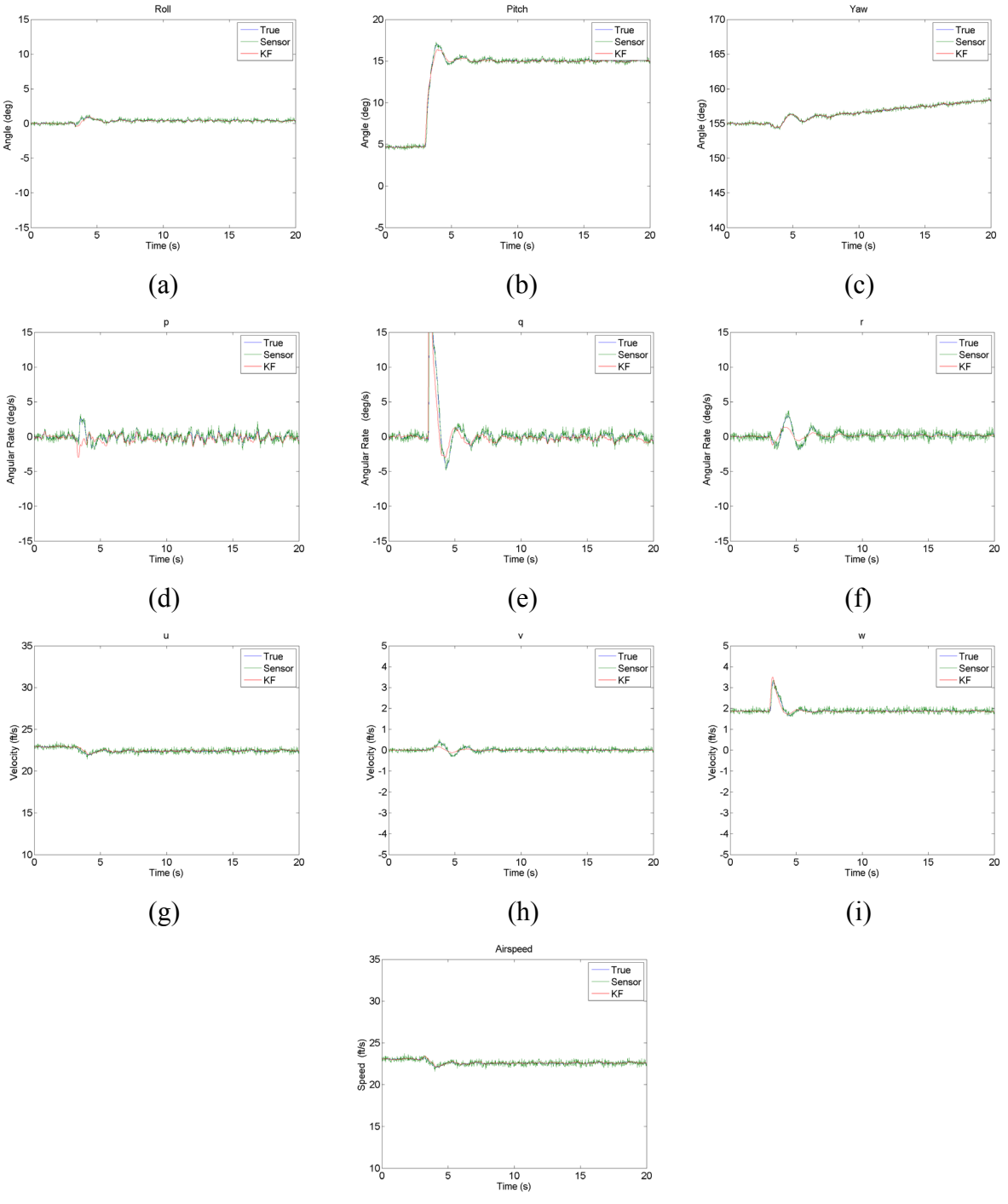


Figure C-24: Kalman Filter Pitch Angle Response — Two Linear Models

C.9.4 Unscented Kalman Filter

The estimator previously developed assumed full state feedback. This is not typical of sensor packages used onboard small unmanned aerial systems such as the Ultrastick. As discussed in

Section C.8 , the Pixhawk autopilot was identified as representative of SUAS autopilots and an state estimator was developed using inputs corresponding to the Pixhawk sensor package. This section provides an overview of the structure of the Unscented Kalman Filter (UKF) [55] that was developed to provide a state estimator architecture that is consistent with the Pixhawk sensor package. Figure C-25 provides a high-level overview of the measurements and desired state estimates (outputs) of a typical SUAS estimator.

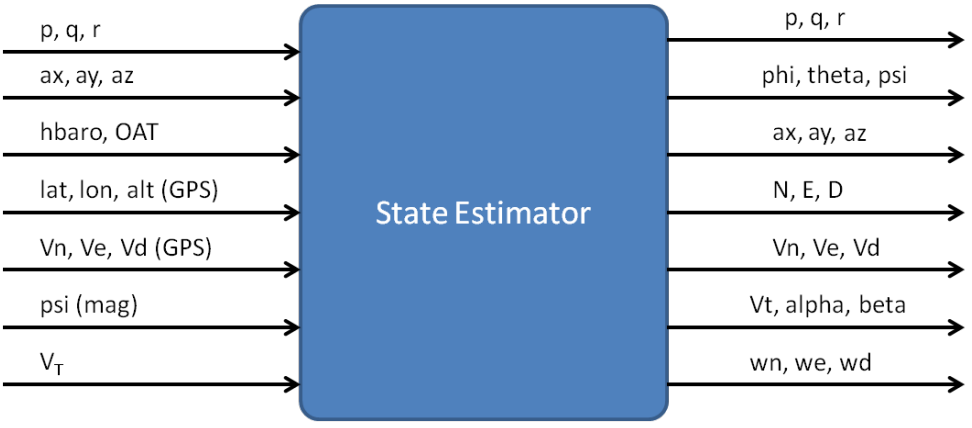


Figure C-25: SUAS State Estimation Overview

In the implementation developed for the present work, winds and aerodynamic angles were not included among the estimator outputs. The measurements and estimator outputs in the present implementation are thus:

Measurements

- (p, q, r): Body-axis angular rates with respect to an inertial frame provided by angular rate gyroscopes from an Inertial Measurement Unit (IMU)
- (ax, ay, az): Body-axis acceleration measurements with respect to an inertial frame provided by the IMU
- (hbaro, OAT): barometric height measurement and Outside Air Temperature provided by a barometer pressure sensor
- (lat, lon, alt): Geographic position provided by a GPS unit. Alternatively, local geodetic data can be provided in terms of North, East, and Down (NED) relative to a flat-earth frame.
- (Vn, Ve, Vd): Inertial velocity provided by a GPS unit
- psi: heading with respect to true north, provided by a magnetometer
- V_T: airspeed measurement provided by a pitot probe

Estimator Outputs

- (p, q, r): Body-axis angular rates with respect to an inertial frame compensated for modeled sensor errors
- (phi, theta, psi): Euler attitude angles. An Euler angle formulation was used for compatibility with the UltraStick simulation.
- (ax, ay, az): Body-axis acceleration measurements with respect to an inertial frame compensated for modeled sensor errors
- (N, E, D): local geodetic position estimates

- (Vn, Ve, Vd): inertial speed estimates

The process model used in the estimator contains 15 states, consisting of Euler angles, inertial position, inertial velocity, gyro bias estimates, and accelerometer bias estimates. The state vector is given by:

$$\mathbf{x} = \left[\mathbf{q}^T, \mathbf{p}^T, \mathbf{v}^T, \mathbf{b}_\omega^T, \mathbf{b}_a^T\right]^T$$

$$\mathbf{q} = \left[\varphi, \Theta, \psi\right]^T \tag{C-12}$$

$$\mathbf{p} = \left[N, E, H_{bar}\right]^T \tag{C-13}$$

$$\mathbf{v} = \left[V_N, V_E, V_D\right]^T \tag{C-14}$$

$$\mathbf{b}_\omega = \left[b_{\omega_x}, b_{\omega_y}, b_{\omega_z}\right]^T \tag{C-15}$$

$$\mathbf{b}_a = \left[b_{a_x}, b_{a_y}, b_{a_z}\right]^T \tag{C-16}$$

Here **v** is velocity in the inertial frame and **q** is the attitude representation using Euler angles.

The vectors $\mathbf{b}_a$ and $\mathbf{b}_\omega$ represent biases of the accelerometer and gyro measurements, respectively. Two approaches can be taken to model this error. The first is a random bias, where the dynamics for a random element, x, are: $\dot{x}=0, x_o \sim (x_o, P_o)$. Here x_o is the initial state estimate and P_o the initial state covariance. A random bias model can be utilized in systems where there is a constant, unknown error source. Alternatively, the error source can be modeled with a time-varying biases, or random walk model. A random walk model can be used to model a constant bias combined with measurement noise. This formulation has the added benefit of estimating error sources that vary with time. The random walk model is: $\dot{x}=w(t), x_o=0, w \sim (0, Q)$, where w is a Gaussian variable. A random walk model is employed here to formulate the error model, accounting for accelerometer and gyro errors.

Note, that while the angular rates and accelerations are not estimated states, the *corrected* angular rates and accelerations are output from the estimator using the estimated bias states.

C.10 Requirements/System Performance Analysis

C.10.1 Control System Input Requirements

The requirements for input signals to the control system were investigated by assessing the capability of the complete system to successfully complete the mission described in Section C.2 with various bias and noise errors on the input signals. Clearly, a complete requirements analysis process would include goals in addition to mission completion. For example, high-frequency content in the actuator command signals is generally considered undesirable and would typically be limited even if it did not directly impact mission success. Such additional considerations can be readily incorporated in future phases of the work, but were not considered critical to the goals of the initial demonstration environment.

The first analysis involved adding either noise or bias errors to a single input signal and assessing the amount of bias or noise that could be tolerated. The goal of this initial analysis was not to precisely determine the allowable level of error on each measurement, but rather to establish rough estimates of error levels on each measurement that have a comparable impact on overall mission performance. In a number of cases, the level of error that could be tolerated from a mission performance perspective was very large. This is particularly true in the case of biases, which are in some cases very effectively compensated for by integral compensation in the control system. Maximum acceptable levels of bias and noise on input signals were thus established through a combination of engineering judgment and observed limits based on mission success. Table C-26 shows the established limits for bias and noise on each input signal.

Table C-26: Error Limits for Individual Controller Inputs

Sensor	Error Standard Deviation Limit	Error Bias Limit
Roll (deg)	12.8	±2.3
Pitch (deg)	3.6	±10.0
Yaw (deg)	3.6	±4.6
p (deg/sec)	2	±15.0
q (deg/sec)	2	±15.0
r (deg/sec)	2	±15.0
Air Speed (m/s)	5	±5.0
Altitude (m)	3.5	±5.0
GPS Position (m)	8	±10.0

The next experiment addressed the amount of bias and noise errors that could be tolerated on input signals when bias and noise existed on all input signals simultaneously. The values in Table C-26 establish a vector of noise standard deviation levels for each input and a vector of bias levels for each input. Experiments were conducted in which the magnitudes of these bias and noise vectors were scaled, but the relative noise levels on the various input remained unchanged as did the relative bias levels on the various inputs. As expected, when bias and noise were applied to all inputs simultaneously, the level of bias or noise that could be tolerated on any given input was lower than it was in the case that other inputs had zero bias and noise. Initial experiments also showed positive and negative altitude biases to have significantly different performance impacts due to the change in the sensor field of view. In the results shown, altitude bias was always set to zero to remove this asymmetric effects. Table C-27 shows the likelihood of mission success for various scalings of the bias and noise vectors in Table C-26. These results reflect simultaneous injection of bias and noise on all inputs except altitude[14]. For each test point shown in the table, 100 Monte Carlo simulation runs were conducted. The results suggest that the allowable noise levels are roughly an order of magnitude smaller when noise is simultaneously

[14] The exceptions are the leftmost column and top row, in which results reflect zero bias and zero noise, respectively, as indicated by the zero scaling

applied all sensors than they are when there is a single noisy sensor. Allowable bias levels are similarly reduced.

Table C-27: Likelihood of Mission Success with Varying Bias and Noise on All Controller Inputs

	Bias Scale Factor								
Noise Scale Factor	-0.18	-0.16	-0.12	-0.1	0	0.1	0.12	0.16	0.18
0	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**
0.08	**92**	**94**	**97**	**100**	**97**	**98**	**90**	81	**95**
0.1	88	**91**	**96**	**97**	**96**	**95**	**91**	81	85
0.12	84	77	**93**	**91**	**90**	**91**	**90**	76	81
0.14	79	83	86	**90**	84	86	89	67	72
0.16	78	76	82	**92**	84	81	85	70	65
0.18	72	76	79	86	80	79	76	65	60

Based on the results in Table C-27 and selecting an acceptable performance threshold of 90% mission success, the noise scale factor is limited to 0.12 and the bias scale factor to ±0.12.

C.10.2 UKF Evaluation with Nominal Sensor Characteristics

Experiments were conducted to determine whether the combination of the Pixhawk sensors and the estimator described in Section C.9.4 could meet the requirements for errors in the control system input signals established in Section C.10.1. Table C-28 shows the nominal bias and noise values used for the sensors. Table C-29 compares the errors at the output of the UKF to the requirements for errors in the control system inputs. In Table C-29, the estimation errors at the output of the state estimator are shown in the fourth and seventh columns. The third and sixth columns show the level of bias and noise that the system was predicted to be able to tolerate based on the analysis in Section C.10.1. The second and fifth columns show the vectors of bias and noise levels referenced in Section C.10.1 that were scaled to determine the maximum bias and noise that the system is expected to tolerate at the input to the feedback control system. The results indicate that the errors in pitch and yaw angle estimates and horizontal position estimates produced by the state estimator exceed the error levels to which the system is expected to be robust.

Table C-28: Nominal Sensor Error Characteristics for Simulation

Estimate	Noise Standard Deviation	Bias
Heading (deg)	1.0	±1.0
Gyros [p,q,r] (deg/sec)	0.0626	±0.5
Accelerometers [x,y,z] (μg)	5012	±501.2
Altitude (m)	6	±0.6
Air Speed (m/s)	0.12	±0.012
GPS Position (m)	3	±0.3

Table C-29: Estimator Results for Nominal Sensors

Control Input	Individual STD Limit	Scaled STD Limit (0.12)	STD of Estimate	Individual Bias Limit	Scaled Bias Limit (0.12)	Mean Bias of Estimate
Roll (deg)	12.8	1.54	0.56	±2.3	±0.28	**0.73**
Pitch (deg)	3.6	0.43	**1.78**	±10.0	±1.2	0.54
Yaw (deg)	3.6	0.43	**0.53**	±4.6	±0.55	**1.02**
p (deg/sec)	2	.24	.11	±15.0	±1.8	0.08
q (deg/sec)	2	.24	.11	±15.0	±1.8	0.06
r (deg/sec)	2	.24	.08	±15.0	±1.8	0.02
Air Speed (m/s)	5	.6	.15	±5.0	±.6	-.02
Altitude (m)	3.5	.42	.04	±5.0	±.6	-.01
GPS Position: East-West (m)	8	.96	**2.88**	±10.0	±1.2	.31

C.10.3 Closed-loop System Performance

Initial experiments indicated that the mission performance of the full system comprising the Ultrastick vehicle model, sensors error characteristics based on the Pixhawk sensor package, the UKF state estimator, and the feedback control system was below the targeted 90% threshold. Preliminary experiments also suggested that the mission performance was particularly sensitive to the amount of error in the horizontal position estimate at the controller input. The following section present results that demonstrate the significant impact on mission performance of changes in position measurement errors, and show that improvements in the estimates of other states have less impact on mission performance.

C.10.3.1 Assessment of Impact of Position Measurement Errors

An analysis was thus conducted to assess the impact of reducing the error in horizontal position measurement at the input to the state estimator. Tables C-30 and C-31 show that with an improved GPS sensor, the pitch and yaw estimates do not need to be improved to achieve the desired mission success rate. These tables show the noise levels and bias levels, respectively, in the state estimate errors at the input to the control system as the level of noise in the position error at the input to the state estimator is varied. The last column in the tables also shows mission success as the noise is varied. The results confirm that mission success is sensitive to noise in the position measurement. In these results and subsequent results in this section, all sensor biases were reduced by 50% with a goal of bringing the worst case bias at the output of the estimator to approximately the level of the predicted limit shown in Table C-29.

Table C-30: Nominal Sensor Noise, 50% Sensor Bias

		Control Signal Noise									
		Roll	Pitch	Yaw	p	q	r	Velocity	Altitude	GPS	Success
GPS Noise SF	Limits	1.54	0.43	0.43	0.24	0.24	0.24	0.60	0.42	0.96	-
	0	1.32	1.13	0.52	0.08	0.05	0.07	0.13	0.04	0.04	96.00
	0.4	1.35	1.14	0.52	0.08	0.05	0.07	0.13	0.04	1.19	94.00
	0.8	1.38	1.16	0.53	0.08	0.05	0.07	0.13	0.04	2.34	76.00
	1.0	1.33	1.13	0.53	0.08	0.05	0.07	0.13	0.04	2.89	62.00

Table C-31: Nominal Sensors, 50% Sensor Bias

		Control Signal Bias									
		Roll	Pitch	Yaw	p	q	r	Velocity	Altitude	GPS	Success
GPS Noise SF	Limits	0.28	1.20	0.55	1.80	1.80	1.80	0.60	0.60	1.20	-
	0	0.31	0.25	0.51	0.04	0.03	0.01	-0.01	-0.01	0.15	96.00
	0.4	0.36	0.23	0.51	0.04	0.03	0.01	-0.01	-0.01	0.16	94.00
	0.8	0.34	0.29	0.51	0.04	0.04	0.01	-0.01	-0.01	0.15	76.00
	1.0	0.33	0.28	0.52	0.04	0.04	0.01	-0.01	-0.01	0.16	62.00

C.10.3.2 Assessment of Impact of Improved Pitch and Yaw Estimates

Table C-29 shows that as for the GPS estimates of position, noise in the pitch and yaw estimates exceeds the predicted limit that the closed-loop system can tolerate while maintaining acceptable performance. Tests were conducted in which the noise and bias of both the pitch and yaw estimate were constrained to the maximum amount that Table C-29 predicts the controller could tolerate. Tables C-32 and C-33 show the success rate along with the noise and bias of the control inputs in this configuration with varying levels of GPS sensor noise. These results confirm the preliminary finding that GPS errors have a far greater effect on the success of the mission than errors in the pitch and yaw estimates.

Table C-32: Improved Pitch and Yaw Estimates with 50% Sensor Bias

		Control Signal Noise									
		Roll	Pitch	Yaw	p	q	r	Velocity	Altitude	GPS	Success
GPS Noise SF	Limits	1.54	0.43	0.43	0.24	0.24	0.24	0.60	0.42	0.96	-
	0	1.30	0.43	0.43	0.08	0.05	0.07	0.13	0.04	0.04	98.00
	0.4	1.32	0.43	0.43	0.08	0.05	0.07	0.13	0.04	1.19	98.00
	0.8	1.29	0.43	0.43	0.08	0.05	0.07	0.13	0.04	2.34	76.00
	1.0	1.30	0.43	0.43	0.08	0.05	0.07	0.13	0.04	2.87	61.00

Table C-33: Improved Pitch and Yaw Estimates with 50% Sensor Bias

		Control Signal Bias									
		Roll	Pitch	Yaw	p	q	r	Velocity	Altitude	GPS	Success
GPS Noise SF	Limits	0.28	1.20	0.55	1.80	1.80	1.80	0.60	0.60	1.20	-
	0	0.34	1.20	0.55	0.04	0.03	0.01	-0.01	-0.01	0.15	98.00
	0.4	0.31	1.20	0.55	0.04	0.02	0.01	-0.00	-0.01	0.15	98.00
	0.8	0.33	1.20	0.55	0.04	0.03	0.01	-0.00	-0.01	0.16	76.00
	1.0	0.37	1.20	0.55	0.04	0.03	0.01	-0.00	-0.01	0.14	61.00

C.10.3.3 Assessment of Impact of Improvement in All Sensors

Further tests were run to simulate an overall improved sensor package using 75% of the nominal noise and 50% of the nominal bias levels shown in Tables C-28. The same set of GPS error levels evaluated in the previous section were again used. As shown in Tables C-34 and C-35, the GPS

sensor still has the largest impact on the success of the mission, and the improvement in mission performance produced by reducing noise in all other sensors is small.

Table C-34: 75% of Nominal Sensor Noise, 50% of Nominal Sensor Bias

		Control Signal Noise									
		Roll	Pitch	Yaw	p	q	r	Velocity	Altitude	GPS	Success
GPS Noise SF	Limits	1.54	0.43	0.43	0.24	0.24	0.24	0.60	0.42	0.96	-
	0	1.27	1.11	0.44	0.07	0.05	0.06	0.10	0.03	0.03	99.00
	0.4	1.34	1.13	0.44	0.07	0.05	0.06	0.10	0.03	1.19	92.00
	0.8	1.32	1.08	0.44	0.07	0.05	0.06	0.10	0.03	2.33	75.00
	1.0	1.31	1.14	0.44	0.07	0.05	0.06	0.10	0.03	2.87	69.00

Table C-35: 75% of Nominal Sensor Noise, 50% of Nominal Sensor Bias

		Control Signal Bias									
		Roll	Pitch	Yaw	p	q	r	Velocity	Altitude	GPS	Success
GPS Noise SF	Limits	0.28	1.20	0.55	1.80	1.80	1.80	0.60	0.60	1.20	-
	0	0.37	0.24	0.51	0.04	0.03	0.01	-0.01	-0.01	0.15	99.00
	0.4	0.38	0.23	0.51	0.04	0.03	0.01	-0.01	-0.01	0.16	92.00
	0.8	0.35	0.22	0.51	0.04	0.03	0.01	-0.01	-0.01	0.14	75.00
	1.0	0.38	0.26	0.51	0.04	0.03	0.01	-0.01	-0.01	0.15	69.00

C.11 Conclusions

A test environment based on a simulation of a small unmanned air vehicle was created to support research on systems of systems interactions. The test environment included a simulation of the air vehicle dynamics, sensor models, a state estimator, and a control system. A mission profile representative of common SUAS missions such as crop monitoring was developed, and inner- and outer-loop control system capabilities were implemented to execute a specific mission. Acceptable performance of the overall system was defined in terms of sensor coverage during this mission. The performance of the control system as a function of errors at the control system inputs, the performance of the state estimator given sensor models representative of SUAS sensing hardware, and the impact of changing sensor performance characteristics on the mission success rate were all investigated.