An Improved Design Process for Complex, Control-Based Systems Using STPA and a Conceptual Architecture[1]

Nancy Leveson
Engineering Systems Lab
Aeronautics and Astronautics Dept.
MIT

January 11, 2020

Table of Contents

Introduction

The Problem with the Conventional Approach and Goal of the Paper

Basic Principles for the Design of Complex Systems

A Conceptual Architecture for Control Systems

Examples:

- Example 1: Generating a Conceptual Architecture for a Manufacturing Robot
- Example 2: Generating an Architecture for Tethered UAVs (Unmanned Aerial Vehicles)
- Example 3: Manned-Unmanned Aircraft Teaming

Generating the Physical/Logical Architecture from the Conceptual Architecture

Potential Role in Certification

How a Conceptual Architecture Satisfies the Basic Engineering Design Principles

Conclusions

Appendix: The Contents of a Conceptual Architecture

[1] This draft is still in the revision stage and will likely change. Any comments about how to improve it are welcomed. It is not ready for wide distribution, but limited distribution to those you think will be particularly interested is fine. A more formal version will soon be submitted for publication.

An Improved Design Process for Complex, Control-Based Systems Using STPA and a Conceptual Architecture

Nancy Leveson
Engineering Systems Lab
Aeronautics and Astronautics Dept.
MIT

Introduction

This paper proposes some fundamental changes in the standard practices for the engineering of control systems. Safety-critical control systems (sometimes called cyber-physical systems[2]), have unique properties that are not currently being satisfied by the use of one generalized system engineering process. A basic premise of this paper is that to make progress, we need to more carefully design an engineering process that is created specifically for this type of system. That will not only make it easier to design safety, security, and other emergent properties into these systems from the beginning, but also provide tremendous increases in our ability to assure, operate, maintain, and evolve these systems within reasonable cost limits. It could also have important uses in the certification of safety-critical systems.

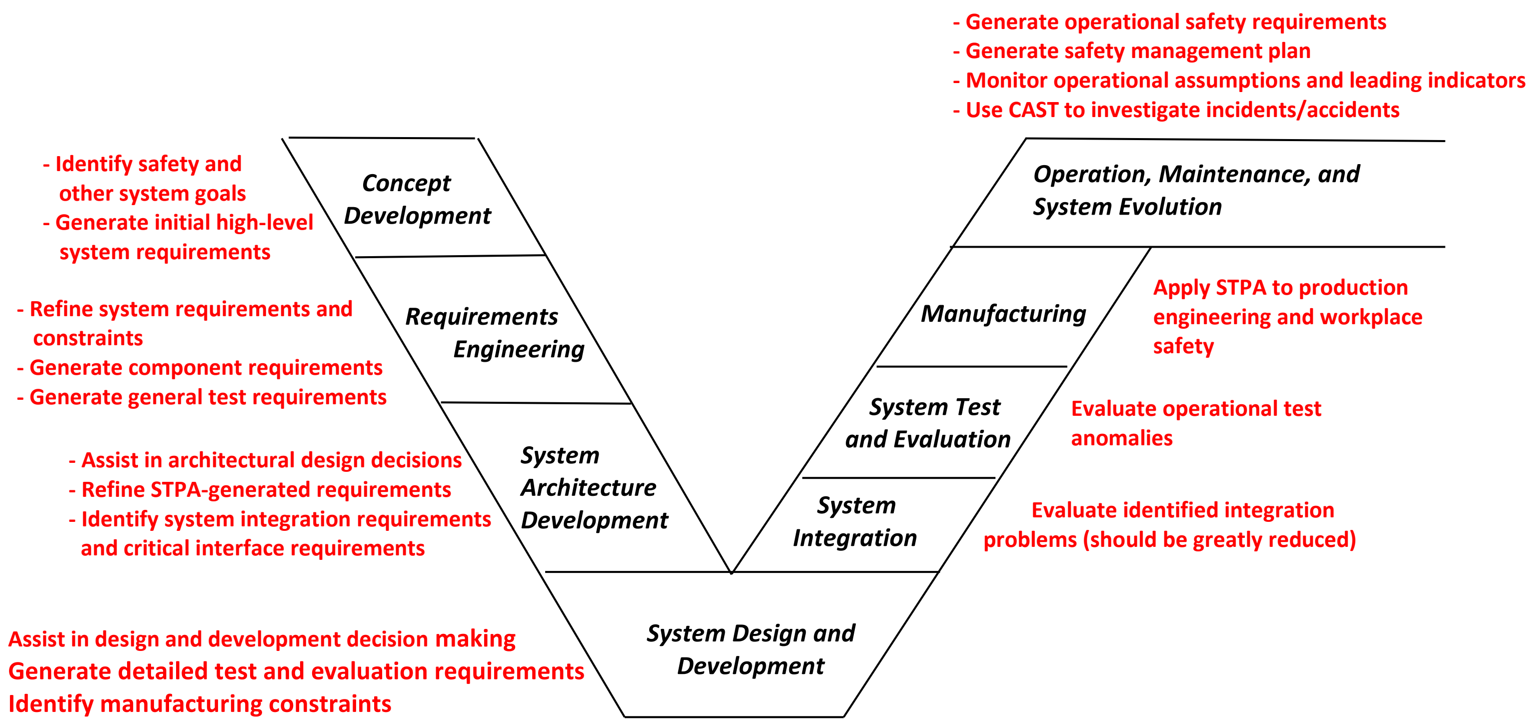
There is an extensive literature on the use of STPA to analyze systems for safety (and other emergent properties such as security, quality, producibility, etc.) throughout the system life cycle of complex, software-intensive systems. These tools are being used successfully in many industries. This paper is focused on a more general topic, i.e., the changes to the basic system engineering process to take most advantage of these powerful new tools and to improve the design and development of safety-critical, control-based systems in general.

Figure 1 shows a standard V-model of the system engineering process, annotated (in red) with the ways that STPA can be used to assist in the process. STPA is not just an assurance or risk assessment process, but a qualitative hazard analysis technique that can be used in a large variety of ways throughout the system life cycle, including operations.

This paper proposes adding a new process (see Figure 2), called Conceptual Architecture Development (shown in green), after Concept Development and Requirements Engineering and before detailed physical/logical architecture development. The new step involves the creation of a conceptual architecture that can potentially reduce costs and improve the results through the life of control systems. The rest of this paper describes how this conceptual architecture can be created and used.

This paper assumes that the reader has a basic understanding of STAMP and STPA. If that is not true, you will first need to read some basic papers in order to understand what is being proposed. Two places to get started are Nancy Leveson (2012) *Engineering A Safer World*, MIT Press and also the STPA Handbook available at http://psas.scripts.mit.edu/home.

[2] The term *cyber-physical systems* (CPS) is used to denote physical and computational systems that are linked to achieve some goal. Control systems is used here as the CPS label is often applied too narrowly or too broadly to include everything that has software in it. The term *control system* is used here to mean systems where software and hardware are used to control physical processes. Humans also need to be considered as components of systems. The term *human-cyber-physical systems* is sometimes used to reflect this.


- Generate operational safety requirements
- Generate safety management plan
- Monitor operational assumptions and leading indicators
- Use CAST to investigate incidents/accidents
- Identify safety and other system goals
- Generate initial high-level system requirements
Concept Development
Operation, Maintenance, and System Evolution
- Refine system requirements and constraints
- Generate component requirements
- Generate general test requirements
Requirements Engineering
Manufacturing
Apply STPA to production engineering and workplace safety
System Test and Evaluation
Evaluate operational test anomalies
- Assist in architectural design decisions
- Refine STPA-generated requirements
- Identify system integration requirements and critical interface requirements
System Architecture Development
System Integration
Evaluate identified integration problems (should be greatly reduced)
Assist in design and development decision making
Generate detailed test and evaluation requirements
Identify manufacturing constraints
System Design and Development

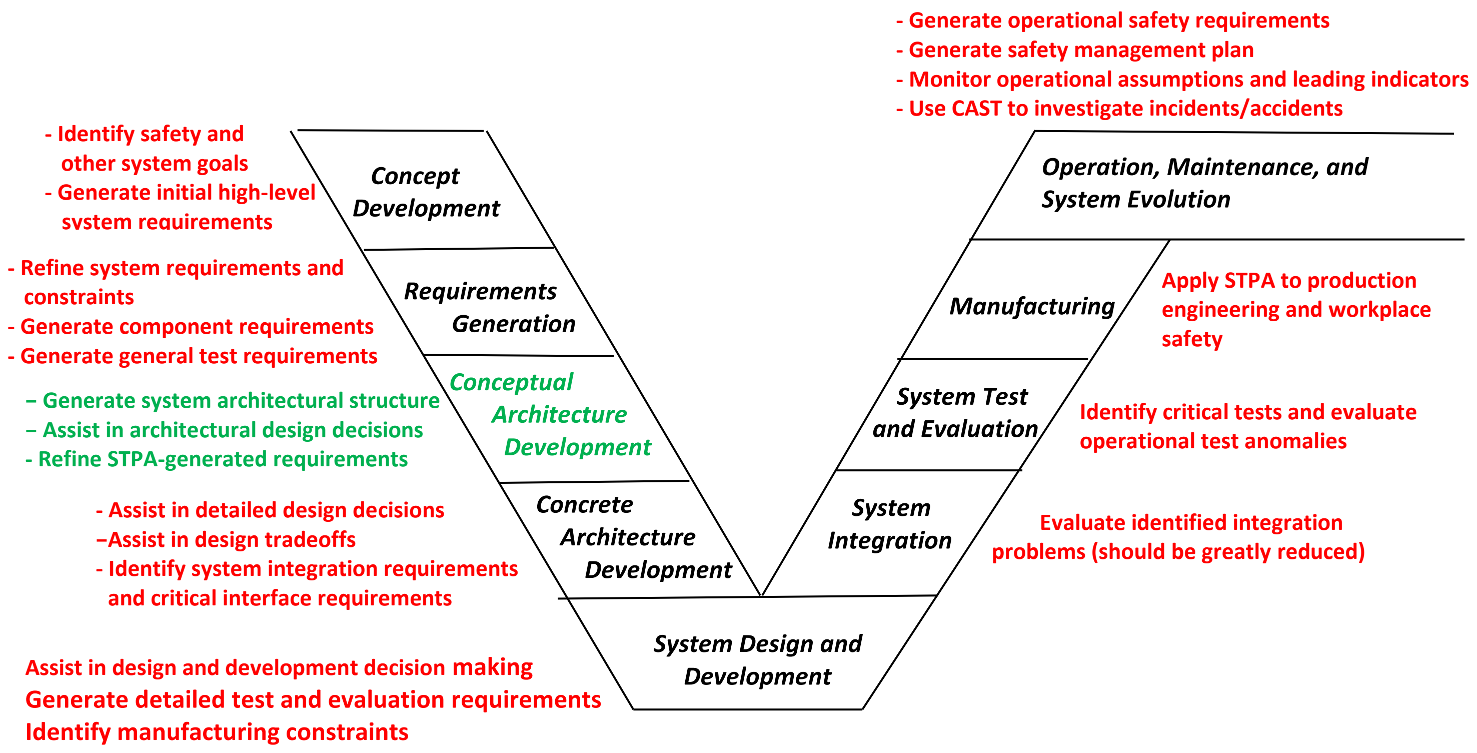

- Generate operational safety requirements
- Generate safety management plan
- Monitor operational assumptions and leading indicators
- Use CAST to investigate incidents/accidents
- Identify safety and other system goals
- Generate initial high-level system requirements
Concept Development
Operation, Maintenance, and System Evolution
- Refine system requirements and constraints
- Generate component requirements
- Generate general test requirements
Requirements Generation
Manufacturing
Apply STPA to production engineering and workplace safety
– Generate system architectural structure
– Assist in architectural design decisions
- Refine STPA-generated requirements
Conceptual Architecture Development
System Test and Evaluation
Identify critical tests and evaluate operational test anomalies
- Assist in detailed design decisions
–Assist in design tradeoffs
- Identify system integration requirements and critical interface requirements
Concrete Architecture Development
System Integration
Evaluate identified integration problems (should be greatly reduced)
Assist in design and development decision making
Generate detailed test and evaluation requirements
Identify manufacturing constraints
System Design and Development

Figure 1 (above) and **Figure 2** (below). Figure 1 shows the standard V-model annotated with with the role of STPA. Figure 2 shows the new conceptual architecture step (in green) added.

The Problem with the Conventional Approach and Goal of the Paper

Large projects usually start by deciding on a set of high-level goals, generating a CONOPS, and then producing requirements to varying degrees of completeness. At this point, developers usually jump into the process of fairly low-level system architecture design. A common strategy for this architecture development is to decompose the system into standard functional components (not necessarily those required or best for this particular system) and begin immediately to specify the connection between the components, often including logical and physical details such as network structures and detailed design as well as interface specifications without adequate consideration of how they will implement the specific functional requirements. "Standard" architectures may even be used, which are basically architectures that have been created for a particular type of system. Examples can be found in this paper.

The problem is that engineers are designing physical interactions among components before they know what connections are important or needed. Developing an architecture for a complex system requires top-down analysis to understand what interactions among system components are necessary and potentially critical.[3] I have seen examples where the first steps in architecture development were to draw a lot of boxes labeled with standard functions and then draw connections between the functions without any tracing to the detailed requirements and desired high-level properties of the system.

The obvious problem with the current approach is that the unique requirements for the system are only vaguely tied to the architecture. At the same time, safety engineering efforts are usually reduced to producing a lot of paper with little impact on the actual system design and resulting safety.[4] Childs called this Cosmetic System Safety, and the problem has only gotten worse since he pointed it out long ago. In addition, security efforts may be delayed until very little impact on system design is possible, resulting in systems that cannot be protected against adversaries. Security becomes an endless and futile game where attackers find a hole and break in, defenders fix that hole, and then the attackers find another hole or find a hole in the new defense.[5]

The negative impact of this disconnect between the system requirements and constraints and the architectural design usually only becomes apparent in later stages of development, where the work involved in showing that the safety, security, and functional requirements are satisfied by the system can be extremely costly and even infeasible. Often, critical design problems are found late in development or during operations, requiring an enormous cost penalty to resolve and sometimes too late to make significant changes in the architectural design. If the architecture is not appropriate for the problem being solved, inefficiencies will occur throughout the system life: Maintaining, evolving, and upgrading the system may be enormously and unnecessarily expensive in terms of time and effort. In some cases, the costs are so enormous that systems have actually been abandoned before they can serve their intended purpose.

The goal of this paper is to suggest a different approach to developing a system architecture that significantly reduces the most resource-intensive aspects of system engineering, particularly those related to designing, assuring, and maintaining system safety, security and other system properties. Effort to achieve these goals is shifted to the early development stages. The most obvious changes from

[3] To avoid getting bogged down with psychological and technical arguments, support for this statement is not provided here. The interested reader might start with Nancy Leveson, Intent Specifications: An Approach to Building Human-Centered Specifications, *IEEE Transactions on Software Engineering*, vol. SE-26, no. 1, pp. 15-35, January 2000.

[4] Charles W. Childs. Cosmetic System Safety, *Hazard Prevention*, May/June 1979.

[5] William Young and Nancy Leveson. An Integrated Approach to Safety and Security Based on Systems Theory, *Communications of the ACM*, Vol. 57, No. 2, Feb. 2014. Downloadable from http://sunnyday.mit.edu/papers/cacm232.pdf

what is commonly done today is (1) the insertion of a safety and security modeling and requirements analysis process using STPA during early system requirements generation (which has been described previously) and (2) a new activity at the beginning of the system architecture and development step that generates a *conceptual architecture* (Figure 2). This paper concentrates on the new conceptual architecture step.

Conceptual architecture generation provides a design, modeling, and analysis step between English language requirements and the development of a detailed logical and physical system design. The added step bridges the gap from a large number of “shall” statements to a detailed architectural design that is better structured and rational for the specific system requirements and constraints. The resulting architecture will enhance our ability to perform later system engineering processes and achieve the system goals.

Architectural validation can start at this point because the conceptual modeling process provides a concrete tracing between requirements and design. Such a tracing should also make it easier to implement changes and upgrades with more assurance that requirements and constraints are still satisfied.

This paper focuses on the left side of the V-model (the beginning steps in system engineering), but changes to other parts of the process will also be helpful. Some companies, for example, are currently using STPA to redesign their production engineering process and to assist in design for manufacturability, and even to optimize their general engineering and supply chain practices.

The goal of creating a conceptual architecture is to improve the design process for control systems. Therefore, the conceptual architecture and the process for creating it should incorporate basic principles for design of complex systems. These are described in the next section and are used to justify the decisions made about conceptual architecture design in the rest of the paper.

Basic Principles for the Design of Complex Systems

Computer science and, in particular, software engineering has had to cope with complex design problems for a long time. In the 1970’s, many of the basic principles of designing complex systems were identified during the development of what was then called structured programming. Most of the principles of structured programming were adopted into fundamental practices and the need for a special name for them disappeared. But what was learned is applicable for complex system design today—not just for software. They are described here and used in the rest of the paper to explain why various decisions were made in the new conceptual architecture development process being proposed. I have left out all but a few major references as there are too many to be relevant in a paper not about the history of design. Many, if not most of these principles, have been documented and taught for a very long time. Table 1 summarizes them.

1. *Good design involves mastering complexity*.

A first principle is that the goal of design and the design process is to *master complexity*. As the complexity of our systems increases, the design process itself becomes more complex. As an example, in the new highly automated aircraft, most incidents and accidents have been blamed on humans, but actually reflect the difficulty of the collateral design of the aircraft, the avionics systems, the cockpit displays and controls, and the demands placed on the pilots. We need to master the complexity inherent in designing these types of systems and provide techniques that assist designers in this process.

Complexity can be defined as intellectual unmanageability. We need design processes that stretch our intellectual limits and help us to gain traction over the complexity of the systems we want to build.

Table 1: Engineering Design Principles and Approaches

Basic Design Principles:
1. Good design involves mastering complexity.
2. The design process should assist in understanding the problem to be solved.
3. Successful design depends on the design strategies and specification methods used.

Successful design involves breaking down the design process into successive steps:
1. Each with a lower complexity than the system itself,
2. While minimizing interactions among the parts,
3. Such that the parts together solve the problem.

Design approaches to deal with complexity:
1. Separation of concerns
2. Restricted visibility (locality of information)
3. Abstraction
4. Simplicity: Design should match the structure of the problem being solved
5. Proper ordering of design decisions

What types of tools can help us master complexity? A clue rises from research by Bill Curtis and others[6] on comparing the critical factors distinguishing between successful and unsuccessful software-intensive projects. They found that exceptional designers *"think on a system level."* They can see the system (the entire system, not just the software) as a whole and not just the separate parts. To be successful, a project does not need to be composed of all system thinkers, but at least one well-placed individual needs to have this perspective. *Our tools should assist in such thinking by providing views of the desired system behavior and encourage thinking of the system as a whole and not just as a combination of its parts.* Such system thinking needs to be done before performing a deep dive into more detailed design and decomposition of components in order to ensure that the resulting design will satisfy the system goals and constraints.

2. The design process should assist in understanding the problem to be solved.

Design involves *problem solving*, i.e., creating a structure for solving a problem. An example is creating a flight control system to control an aircraft in such a way that efficiency, safety, and other goals (requirements) are achieved. Understanding the problem to be solved is critical in producing a useful design. But all the necessary understanding may not exist at the beginning of development. Therefore, one goal of a design process is to assist in enhancing our understanding of the problem before design decisions are made that cannot be undone. In other words, the design process should proceed through steps that allow us to increasingly learn more about the problem we are trying to solve as we generate solutions.

[6] B. Curtis, H. Krasner, and N. Iscoe. A Field Study of the Software Design Process for Large Systems, *Communications of the ACM*, vol. 31, no. 2, pp. 1268-1287, 1988.

In addition, design methods impose demands on the humans using them and have a profound impact on our problem-solving ability and the strategies we use.[7] Our goal should be to create and use design methods that impose reasonable demands on the designers and assist them in carrying out their tasks. The way we go about designing complex systems will have an impact on our problem-solving ability and on the errors that we make while trying to solve the problems and create a good design. Our engineering processes, therefore, need to reflect what is known about human limitations and capabilities, not just those of the humans who will eventually operate the systems that will result from the design process but also the limitations and demands on the designers themselves.

3. *Successful design depends on the design strategies and specification methods used.*

A problem-solving activity involves achieving a goal by *selecting and using strategies* to move from the current state to the goal state. Success depends on selecting an effective strategy or set of strategies and obtaining the information necessary to carry out that strategy successfully. Success also depends on using design specifications and techniques that assist in carrying out the selected strategy successfully.

Cognitive psychology has established that the representation of the problem provided to problem solvers can critically affect, i.e., degrade or support, their performance. David Woods, a cognitive psychologist, claims that there are no neutral representations. For example, the representation(s) of the system used during design can ideally reduce memory load on the designers or display the critical attributes needed to solve the problem in a perceptually salient way and thus enhance the potential for success. Some problem representations can also make solving the problem more difficult. The same is true not only during the original system development but for later maintenance and evolution activities.

Designs are created and used by humans. This observation has led to the concept of human-centered design processes and user-centered design artifacts. System engineering processes should enhance human processing and not just processing by automated tools. Usually, model-based design tools are able to check only for relatively simple properties, such as the inclusion of specific types of information or connections. Even if automated tools are used, humans need to be able to understand and check their results to ensure they satisfy the overall project goals and satisfy the required system properties. Such human review cannot be fully automated. If a choice needs to be made, human reviewability should be prioritized over ease of automation.

Important Aspects of a Good Design Process and Tools

Keeping these three general principles in mind, we can now consider more detailed aspects of creating a good design process. The design problem can be characterized as how to break the design process into successive steps:

- Each with a lower complexity than the system itself
- While minimizing interactions among the parts
- Such that the parts together solve the problem.

There is no universal way to accomplish this. Many methods have, of course, been suggested but none seems to be optimal for all problems.

Design Approaches to Deal with Complexity

[7] Nancy Leveson, Intent Specifications: An Approach to Building Human-Centered Specifications, *IEEE Transactions on Software Engineering,* vol. SE-26, no. 1, pp. 15-35, January 2000.

Along with these general principles, there are five more specific design approaches that have been found to be useful as system complexity increases:

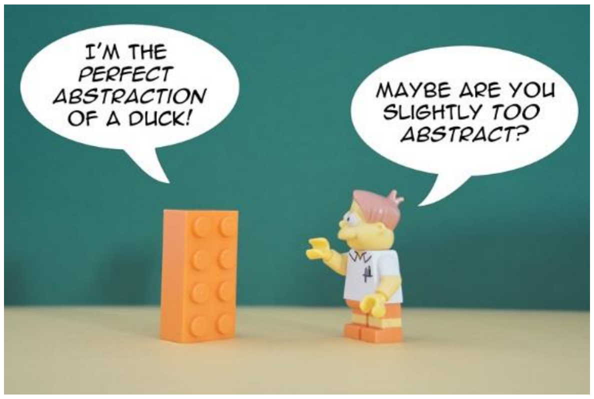
(1) Separation of Concerns: Designers need to deal with the different aspects of a problem separately. Our minds are just not capable of coping with and resolving all concerns at once. Instead, we need to use design methods that allow us to separate decisions into smaller ones that we are able to handle one at the time—while still keeping in mind the system-level concerns that the decisions are affecting, For example, designers should be able to separate concerns about identifying *what* functions their system must provide from decisions about *how* to provide them. Or they should be able to separate concerns about what types of communication are necessary and between what system components from concerns about the design of the communication channels themselves.

(2) Restricted visibility (locality of information): The design process should allow designers (and design components) not to need to understand the whole system in order to make decisions. This seems to contradict the "whole picture" requirement, but what it means is that while the whole picture is needed for global decision making, the information to make local decisions needs to be available locally and not require understanding the entire very complex system for every local decision. One implication is that design components should communicate only through well-defined interfaces. Pieces of the system should not be allowed to communicate in an unorganized fashion. The latter prevents the designers from being able to make decisions without understanding all the details of the larger system, usually impossible in systems today. A few designers should concentrate on the bigger picture and not get involved in the details. Once the larger picture is understood, it should be possible for other designers and tools to make local decisions without violating the coherence of the overall global design.

Designers are sometimes taught to concentrate on minimizing, ordering, and making explicit the connections between modules separate from the uses of those connections. Before designing connections, however, it is important to understand how those connections will be used. This is an explicit goal of creating a conceptual architecture. Simply minimizing connections or making them explicit does not provide much assistance in producing a better system design.

(3) Abstraction: The problems in building systems that satisfy stakeholder goals are rooted in complexity and intellectual unmanageability, as already stated. If humans want to successfully build and operate increasingly complex systems, we need to increase the intellectual manageability of our designs and design processes. That is, we need to find ways to augment human ability. Jens Rasmussen has observed that complexity depends on the level of resolution upon which the system is being considered. A seemingly simple object becomes complex if observed through a microscope. A way to cope with complex systems is to structure the design process such that the designer can transfer a particular problem being solved to a level of abstraction with less resolution.

Figure 3 shows two abstractions for a "duck." While usefulness depends on the goal of the abstraction effort, the first one eliminates so much information that it is difficult to think of any important uses. The usefulness of the one on the right depends on what use is intended.


I'M THE PERFECT ABSTRACTION OF A DUCK!
MAYBE ARE YOU SLIGHTLY TOO ABSTRACT?

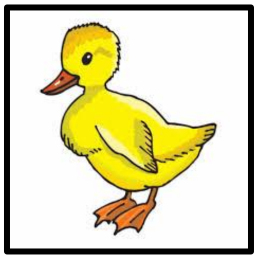
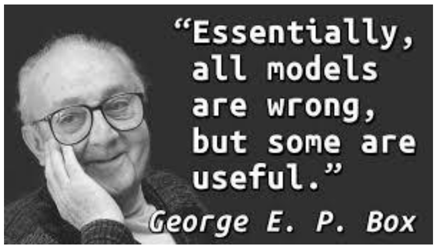
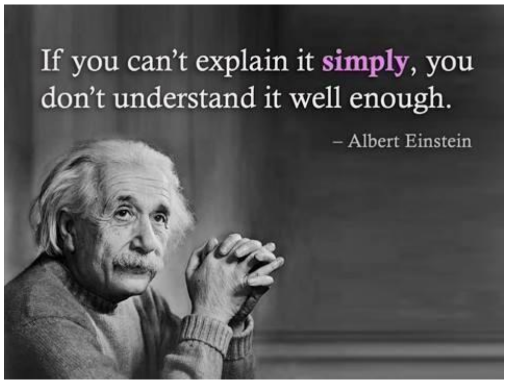


Figure 3: Two abstractions of a duck. The one on the left does not seem very useful. The usefulness of the one on the right depends on the intended use of the abstraction.

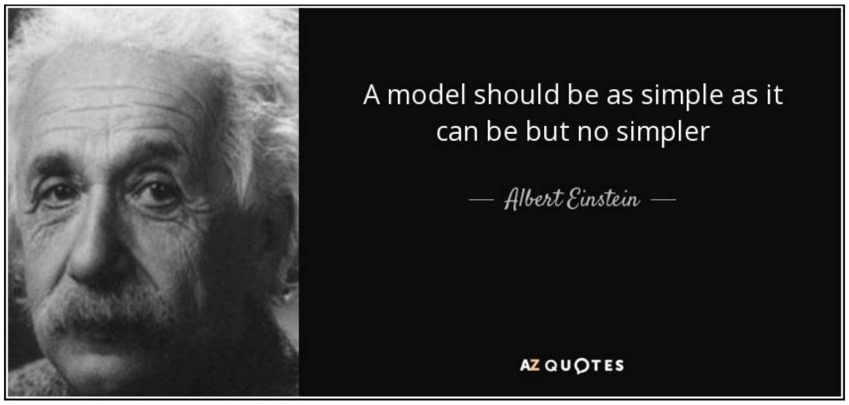
Abstractions or models are not right or wrong, they only have comparative effectiveness. George Box famously argued that "All models are wrong; some models are useful." That is, all models are incomplete. They are an abstraction placed on a messy world to make it appear less "messy." By definition, an abstraction leaves out something—otherwise it is not a model (i.e., abstraction) but the thing itself. We hope that the model does not omit anything important with respect to our goals for the use of the model.:



We like simple models, and a simple model that is useful is indeed a good thing: omitting irrelevant information assists in using the model as intended. Einstein said:


If you can't explain it simply, you don't understand it well enough.
– Albert Einstein

But at the same time:



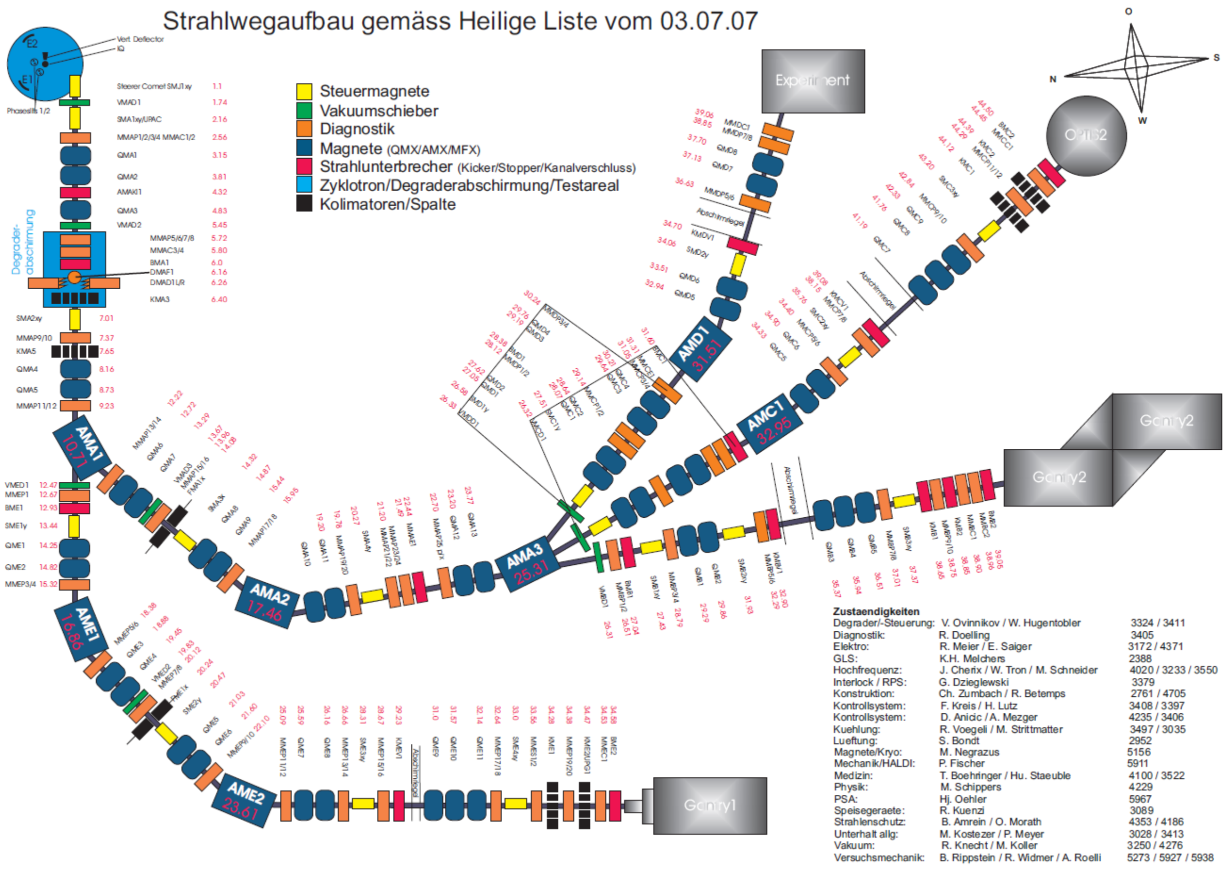
Abstraction is essentially the process of identifying the important aspects of a phenomenon to assist in solving a problem (the goal of the abstraction) and ignoring details that are irrelevant at the current stage of problem solving. Not only does this allow us to concentrate or focus on smaller problems, but it also acknowledges that we cannot completely understand problems until we start to solve them. Many of the details may be too vague at any point to include them in the design process until later. In STPA, the control structures use hierarchical abstraction, where the abstractions used at any point in the process can include varying levels of detail.

Figure 4 shows the very high-level physical structure abstraction for a radiation treatment system called the Gantry 2 that uses protons to treat patients. Figure 4 is a useful abstraction for the engineers that are designing the physical linear accelerator that produces the protons for Gantry 2.

In contrast, Figure 5 shows the very high-level control structure (abstraction) for treatment by the Gantry 2 system. Notice that this abstraction does not provide details about ordering of the functions or activities (and definitely not the physical design of the Gantry 2) but simply the control structure. Notice how different these abstractions (Figure 4 and Figure 5) are although they are abstractions of the same system. The difference is in the information included and what is excluded.

The high-level abstraction shown in Figure 6 shows more detail about the control structure within the Treatment Definition box. Figure 7 elaborates on the Treatment Delivery abstraction. The final abstraction delves more deeply into the control of beam and patient alignment and the physical devices being controlled. Boxes with the names of all the high-level control structures (but not the details) such as treatment planning and treatment delivery are left in each more refined abstraction in order to provide context and help the user see the larger picture within which the more detailed control structure fits.

Using different levels of abstraction assists in creating the system in a step-by-step process as the problem is better understood and more detailed design decisions are made.



Zustaendigkeiten		
Degrader/-Steuerung:	V. Ovinnikov / W. Hugentobler	3324 / 3411
Diagnostik:	R. Doelling	3405
Elektro:	R. Meier / E. Saiger	3172 / 4371
GLS:	K.H. Melchers	2388
Hochfrequenz:	J. Cherix / W. Tron / M. Schneider	4020 / 3233 / 3550
Interlock / RPS:	G. Dzieglewski	3379
Konstruktion:	Ch. Zumbach / R. Betemps	2761 / 4705
Kontrollsystem:	F. Kreis / H. Lutz	3408 / 3397
Kontrollsystem:	D. Anicic / A. Mezger	4235 / 3406
Kuehlung:	R. Voegeli / M. Strittmatter	3497 / 3035
Lueftung:	S. Bondt	2952
Magnete/Kryo:	M. Negrazus	5156
Mechanik/HALDI:	P. Fischer	5911
Medizin:	T. Boehringer / Hu. Staeuble	4100 / 3522
Physik:	M. Schippers	4229
PSA:	Hj. Oehler	5967
Speisegeraete:	R. Kuenzi	3089
Strahlenschutz:	B. Amrein / O. Morath	4353 / 4186
Unterhalt allg:	M. Kostezer / P. Meyer	3028 / 3413
Vakuum:	R. Knecht / M. Koller	3250 / 4276
Versuchsmechanik:	B. Rippstein / R. Widmer / A. Roelli	5273 / 5927 / 5938

Figure 4. A model (abstraction) of the physical design of the Gantry 2.

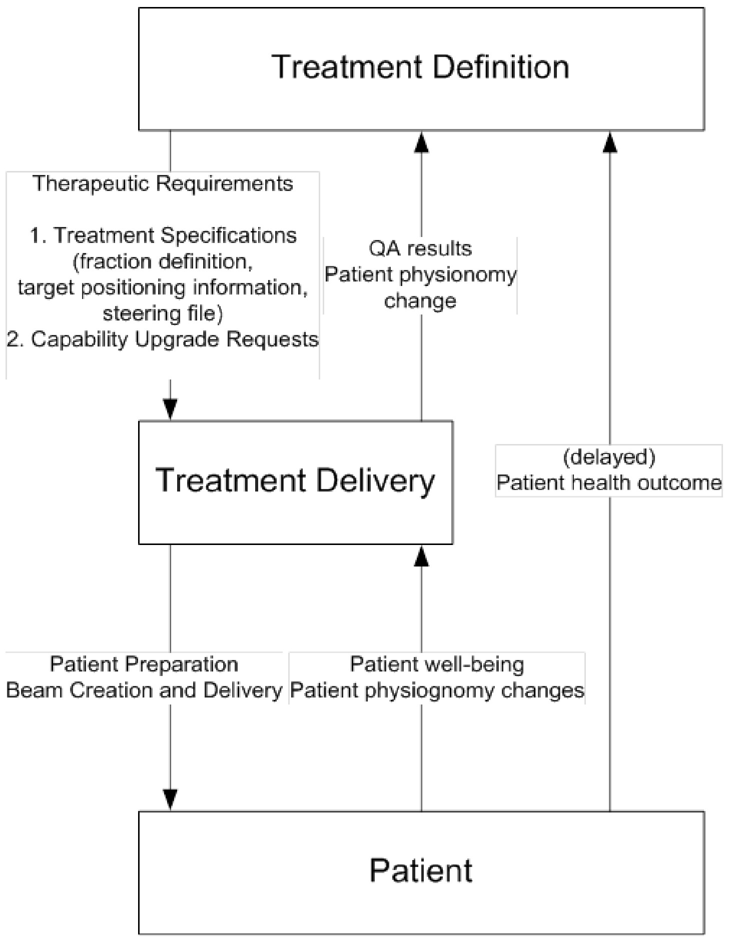

Treatment Definition
Therapeutic Requirements
1. Treatment Specifications (fraction definition, target positioning information, steering file)
2. Capability Upgrade Requests
QA results
Patient physionomy change
Treatment Delivery
(delayed) Patient health outcome
Patient Preparation
Beam Creation and Delivery
Patient well-being
Patient physiognomy changes
Patient

Figure 5. High-level abstraction of treatment by Gantry 2

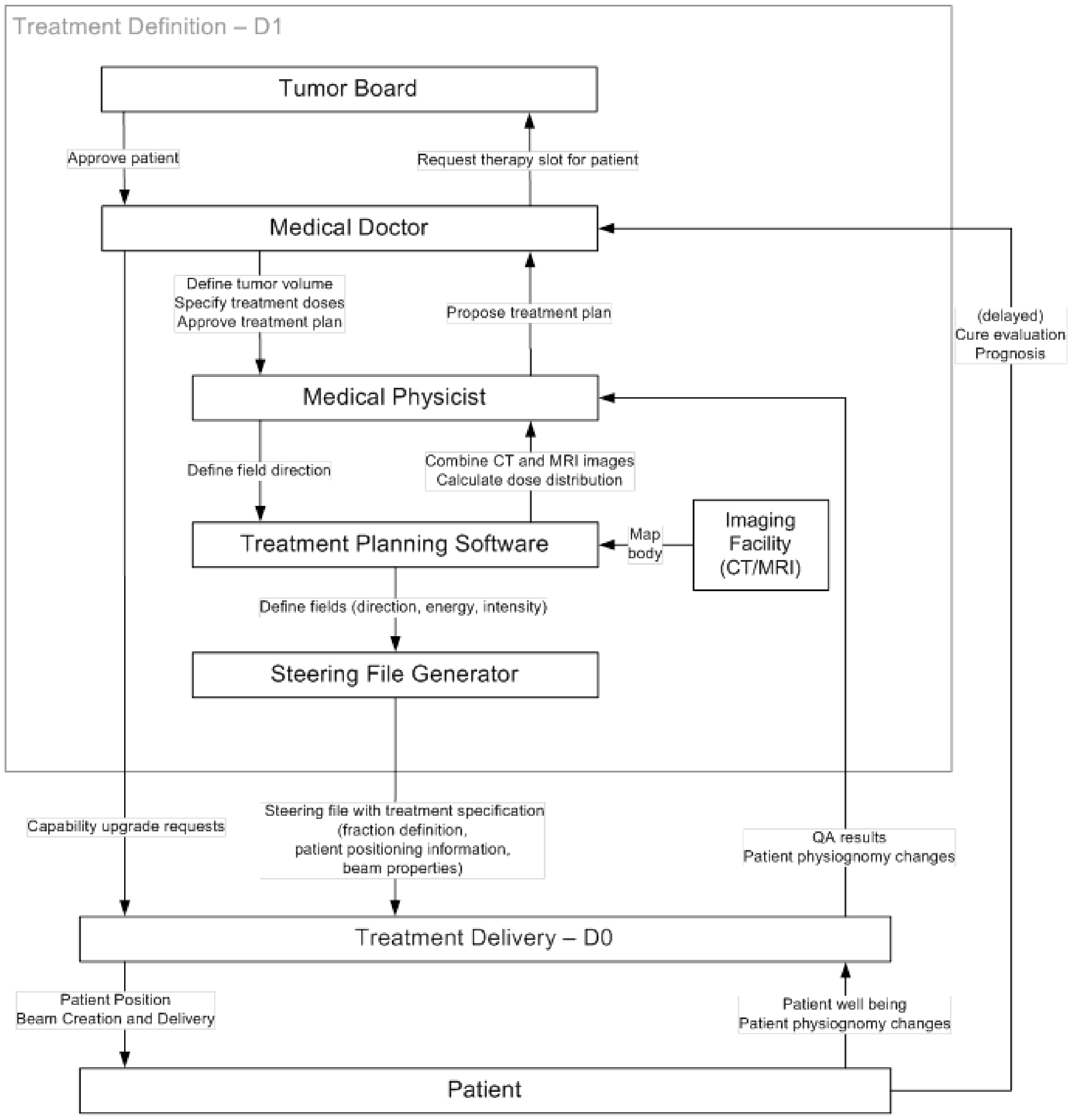

Treatment Definition – D1
Tumor Board
Approve patient
Request therapy slot for patient
Medical Doctor
Define tumor volume
Specify treatment doses
Approve treatment plan
Propose treatment plan
(delayed)
Cure evaluation
Prognosis
Medical Physicist
Define field direction
Combine CT and MRI images
Calculate dose distribution
Treatment Planning Software
Map body
Imaging Facility (CT/MRI)
Define fields (direction, energy, intensity)
Steering File Generator
Capability upgrade requests
Steering file with treatment specification (fraction definition, patient positioning information, beam properties)
QA results
Patient physiognomy changes
Treatment Delivery – D0
Patient Position
Beam Creation and Delivery
Patient well being
Patient physiognomy changes
Patient

Figure 6. More detail about treatment definition.

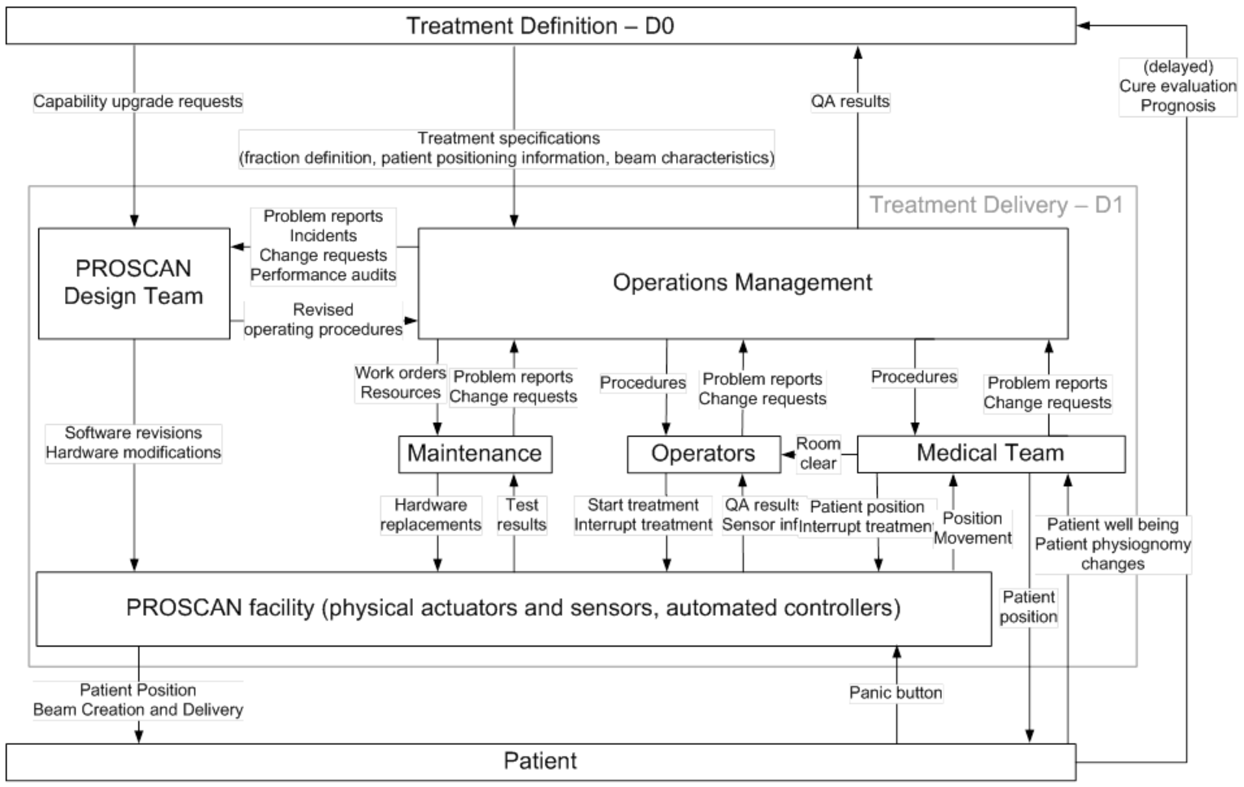


Figure 7. Zooming in on treatment delivery.

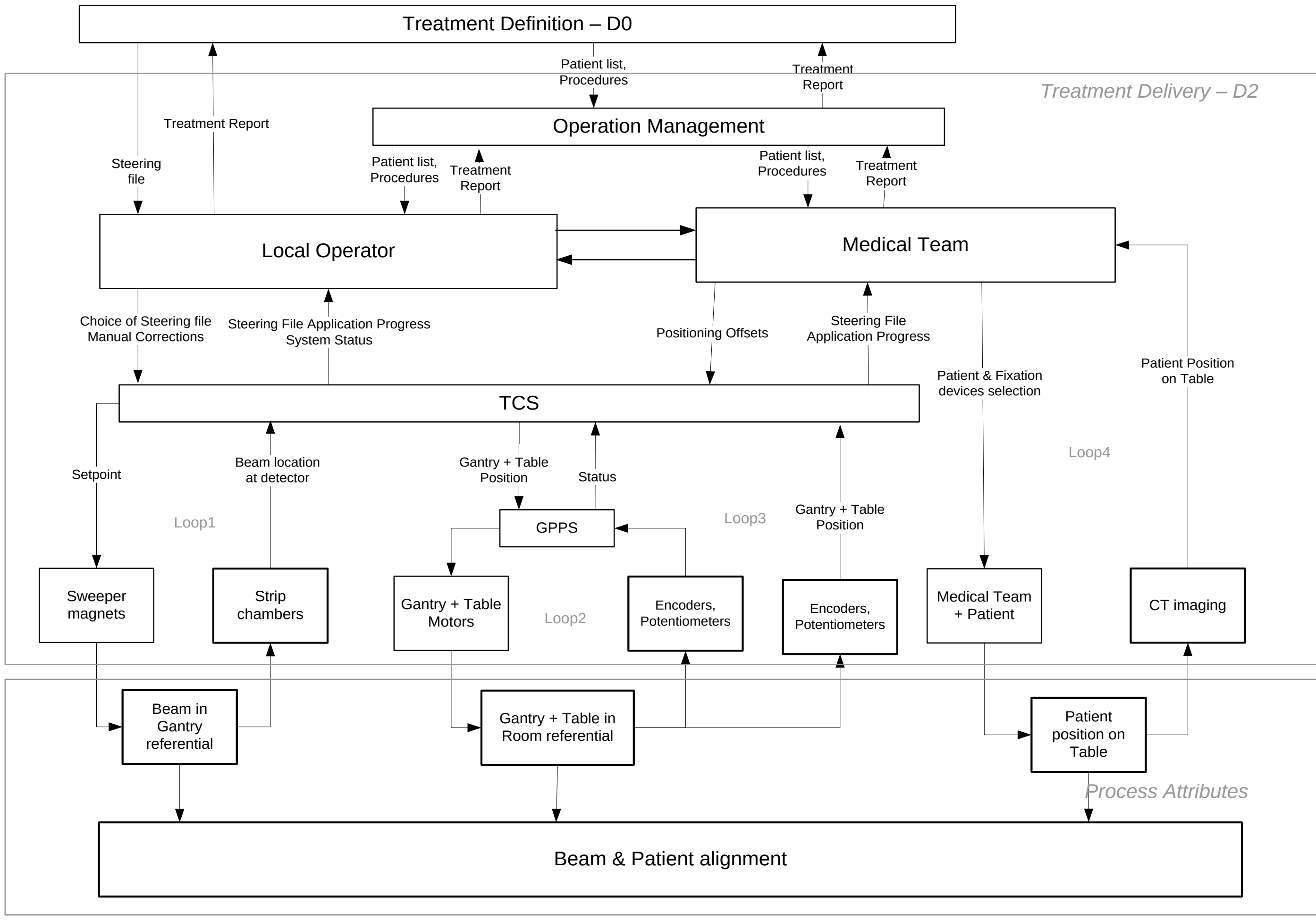


Figure 8. Including even more detail about treatment delivery.

There are four basic types of hierarchical abstraction, which determine what details we will include and what we will exclude:

- Procedural (functional) abstractions specify the system functions and how they will be performed. Examples include a flow chart or sequence of tasks (the task flow) to be performed or the sequence of decision making by the system controllers.
- Data abstraction concentrates on the types of data in a problem solution and how the data will be manipulated. Data can be considered at various levels of detail, such as files, records, and fields.
- Object abstraction focuses the design process on objects (usually physical but they could be logical) in the system design and the operations that need to be performed on the objects. In object-oriented design, focus is on the interacting objects needed to achieve the system requirements. While related to data abstraction, the objects represent physical objects that use or manipulate data. Objects for a plane might be the engines, fuel tanks, control surfaces, pilots, etc. Figure 4 shows the physical objects in Gantry 2 along with their connections.

- Control abstraction abstracts from the precise sequence of events to instead concentrate on the system (emergent) properties that need to be controlled. In an aircraft, the control abstractions describe how desired system (emergent) properties such as trajectory (navigation), lift, propulsion, etc. are achieved and maintained. The things controlled may be objects (e.g., the control surfaces or the engines), but the abstraction is not focused on the objects themselves but on the types of control that must be provided over those objects and their interactions and how they are controlled to achieve the overall system goals and constraints. For Gantry 2, the control abstraction shows the control structure, the control actions, and the feedback between controlled components and controllers as shown in Figures 5 to 8.

While any of these types of abstractions might be used in problem solving, the one that allows us to deal with the most difficult aspects of the specific problem being solved should be preferred. For some problems, identifying the data and how it needs to be manipulated is the most important and difficult part of the search for a good design. In others, the computation of the functions to solve the problem may be critical. In a control system, the control aspects of the problem are usually the most critical.

In practice, it is helpful to have multiple types of models and abstractions of the system to be used during development and operations. Those abstractions or models will assist in different aspects of the development process. Unfortunately, much of the design literature has been produced by people who design systems where data or object abstraction is the most useful. In addition, tool developers and purveyors find it much more profitable to provide only one tool for all system designers. Object-oriented design (OOD) is often used, then, in the basic design of systems for which a different type of abstraction would be more helpful. Using object-oriented design tools, which have become the focus of almost all tool developers, to design systems for which other types of abstractions are better suited can result in designs that are more difficult to understand, to assure, and to change. It may also increase the difficulty of performing the design process itself if abstraction based on objects is not the most natural for the system being created.

Abstractions and models are used by *design methods*, defined as the general processes used to create the system design. The goal of a design method should be to make it easy for users to extract and focus on the important information needed at that point in the design process. While OOD has advantages for *implementing* a design concept, it may not be the best way to *create* that design concept. The difference is what DeRemer and Kron labeled in software engineering as “programming in the large versus programming in the small.” Designing the overall structure of the software (programming in the large) may require different types of languages and tools than designing the software system components (programming in the small).

As another example, Figures 9(a) and 9(b) show two models of the same spacecraft. Figure 9(a) shows the objects used to construct the spacecraft and their connections while Figure 9(b) shows the high-level control structure that is implemented by those objects. Figure 9(c) zooms into and shows more detail of the abstraction (control structure) in Figure 9(b). While Figure 9(a) has more detail, even if that detail was eliminated, Figure 9(a) contains very different information than Figures 9(b) and 9(c). Different abstractions support different engineering activities.

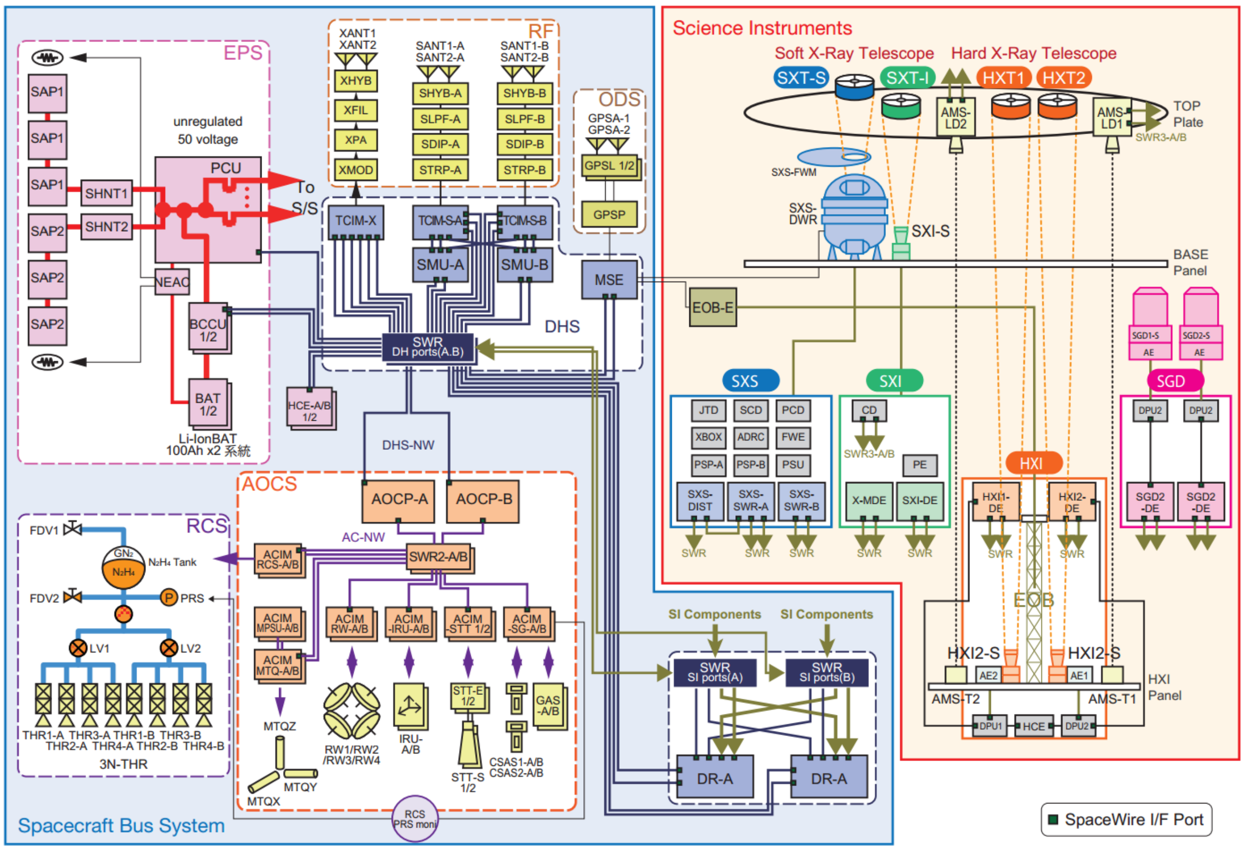


Figure 3.9: System block diagram. A is the primary and B is the redundant system.

Figure 9(a). Spacecraft physical/logical design

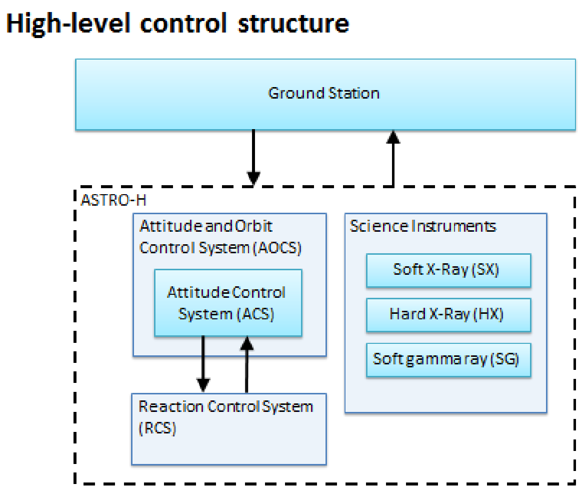

High-level control structure
Ground Station
ASTRO-H
Attitude and Orbit Control System (AOCS)
Attitude Control System (ACS)
Reaction Control System (RCS)
Science Instruments
Soft X-Ray (SX)
Hard X-Ray (HX)
Soft gamma ray (SG)

Figure 9(b). High-level spacecraft control structure

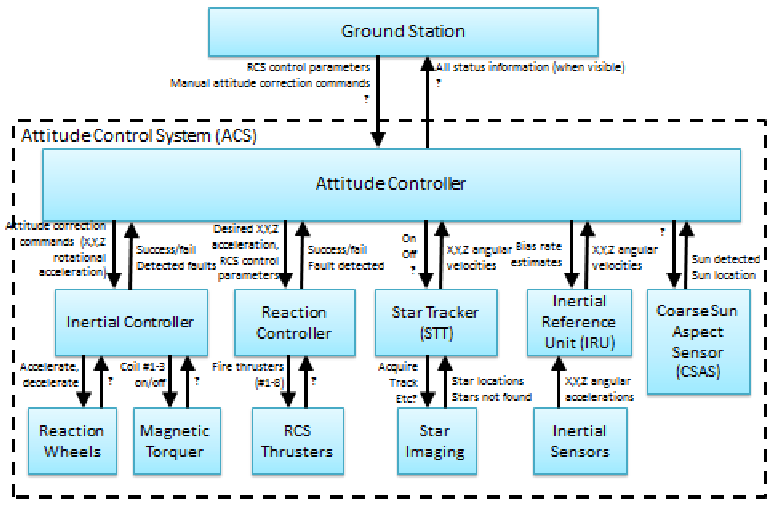


Figure 9(c). More detailed abstraction of the spacecraft control structure

So far, hierarchical abstraction has been described. Another useful abstraction is one that Jens Rasmussen called a "means-ends" abstraction and I have called an intent abstraction.[8] It abstracts the system on the basis of goals, constraints, and design rationale. Basic hierarchical abstraction can be thought of as providing "what" information at one level while the next lower level describes "how." Such hierarchies, however, do not provide information about "why." An intent abstraction provides goal-oriented links between the levels of system abstraction. Engineering a Safer World, Chapter 10, provides more information about Intent Specifications as does the more technical paper cited at the bottom of this page.

In summary, the abstractions at each point in development must support the various types of formal and informal analysis used to decide between alternative designs and to verify the results of the design process. They should also assist in the coordinated design of the components and the interactions among them.

There is no need to pick only one of these types of abstraction. Hierarchical, decompositional, and intent may be combined in various ways to assist in the design process. In fact, the combined use of these different types of abstractions will assist in the coordinated design of the components and the interfaces. The only reason for limiting ourselves to using only one type of model in system engineering is to optimize the profits of tool developers and minimize the educational process for designers. Neither of these is likely to optimize the system engineering process. MBSE will never deliver its promise until the concept includes multiple types of models to provide different information about the system.

(4) Simplicity: The fourth general design principle is simplicity. Emphasis during design should be on clear and simple designs that are easy to check, understand, and modify. This principle is related to the use of abstractions that focus on the most difficult aspects of the problem, which will influence the simplicity and ease of using the final design. But there is more to consider here. An important basic principle related to simplicity is that the design (solution) should match the structure of the problem being solved.

[8] Nancy G. Leveson, Intent Specifications: An Approach to Building Human-Centered Specifications, *IEEE Transactions on Software Engineering*, Vol. 26(1):15-35, January 2000.

Why is this true? When we solve problems or try to understand a system design, we build models in our minds of the structure of the problem and the solution. The difference between the model in the specifications we use and the model we have in our heads is called *semantic distance* or (the opposite) *semantic similarity*. In the process of creating a system and using one, we must translate the model on paper or a computer screen to the one in our head to understand it, find flaws in it, and when finished use it. The smaller the semantic distance between the design model (specification) and our mental model, the easier it is for designers to create and find errors in the design. I personally have found, for example, that when people try to translate an STPA control structure into a SysML object-oriented model, it is extremely difficult and often impossible, for me to view the resulting structure as a control loop. They have taken the components of the control structure apart and labeled them with their *role* in the control loop using boxes and arrows and other notations that are not in my mental model of a control loop. While such translation of intuitive human models to standard computer models certainly can be done, the more important question is *should* it be done. I recognize that this is a very controversial statement, but I believe that while an object-oriented model may be appropriate at lower levels of abstraction or for specific decomposed system components, it is not appropriate at the basic system conceptual level for control-oriented systems such as aircraft, nuclear reactors, or spacecraft. The design of STPA reflects that belief.

(5) <u>Proper Ordering of Design Decisions</u>. Ordering of decisions is also important. The first decisions made should be general ones that are unlikely to change, i.e., those that will be shared by all potential designs. Detailed design decisions should be put off as long as possible, in general, so that they can reflect the greater understanding of the problem that evolves during the design process. That will reduce the amount of backtracking required.

To summarize, the goal of the design process is to master complexity, i.e., it must make the problem and its solution intellectually manageable. The key to intellectual manageability is the structure of the artifacts used in design and the structure of the final design. The design (solution) structure should fit the problem structure, so as to reduce semantic distance between the design as it is created and the model of the system that exists in the minds of the designers and system experts who will be working with the proposed solution, for example those reviewing the design structure or those creating design artifacts such as testing plans. It must also reflect the types of mental models that users of the system form.

Introducing a Conceptual Architecture into the System Engineering Process for Control Systems

We are now ready to consider how to improve the architectural design process for control-oriented (human-cyber-physical) systems. STPA is a hazard analysis process, not a design process. It generates requirements and information for designers, but does not itself directly assist in the architecture generation and system design and specification processes.

Instead, applying STPA results in a set of system requirements, constraints, and scenarios that can lead to the violation of the system constraints. These results can be used by engineers in the architectural design process and augmented as more decisions are made.

Hazard analysis, in general, is usually considered to be a separate or side activity that is not part of the design or V-model process. The results are primarily used for after-the-fact assurance of the system when the design is basically finished. In reality, of course, safety and security cannot be assured after the fact unless these properties are already there. Emphasis, then, should logically be placed on the design generation process rather than on the assurance or assessment process. Because of its different types of results and process, STPA can be completed earlier than the traditional hazard analysis techniques, which require a detailed design to already exist.

When a conceptual architecture is inserted, better decisions can be made at the earliest possible time in the system engineering process. STPA results are used to generate a conceptual system architecture at a higher level of abstraction than usually found today in architectural design languages and designs (Figure 2).

Two questions immediately arise: (1) why would anyone want to do this and (2) how can it be done.

Why would anyone want to do this?

In the standard V-model, going from a high-level conceptual view of a system or CONOPS, agreed upon by the stakeholders, to detailed requirements and then to a physical/logical architecture requires a lot of big jumps without having much assistance in making the design decisions involved. These jumps need to be simplified and assistance provided in making them if we want to produce better designs.

The introduction of modeling and analysis tools, called Model-Based System Engineering, seems like an obvious potential solution, but most of the models used today involve designing the detailed architecture itself and also use very limited types of models, often only one (such as SysML). In fact, most of the modeling languages and types of models used in MBSE are quite old and date from the mid-1980s. They also focus only on object-oriented design. For control systems, object-oriented design (OOD) is not the most appropriate system design paradigm—although the systems can be forced into such a design by contractual requirements imposed to use commercial MBSE tools.

OOD emphasizes (i.e., builds an abstraction of) the system objects. As a result, control functions may be spread over many or all of the objects and be nearly impossible to validate and verify. For example, in designing an aircraft avionics system, control requirements such as navigation (control over the route), propulsion (control over acceleration), and lift (control over the trajectory) may be a better focus during design, particularly early design and requirements analysis, than on the physical objects such as the horizontal stabilizer, the engines, the fuel tanks, the slats and flaps, etc. It is more efficient to start from considering the requirements for navigation or propulsion and determine what that implies for the design and operation of the physical components (aircraft objects) than to start from the objects and try to determine what functions they might be used to implement. And, as noted, the control functions, such as navigation or trajectory control, may be spread throughout the design and associated with multiple objects, making it more difficult to verify that they are handled correctly.

Getting away from this object-orientation to a control orientation will require the use of new types of models and analysis methods than are used today. In this paper, a more control-oriented modeling and structured design process is proposed. Although safety and security are emphasized, the approach is appropriate for any emergent (system) properties. Remember, object-oriented design approaches may be appropriate later in the design process for control systems when detailed component (vs. system) design is being considered.

A second reason for generating a conceptual architecture as described in this paper, is to augment our ability to produce user-centered designs. We blame most accidents on the operators (pilots, drivers, etc.) but have few tools that can forge an effective partnership between human factors experts who are designing system interfaces (control panels, displays, physical controls) and operator procedures and the engineers who are focusing on the physical (hardware) and logical (software) parts of the system. Too often today, these two groups work relatively independently and we end up creating systems with the potential for mode confusion, situational awareness problems, etc. These problems need not have been created if the designers worked together as an integrated team.

How can this new process be implemented?

The approach being suggested starts from a foundation in systems theory and involves the use of hierarchical control structures. It begins after the basic system requirements ("shall statements") have been specified and validated. The requirements are used to guide the generation of control system designs to implement those requirements. The greatest success will follow from the use of a specification method that is human-centered (not software-tool centered) and incorporates intent and design rationale within the specification language itself and not as an "add on." The tools should minimize semantic distance. These properties are incorporated in "intent specifications,"[9] but this topic is only peripherally touched on in this paper. Other specification approaches are possible.

If the design process for control systems focuses on how to create an appropriate abstraction (model) of the problem being solved, then the effort involved in verifying that the design satisfies the requirements will be greatly reduced. Most of the formal specification languages promoted by computer scientists are too unreadable by the application engineers to be useful here. The most critical property of any analyzable specification language is that the specifications are readable by the application experts. A second (related) goal is to minimize the semantic distance between the model used in the specification and the mental models used by the application experts.

The new control structure model, with its associated analysis tools, will be inserted between the requirements specification and validation process and the start of the architectural design process. The type of control model being proposed is that used in STPA and CAST analyses. While we have used this model and associated analysis tools for system safety and security, the model is equally appropriate for the analysis of most (emergent) system engineering properties and the generation of general system architectures that have carefully considered tradeoffs among other emergent properties.

In this paper, the control model used by STPA and CAST is assumed, but when using it in design, there may be augmentations to the basic model that provide advantages and more features. That will be left for future research. Instead, this paper focuses on the process. New structures created to improve that process can be considered later.

Here again is the basic process to be followed when creating a conceptual architecture during concept development:

1. Establish system goals
2. Create CONOPS
3. Identify high-level requirements and system hazards
4. Define basic control structure
5. Derive high-level safety constraints using STPA and the control structure
6. Assess risk using STPA scenarios and the risk matrix (optional)
7. Create the initial conceptual architecture (from the basic control structure) and refine it using the STPA results.
8. Create physical/logical architecture from the conceptual architecture
9. Create a detailed systems design using additional STPA analysis in decision making
10. etc.

In the new, enhanced system engineering process, the normal STPA and concept development process is assumed. The system goals and high-level requirements are generated by the usual requirements engineering process and the potential losses and hazards to be considered are agreed

[9] Nancy Leveson, Intent Specifications: An approach to building human-centered specifications, IEEE Transactions on Software Engineering, vol. 26, no. 1, January 2000, pp. 15-35.In

upon by the stakeholders. The hazards are represented by constraints on how the goals can be accomplished. For example, in the design of TCAS, an aircraft collision avoidance system, one system-level safety-critical constraint was that TCAS not interfere with the operation of the existing Air Traffic Control system. Notice that this constraint cannot be stated using standard "shall" statement (which usually forbids including a "not").

Then a system-level hierarchical control structure is generated, which will serve as the initial version of the conceptual functional architecture. Scenarios and constraints for safety and security are generated by first performing STPA on the control structure and recommendations are generated for the design. As stated earlier, STPA could also be used to generate requirements and recommendations for other emergent system properties. These requirements and design recommendations will be used to refine the first, usually very high-level hierarchical control structure into a more detailed conceptual control structure or architecture. Once a detailed conceptual design is completed, the logical/physical architecture can be generated from it.

Figure 10 shows a conceptual control model for a generic control system architecture. This model is augmented from our past models to include more facets of such an architecture. The conceptual architecture for a specific system will particularize this generic model into a conceptual model (abstraction) to describe the solution for a specific problem. The generic components in Figure 10 will correspond to the parts of your model. A brief description is included here of each of the components of the generic model along with descriptions of some of the analyses that might be performed. More details can be found in the Appendix of this paper.

Controlled Process

A hazard is defined in terms of the state of the controlled process at the bottom of Figure 10, e.g., the attitude of the aircraft, the speed of the automobile, the position of the robot. States are composed of components or variables. As the goal of STPA is to identify how the controlled process could get into a hazardous state, then we need to look at the ways the controlled process can change state. Anything that can change that state may potentially be part of a causal scenario for a hazard.

The conceptual architecture must include the process inputs and outputs, potential external disturbances, and all the controllers of the process. The analysis of the conceptual architecture will include these components of the architecture as well as failures or degradation over time of the controlled process components.

Automated Controller(s)

Only one automated controller is shown in the diagram (the blue box) although there may be several of these and they may be designed with more hierarchical levels. Because the model for automated and human controllers always have the same components, only one is shown here.

Except in very simple systems, automated controllers provide control actions to actuators, which change the controlled process state. Sensors provide feedback about the current process state.

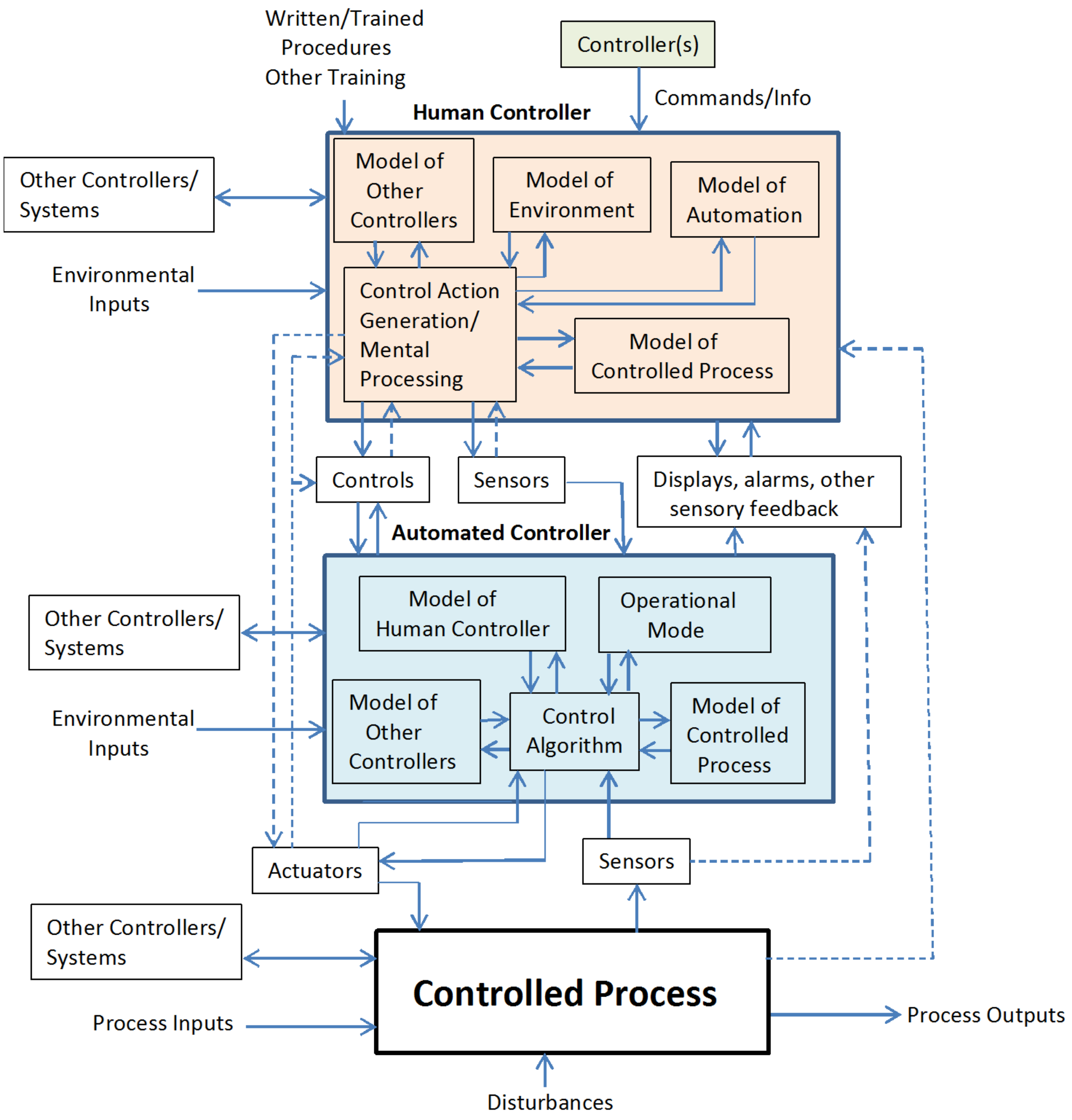
The conceptual architecture must contain the control path from the automated controller to the controlled process through the actuator and a feedback path from the controlled process to the automated controller through sensors.

Within the automated controller, the conceptual architecture contains the following components (all described in more depth in the Appendix):

- The automated control algorithm,
- A model of the controlled process,
- A model of the operational modes (the controlled process mode, the automation mode, the supervisory mode, and the display mode),
- A model of the human controller (if relevant), and

- A model of other controllers.

The conceptual architecture model must also contain the environmental inputs, other controllers or systems, transmission of information between the automation and its supervisor(s), and direct changes to the automated controller that do not go through the control algorithm.



Written/Trained Procedures Other Training
Controller(s)
Commands/Info
Human Controller
Other Controllers/ Systems
Model of Other Controllers
Model of Environment
Model of Automation
Environmental Inputs
Control Action Generation/ Mental Processing
Model of Controlled Process
Controls
Sensors
Displays, alarms, other sensory feedback
Automated Controller
Other Controllers/ Systems
Model of Human Controller
Operational Mode
Environmental Inputs
Model of Other Controllers
Control Algorithm
Model of Controlled Process
Actuators
Sensors
Other Controllers/ Systems
Controlled Process
Process Inputs
Process Outputs
Disturbances

Figure 10. A generic conceptual architecture.

<u>Human Controller(s)</u>

Human controllers are the most complex component of the model in Figure 10 (although system designers are quickly increasing the complexity of automated controllers) and thus provide some of the most important parts of the causal scenario and conceptual architecture generation processes. France has defined an extended model of human controllers.[10] All the same components are in this model, but they are connected differently than France did. An important role for this part of the conceptual architecture development is to allow interaction and collaboration between human factors experts and hardware/software engineers. While the human controller model is obviously not "implemented" in the physical/logical system architecture, the model provides important information for the concrete architecture. It also serves as a conceptual architecture for Human–Machine Interaction and augments communication between the hardware/software engineers and the human factors experts so that integrated system design, including operators and physical system designers, can be achieved. Implications of identifying human controller hazardous scenarios will also impact the physical human-machine interface design and operator training.

There are many relevant components of the model that should be included in the conceptual architecture, including control action generation and mental processing and the mental models related to these functions.

Many types of design problems can be identified using this part of the conceptual architecture including:

- Mode confusion is a common cause of accidents in mode-rich systems where the human is confused about the mode of the automation or the automation may be confused about the current mode of the controlled process. Mode confusion may be caused by incorrect updating in process models, inputs may be incorrect or delayed, updating may be delayed or the human controller may be informed of the mode change but does not notice or process that information. When automation can change the mode of the controlled process without being directed to do so by the human controller, mode confusion and potential unsafe control actions can result.
- "Situation awareness" is commonly cited as the cause of accidents without a careful definition of what that term means. A large number of errors can fall into this category. Note that the modeling used in STPA can identify situation awareness errors as they simply mean that the human controller mental models do not match the real state. The conceptual architecture should be designed to minimize such errors.
- Humans can easily be confused by automation that is nondeterministic or acts one way most of the time but in a few special cases behaves differently. Such automation also greatly increases training difficulties. Careful design of the conceptual architecture should be able to reduce such confusion as well as the other types of errors described here.
- Some automation is programmed such that the logic can result in side effects that are not intended or known by human controllers. If such side effects cannot be eliminated, then they need to be part of pilot training if they could lead to hazardous behavior.
- While the design of the system may include feedback to humans, there are many reasons why that feedback may not be noticed such as distraction or lowered alertness when monitoring something that rarely changes or is rarely incorrect.
- Complacency and overreliance on automation by humans is increasingly becoming a problem in automated systems today.
- Automation may fail so gracefully that the human controller does not notice the problem until late in the process. Humans may also think that automation is shut down or has failed when it has not. This type of problem has arisen when robots and humans must work in the same areas.

[10] Megan France, Engineering for Humans: A New Extension to STPA, M.S. thesis, Aeronautics and Astronautics Department, MIT, June 2017.

The logout/tagout problem, where humans think energy is off but is actually on, leads to a large number of accidents in the workplace.

This list is only meant to indicate that there are many causes that must be considered when humans are part of a system. It is far from exhaustive. Eliminating or controlling such factors are often assigned to human factors experts but they cannot be eliminated in the human–automation interface alone. Solutions must be reflected in the hardware and software design as well. The conceptual architecture provides a means for such collateral design. STPA operates on the human, hardware, and software parts of a system as well as managerial and social aspects. I am assuming here that managerial and social components of the system will be modeled in the same way as the direct controllers of the automation, but further investigation may determine that the conceptual architecture format shown in Figure 10 has to be extended.

To summarize, the new conceptual architecture design process starts with the identification of the hazards related to the controlled process. After the system-level hazards are identified (and perhaps refined, see the STPA Handbook), an initial high-level control structure is instantiated for the system being created. STPA is performed and the results are used to refine the conceptual architecture iteratively until a detailed conceptual architecture is created. At that point, the requirements on the physical/logical design should be complete and the standard design/architecture process can begin by generating detailed design features for the conceptual architecture.

While the emphasis in this paper is on the technical and human-operator parts of control systems, the same process could be used for management and social structures and other aspects of critical systems such as certification and operational controls.

Examples

Examples are necessary to understand the conceptual architecture process because, in past papers, we have not carefully distinguished the analysis of the detailed control structure from the original high-level analysis and did not show the process of refining the control structure. That is, we usually stopped after generating causal scenarios and often jumped around in the level of abstraction of the control structure being used. This paper tries to detangle and clarify all this. Essentially, the conceptual control structure is started at a very high-level of abstraction and then the STPA results are used to refine it into a more detailed conceptual architecture to the point where it is ready to be used for creating a physical/logical architecture and system design. Hopefully, if careful analysis has been done early in the system development, later changes will require backtracking only to the detailed design. Basic changes to the system requirements and goals, however, may require going back to the conceptual architecture. This should be easier than directly trying to change the physical/logical architecture as there is usually not the tracing from requirements and constraints to the physical/logical architecture that exists when using STPA and a conceptual architecture.

Three examples are provided here. The first is from Engineering a Safer World (Chapter 9), and was used there to illustrate safety-guided design. The example is changed slightly here to show the process for creating a conceptual architecture before the usual logical or physical architecture is created.

The second example is from the M.S. thesis of David Horney.[11] For simplicity, the first example stresses only one goal for the design process, i.e., safety. Design, however, always has multiple goals and

[11] David Craig Horney, *Systems-Theoretic Process Analysis and Safety-Guided Design of Military Systems*, Aeronautics and Astronautics Dept., MIT, 2017. This thesis can be obtained from the MIT Library or from http://sunnyday.mit.edu/STAMP-publications-sorted.pdf. The latter is a list (with hyperlinks) of recent STAMP-related publications including many papers and theses on STAMP and showing examples of STPA and CAST in a variety of application areas.

constraints and these must be subject to tradeoff analyses during the design and development process. Horney discussed this tradeoff process using a class of military helicopters that was in the early concept development stage to demonstrate the use of STPA to make complex tradeoff decisions among multiple possible conceptual architectures and system design features. Tethering of UAVs is used in this example.

The final example involves a system composed of manned and unmanned aircraft and illustrates the different results obtained by starting with an STPA-generated conceptual architecture compared to starting with generic architectures that are not tailored for or designed with respect to the specific system requirements and constraints.

Example 1: Generating a Conceptual Architecture for a Manufacturing Robot

This example involves an experimental robotic system, called the Tesselator, designed to service the Space Shuttle thermal tiles between flights.[12] The Tesselator was designed at Carnegie Mellon but was never used, as we understand, because of NASA safety and other concerns. Our Thermal Tile Processing System (TTPS) was based on the CMU Tesselator robot. Some students and I redid the design to demonstrate more sophisticated hazard analysis and Intent Specifications.[13]

The goal of the TTPS system was to inspect and waterproof the thermal protection tiles on the belly of the Space Shuttle, thus saving humans from a laborious task, typically lasting three to four months, that began within minutes after the Shuttle landed and ended just prior to launch. Upon landing at either the Dryden facility in California or Kennedy Space Center in Florida, the orbiter was brought to either the Mate-Demate Device (MDD) or the Orbiter Processing Facility (OPF). These large structures provided access to all areas of the orbiters.

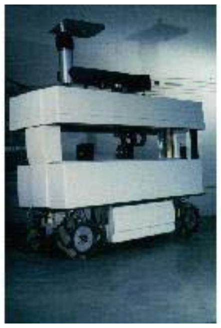


Figure 11 The original Tesselator robot

The Space Shuttle was covered with several types of heat-resistant tiles that protected the orbiter's aluminum skin during the heat of reentry. While the majority of the upper surfaces were covered with flexible insulation blankets, the lower surfaces were covered with silica tiles. These tiles had a glazed coating over soft and highly porous silica fibers. The tiles were 95 percent air by volume, which made them extremely light but also made them capable of absorbing a tremendous amount of water. Water in the tiles caused a substantial weight problem that could adversely affect launch and orbit capabilities for the shuttles. Because the orbiters could be exposed to rain during transport and on the launch pad, the tiles had to be waterproofed. This task was accomplished through the use of a specialized hydrophobic

[12] K. Dowling, R. Bennett, M. Blackwell, T. Graham, S. Gatrall, R. O'Toole, and H. Schempf. A mobile robot system for ground servicing operations on the Space Shuttle, *Cooperative Intelligent Robots in Space*, SPIE, November 1992.

[13] Israel Navarro, Kristina Lundqvist, and Nancy Leveson. An Intent Specification Model for a Robotic Software Control System, *20th DASC (Digital Avionics Systems Conference)*, 2001. The paper can be downloaded from: http://sunnyday.mit.edu/papers/dasc-maps.pdf

chemical, DMES, which was injected into each tile. There were approximately 17,000 lower surface tiles covering an area that was roughly 25m × 40m.

In the standard process, DMES was injected into a small hole in each tile by a handheld tool that pumped a small quantity of chemical into the nozzle. The nozzle was held against the tile and the chemical was forced through the tile by a pressurized nitrogen purge for several seconds. It took about 240 hours to waterproof the tiles on an orbiter. Because the chemical is toxic, human workers had to wear heavy suits and respirators while injecting the chemical and, at the same time, maneuver in a crowded work area. One goal for using a robot to perform this task was to eliminate a very tedious, uncomfortable, and potentially hazardous human activity.

The tiles also had to be inspected. A goal for the CMU Tesselator and later for our TTPS was to inspect the tiles more accurately than the human eye and therefore reduce the need for multiple inspections. During launch, reentry, and transport, a number of defects could occur on the tiles in the form of scratches, cracks, gouges, discoloring, and erosion of surfaces. The examination of the tiles determined if they needed to be replaced or repaired. The typical procedures involved visual inspection of each tile to see if there was any damage and then assessment and categorization of the defects according to detailed check-lists. Later, work orders were issued for repair of individual tiles.

Like any design process, creation of a conceptual architecture starts with identifying the goals for the system and the constraints under which the system must operate. The high-level goals for the TTPS were to:

1. Inspect the thermal tiles for damage caused during launch, reentry, and transport
2. Apply waterproofing chemicals to the thermal tiles

Environmental constraints (ECs) delimited how these goals could be achieved and identifying those constraints, particularly the safety constraints, is an early goal in conceptual architecture design. The environmental constraints in this case stemmed from physical properties of the Orbital Processing Facility (OPF) at Kennedy Space Center, such as size constraints on the physical system components and the necessity of any mobile robotic components to deal with crowded work areas and for humans to be in the area. Example work area environmental constraints for the TTPS are:

EC1: The work areas of the Orbiter Processing Facility (OPF) could be very crowded. The facilities provides access to all areas of the orbiters through the use of intricate platforms that were laced with plumbing, wiring, corridors, lifting devices, and so on. After entering the facility, the orbiters were jacked up and leveled. Substantial structure then was moved into place so that it surrounded the orbiter on all sides and at all levels. With the exception of the jack stands that supported the orbiters, the floor space directly beneath the orbiter was initially clear but the surrounding area could be very crowded.

EC2: The mobile robot had to enter the facility through personnel access doors 1.1 meters (42″) wide. The layout within the OPF allowed a length of 2.5 meters (100″) for the robot. There were some structural beams whose heights were as low as 1.75 meters (70″), but once under the orbiter the tile heights ranged from about 2.9 meters to 4 meters. The compact roll-in form of the mobile system had to maneuver these spaces and also raise its inspection and injection equipment up to heights of 4 meters to reach individual tiles while still meeting a 1-millimeter accuracy requirement.

EC3: Additional constraints involved moving around the crowded workspace. The robot had to negotiate jack stands, columns, work stands, cables, and hoses. In addition, there were hanging cords, clamps, and hoses. Because the robot might cause damage to the ground obstacles, cable covers were used for protection and the robot system had to traverse these covers.

Other system design constraints on the TTPS included:

DC1: Use of the TTPS must not negatively impact the flight schedules of the orbiters more than that of the manual system being replaced.

DC2: Maintenance costs of the TTPS must not exceed *x* dollars per year.

DC3: Use of the TTPS must not cause or contribute to an unacceptable loss (accident) as defined by Shuttle management.

The prioritized losses associated with the Shuttle and the Thermal Tile Processing System (TTPS) are defined to be (for this example):

A-1 *Level 1*

- **A1-1**: Loss of the orbiter and crew (for example, due to inadequate thermal protection).
- **A1-2**: Loss of life or serious injury in the processing facility.

A-2 *Level* 2

- **A2-1**: Damage to the orbiter or to objects in the processing facility that results in delay of a launch or in a loss greater than *x* dollars.
- **A2-2**: Injury to humans requiring hospitalization or medical attention and leading to long-term or permanent physical effects.

A-3 *Level 3*

- **A3-1**: Minor human injury (does not require medical attention or requires only minimal intervention and does not lead to long-term or permanent physical effects.
- **A3-2**: Damage to orbiter that does not delay launch and results in a loss of less than *x* dollars.
- **A3-3**: Damage to objects in the processing facility (both on the floor or suspended) that does not result in a delay of a launch or a loss of greater than *x* dollars.
- **A3-4**: Damage to the mobile robot.

  [**Assumption**: It is assumed that there is a backup plan in place for servicing the thermal tiles in case the tile processing robot has a mechanical failure and that the same backup measures can be used in the event the robot is out of commission due to other reasons.]

The hazards for this system can be defined as:

H1: Violation of minimum separation between the robot mobile base and objects (including the orbiter and humans) [**A1-2**, **A2-1, A3**].

H2: Unstable robot base [**A1-2**, **A2-1, A3**].

H3: Movement of the robot causing injury to humans or damage to the orbiter [**A1-2**, **A2-1, A3**].

H4: Conditions that could lead to damage to the robot [**A3-4**].

H5: Conditions that could lead to fire or explosion [**A1-2, A-2, A-3**].

H6: Contact of human with DMES waterproofing chemical [**A2-2, A3-1**].

H7: Inadequate orbiter thermal tile protection [**A1-1**].

These very high-level hazards will be refined using STPA. Then the STPA results will in turn be used to refine and analyze the conceptual architecture (design) alternatives.

After the hazards are identified, system-level safety-related requirements and design constraints are derived from them. As an example, for hazard H7 (inadequate thermal protection), a system-level safety design constraint is that the mobile robot processing must not result in any tiles being missed in the inspection or waterproofing process. More detailed design constraints will be generated during the conceptual architecture development process as more details are learned about the inspection and waterproofing processes.

To get started, an initial system conceptual architecture must be selected (figure 12). Let's assume that the initial TTPS architecture consists of a mobile base on which tools will be mounted, including a manipulator arm that performs the processing and contains the vision and waterproofing tools. This very early decision may be changed after the safety-guided design process starts, but some basic initial assumptions are necessary to get going: As the concept development and design process proceeds, information generated about hazards and design tradeoffs may lead to changes in the initial configuration. Alternatively, multiple design configurations may be considered in parallel.

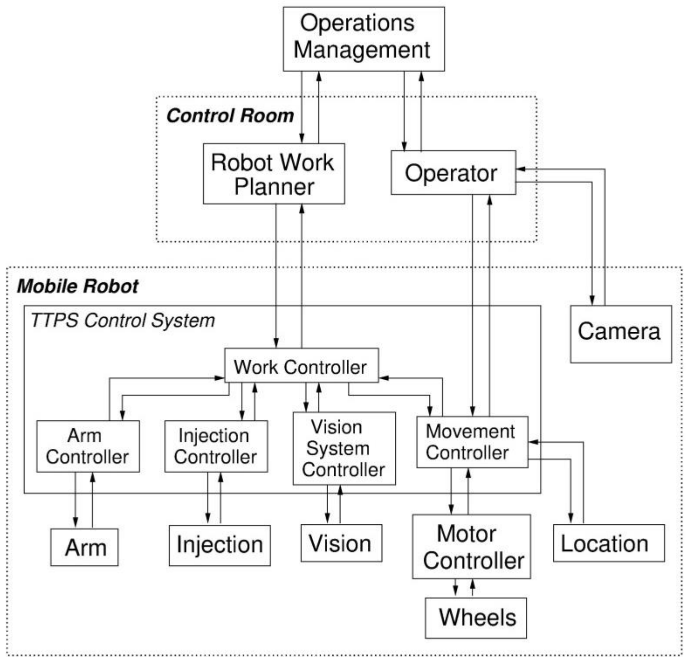


Figure 12. Initial Conceptual Architecture. To avoid clutter, the responsibilities for each component, the control commands, and feedback are omitted here but will need to be documented.

In the initial candidate conceptual architecture, a decision is made to introduce a human operator to supervise robot movement as so many of the hazards are related to movement. At the same time, it may be impractical for an operator to monitor all the activities so the first version of the system architecture assigns the TTPS control system the responsibility for non-movement activities and assumes both the TTPS and the control room operator will share control of movement. The conceptual architecture development process, using STPA, will identify the implications of this decision and will assist in analyzing the allocation of tasks to the various components to determine the safety tradeoffs involved. It is important that the rationale and assumptions underlying all these decisions are carefully documented. Again, an Intent Specification using a conceptual architecture at the system design level could be used to accomplish this goal.

There is also an automated robot work planner in the initial conceptual architecture that generates the overall processing goals and tasks for the TTPS. A location system is needed to provide information to the movement controller about the current location of the robot. A camera is used to provide information to the human controller, as the control room will be located at a distance from the orbiter. The role of the other components should be obvious.

The proposed architecture has two potential movement controllers, so this provides a goal for the conceptual designers to identify and eliminate potential coordination problems, which is a common cause of hazards. In this example, the operator could control all movement, but that may be considered impractical given the processing requirements and human factors issues, or the robot controller could be assigned this responsibility. To assist with this decision process, engineers may perform a human task analysis using the concept of operations (assumed to have been created before the architectural design process is started) and other information available about the proposed system.

The conceptual architecture development process, including STPA, will identify the implications of the basic decisions in the candidate tasks and will assist in analyzing the allocation of tasks to the various components to determine the safety tradeoffs involved.

The conceptual architecture analysis process is now ready to start. Using the information already specified, particularly the general functional responsibilities assigned to each component, designers will identify potentially hazardous control actions by each of the system components that could violate the safety constraints, determine the causal factors that could lead to these hazardous control actions, and prevent or control them in the conceptual architecture design. The process thus involves a top-down identification of scenarios in which the safety constraints could be violated. The scenarios can then be used to guide more detailed conceptual design decisions.

In general, the conceptual design process involves first attempting to eliminate the hazard from the design and, if that is not possible or requires unacceptable tradeoffs, reducing the potential for the hazard to occur, reducing the negative consequences of the hazard if it does occur, and implementing contingency plans for limiting damage.

Chapter 9 of *Engineering a Safer World* contains information about designing a control structure for safety, particularly when human controllers are included. Too often, detailed architectures and designs are created without adequate consideration of the role of human controllers in the system. The resulting design then has to rely on operator training, following written procedures, and not making natural human mistakes. Designing for human control from the beginning creates the potential for designing to reduce common types of human error such as mode confusion. The new conceptual architecture phase is a good time for a human-centered design process to commence.

As design decisions are made, STPA-based hazard analysis is used to inform these decisions. Early in the system design process, little information is available, so the hazard analysis will be very general. It will be refined and augmented as additional information emerges through the system design activities.

For the example, let's focus on the robot instability hazard. The first goal should be to eliminate the hazard in the physical (controlled) system design. One way to eliminate potential instability is to make the robot base so heavy that it cannot become unstable, no matter how the manipulator arm is positioned. A heavy base, however, could increase the damage caused by the base coming into contact with a human or object or make it difficult for workers to manually move the robot out of the way in an emergency situation. An alternative solution is to make the base long and wide so the moment created by the operation of the manipulator arm is compensated by the moments created by base supports that are far from the robot's center of mass. A long and wide base could remove the hazard but may violate the environmental constraints in the facility layout, such as the need to maneuver through doors and in the crowded OPF.

The environmental constraint **EA2** above implies a maximum length for the robot of 2.5 meters and a width no larger than 1.1 meter. Given the required maximum extension length of the manipulator arm and the estimated weight of the equipment that will need to be carried on the mobile base, a calculation might show that the length of the robot base is sufficient to prevent any longitudinal instability, but that the width of the base is not sufficient to prevent lateral instability.

If eliminating the hazard is determined to be impractical (as in this case) or not desirable for some reason, the alternative is to identify ways to control it. This new decision to try to control it may turn out not to be practical or later may seem less satisfactory than increasing the weight (the solution earlier discarded). All decisions should remain open as more information is obtained about alternatives and back-tracking is an option.

At the initial stages in design, we identified only the general hazards—for example, instability of the robot base and the related system design constraint that the mobile base must not be capable of falling over under worst-case operational conditions. As design decisions are proposed and analyzed, they will lead to additional refinements in the hazards and the design constraints.

For example, a potential solution to the stability problem is to use lateral stabilizer legs that are deployed when the manipulator arm is extended but must be retracted when the robot base moves. Let's assume that a decision is made to at least consider this solution. That potential design decision generates a new refined hazard from the high-level stability hazard (H2):

H2.1: The manipulator arm is extended while the stabilizer legs are not fully extended.

Damage to the mobile base or other equipment around the OPF is another potential hazard introduced by the addition of the legs if the mobile base moves while the stability legs are extended. Again, engineers would consider whether this hazard could be eliminated by appropriate design of the stability legs. If it cannot, then that is a second additional hazard that must be controlled in the design with a corresponding design constraint that the mobile base must not move with the stability legs extended.

There are now two new refined hazards that must be translated into design constraints:

1. The manipulator arm must never be extended if the stabilizer legs are not fully extended.
2. The mobile base must not move with the stability legs extended.

STPA can be used to further refine these constraints and to evaluate the resulting designs. In the process, the control structure will be refined and perhaps changed. In this case, a controller must be identified for the stabilizer legs, which were previously not in the design. As a starting point, let's assume that the legs are controlled by the TTPS movement controller (Figure 13). Alternatively, there could be a separate leg controller. However, leg extension and retraction are so intimately connected to movement in this design, we might decide to group these functions in a single controller.

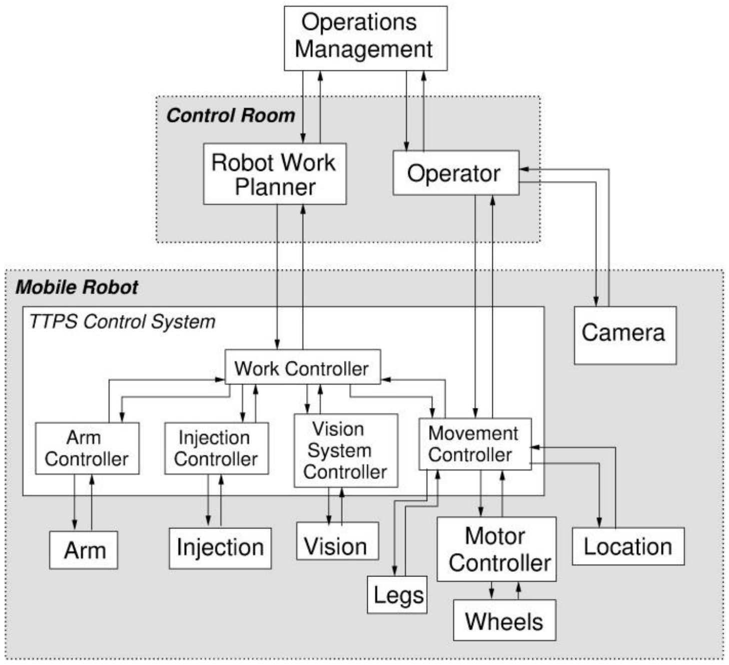


Figure 13. A refined control structure for the TTPS with stabilizer legs added.

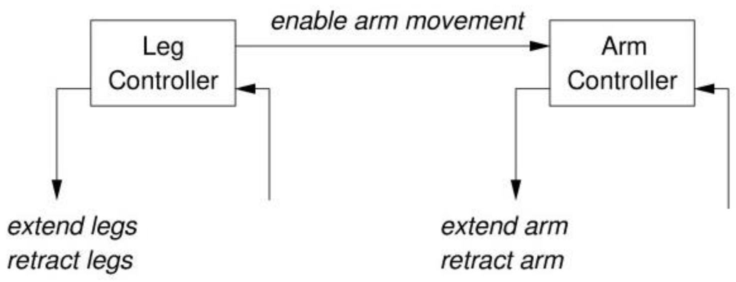
Using the augmented control structure, the remaining activities in STPA are to identify potentially hazardous control actions by each of the system components that could violate the safety constraints, determine the causal factors (scenarios) that could lead to these hazardous control actions, and prevent or control them in the conceptual architecture. The process thus involves a top-down identification of potentially hazardous scenarios so that they can be used to guide more detailed design decisions.

The refined hazards for the system components are at this point:

H1: Arm extended while legs retracted

H2: Legs extended during movement

The control actions for this part of the design are shown in Figure 14 along with the UCAs. Here a little more detail about the control of the leg, which may be part of the movement controller or controlled by the movement controller (a decision will be made based on the identified hazards and causal scenarios) is shown.



Control action	Not Provided	Provided	Early/late/wrong order	Stopped too soon
Extend legs	Legs not extended before arm extended **H1**	Extend legs during movement **H2**	Extend arm before legs extended **H1**	Stop before legs fully extended **H1**
Retract legs	Legs not retracted before movement **H2**	Retract legs while arm extended **H1**	Retract legs before arm fully stowed **H1**	Stop while legs still partially extended **H1**

Control action	Not Provided	Provided	Early/late/wrong order	Stopped too soon
Extend arm	(tile processing hazard)	Extend arm when legs retracted **H1**	Extend arm before legs fully extended **H1**	(tile processing hazard)
Retract arm	Not retracted before movement starts **H2**	(tile processing hazard)	(tile processing hazard)	Stop before arm fully stowed and movement starts or legs retracted **H1 H2**

Figure 14. Part of the STPA Analysis

Combining similar entries for H1 in the table leads to the following unsafe control actions by the leg controller with respect to the instability hazard:

1. The leg controller does not command a deployment of the stabilizer legs before the arm is extended.
2. The leg controller commands a retraction of the stabilizer legs before the manipulator arm is fully stowed.
3. The leg controller commands a retraction of the stabilizer legs after the arm has been extended or commands a retraction of the stabilizer legs before the manipulator arm is stowed.
4. The leg controller stops extension of the stabilizer legs before they are fully extended.

A potential unsafe control action by the arm controller is:

1. The arm controller extends the manipulator arm when the stabilizer legs are not extended or before they are fully extended.

The inadequate control actions can be restated as system safety constraints on the controller behavior (whether the controller is automated or human):

1. The leg controller must ensure the stabilizer legs are fully extended before arm movements are enabled.

2. The leg controller must not command a retraction of the stabilizer legs when the manipulator arm is not in a fully stowed position.

3. The leg controller must command a deployment of the stabilizer legs before arm movements are enabled; the leg controller must not command a retraction of the stabilizer legs before the manipulator arm is stowed.
4. The leg controller must not stop the leg extension until the legs are fully extended.

Similar constraints will be identified for all hazardous commands: for example, the arm controller must not extend the manipulator arm before the stabilizer legs are fully extended.

These system safety constraints might be enforced through physical interlocks, human procedures, and so on. Using STPA to identify the causal scenarios for unsafe control actions provides valuable information during conceptual and later detailed design (1) to evaluate and compare the different design choices, (2) to design the controllers and design fault tolerance features for the system, and (3) to guide the test and verification procedures and training for human controllers. As design decisions and safety constraints are identified, the functional specifications for the controllers can be created.

To produce detailed scenarios for the violation of safety constraints, the control structure is augmented with process models and feedback. The preliminary design of the process models comes from the information necessary to ensure the system safety constraints hold. For example, the constraint that the arm controller must not enable manipulator movement before the stabilizer legs are completely extended implies there must be some type of feedback to the arm controller to determine when the leg extension has been completed.

As more information is obtained from the hazard analysis and causal scenarios, the original conceptual architecture may be altered to minimize safety-critical fault tolerance and communication requirements. For example, at this point the need for the process models of the leg and arm controllers to be consistent and the communication required to achieve this goal may lead the designers to decide to combine the leg and arm controllers (Figure 15). This architectural decision is not one that might be considered by the designers without the information provided by STPA, but it turns out to save much effort (both development and verification) and reduce risk by eliminating many plausible and common causal scenarios for unsafe control.

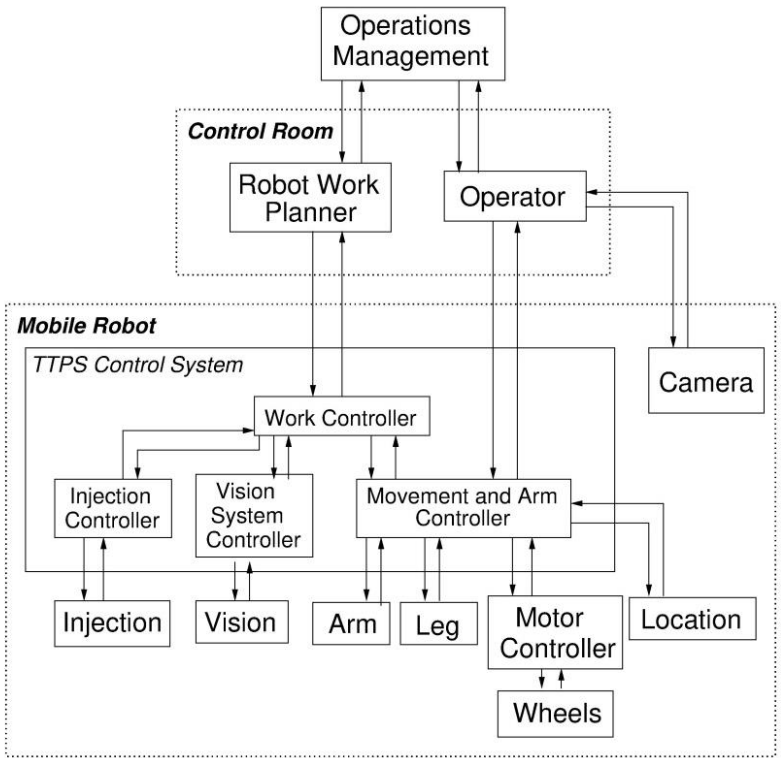


Figure 15. Combining the movement and arm control

Using the causal scenarios generated by STPA for the stability hazard, it is clear that feedback about the position of the legs is critical to ensure that the process model of the state of the stabilizer legs is consistent with the actual state. The movement and arm controller cannot assume the legs are extended simply because a command was issued to extend them. The command may not be executed or may only be executed partly. One possible scenario, for example, involves an external object preventing the complete extension of the stabilizer legs. In that case, the robot controller (either human or automated) may assume the stabilizer legs are extended because the extension motors have been powered up (a common type of design error). Subsequent movement of the manipulator arm would then violate the identified safety constraints. Just as the analysis assists in refining the component safety constraints (functional requirements), the causal analysis can be used to further refine those requirements and to design the control algorithm, the control loop components, and the feedback necessary to implement them.

Many of the causes of inadequate control actions are so common that they can be restated as general design principles for safety-critical control loops. The requirement for feedback about whether a command has been executed in the previous paragraph is one of these. Others result from having multiple controllers (as in this case) over the controlled system. More general design principles for safety can be found in Chapter 9 of *Engineering a Safer World* and Chapters 16 and 17 in *Safeware*.

To summarize, hazard analysis using STPA will identify application-specific safety design constraints that must be enforced by the control algorithm. For the thermal-tile processing robot, a safety constraint identified above is that the manipulator arm must never be extended if the stabilizer legs are not fully extended. Causal analysis can identify specific causes for the constraint to be violated and design features can be created to eliminate or control them in the conceptual architecture.

More general principles of safe control algorithm functional design can also be identified by using the general causes of accidents as defined in STAMP, general engineering principles, and common design flaws that have led to accidents in the past.

Accidents related to software or system logic design almost always result from incompleteness and unhandled cases in the functional design of the controller. This incompleteness can be considered a requirements problem. Some requirements completeness criteria were identified in *Safeware* and specified using a state machine model. Those criteria plus additional design criteria are translated into functional design principles for the components of the control loop, again in Chapter 9 of *Engineering a Safer World*.

Example 2: Generating an Architecture for Tethered UAVs

The second example demonstrates a more complex design and how the STPA results can be used in the conceptual architecture development process and to assist in design/architectural tradeoffs. The example comes from a master's thesis by David Horney in the MIT Aeronautics and Astronautics Dept.[14]

The example involves an Army hypothetical light military transport aircraft. It must be capable of carrying fourteen combat troops into battle in full gear. Each soldier will weigh approximately 350 pounds with their gear. The aircraft must also be capable of transporting a payload of 15,000 pounds over a range of 800 nautical miles without outside support. It must be able to deliver troops and cargo to remote bases and land on unimproved runways with short take-off and landing (STOL) capability. All fourteen troops must be able to unload with their gear in 60 seconds. Finally, the aircraft must be able to travel in a tethered formation. A single crew must be able to control three aircraft from takeoff to landing at improved airports with instrument landing system (ILS) capabilities.

The mishaps for this example are defined as:

M-1: Serious injury or fatality to personnel
M-2: Loss of or damage to the aircraft or equipment on the aircraft
M-3: Inability to complete the mission

These are the preliminary, most general mishaps focused on designing the aircraft to operate safely. Later in conceptual development, other mishaps related to operations and specific missions and capabilities can be added as the CONOPS is refined.

The high-level hazards are shown below.

Hazard	Constraint
H1: Violation of minimum separation standards (M1, 2, 3)	The aircraft must maintain minimum separation from potential sources of collision.
H2: Inability to control the aircraft (M1, 2, 3)	The aircraft must be controllable by the pilot or piloting function in an OPV (optionally piloted vehicle) at all times.

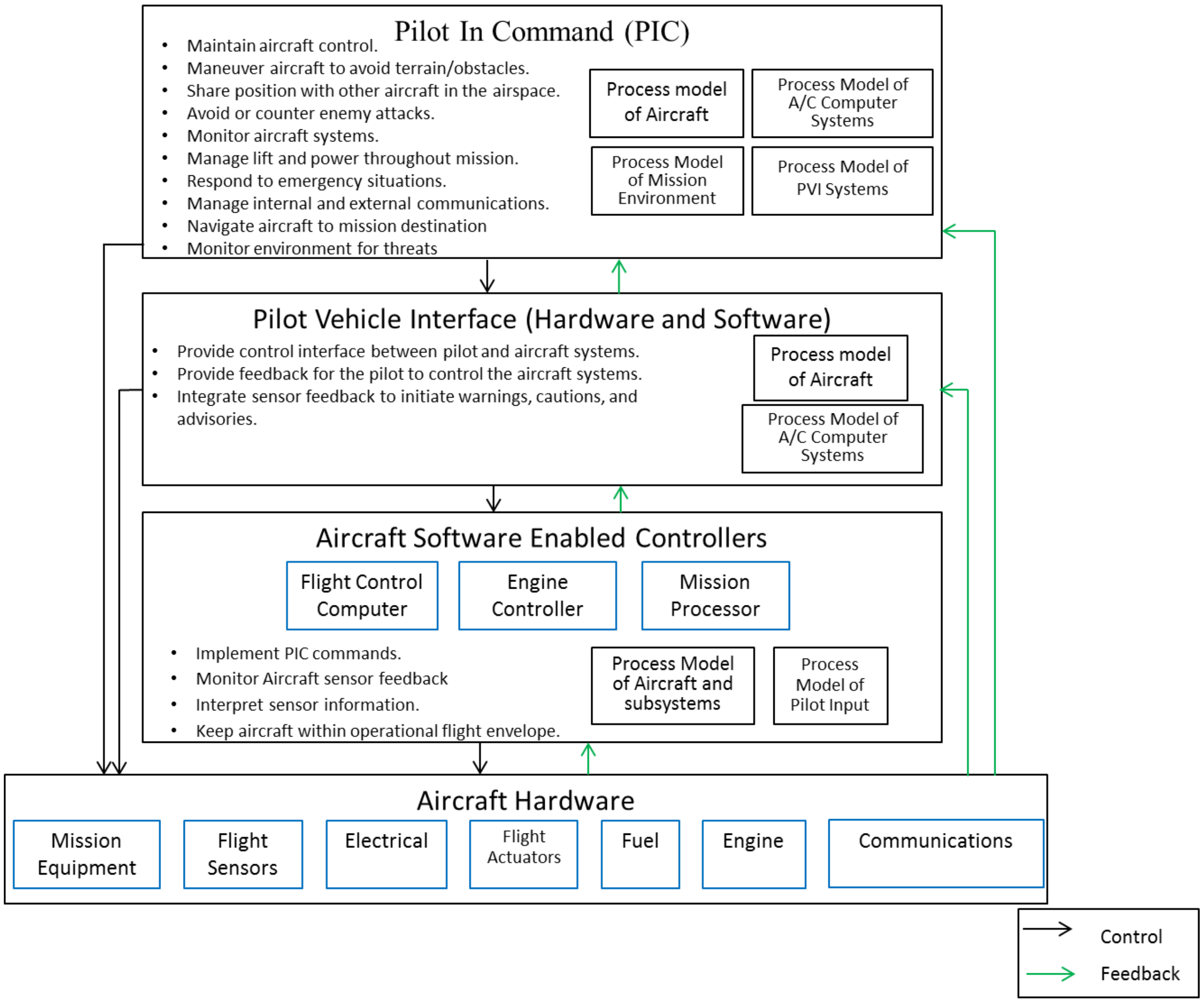
[14] David Craig Horney, *Systems-Theoretic Process Analysis and Safety-Guided Design of Military Systems*, Aeronautics and Astronautics Dept., MIT, 2017. This thesis can be obtained from the MIT Library or from http://sunnyday.mit.edu/horney-thesis.pdf . The latter is a list (with hyperlinks) of recent STAMP-related publications including many papers and theses on STAMP and it's tools and showing examples of STPA and CAST in a variety of application areas.

.

H3: Loss of airframe integrity (M1, 2, 3)	Airframe integrity must not be lost during flight.

Figure 16 shows the high-level conceptual architecture for the piloted aircraft. The management and higher-level control structure is omitted here, but including the higher management and social levels of the control structure in the system conceptual development process will allow including considerations that go beyond the aircraft itself, such as the chain of command.

From this high-level system architecture, more detailed models of different aircraft functions will be created (not shown here). The general constraints and design recommendations resulting from the STPA analysis are also not included here. Instead, an example is presented that shows how tradeoffs in architectural design might be analyzed and resolved. The example involves tethering capability. The example makes some assumptions: (1) the lead aircraft and its tethered counterparts will travel together in formation and (2) a single software-enabled controller is primarily responsible for executing the tethering mission in each aircraft. Changes in these assumptions will simply change the reasoning involved in the architectural decisions.



Pilot In Command (PIC)
Maintain aircraft control.
Maneuver aircraft to avoid terrain/obstacles.
Share position with other aircraft in the airspace.
Avoid or counter enemy attacks.
Monitor aircraft systems.
Manage lift and power throughout mission.
Respond to emergency situations.
Manage internal and external communications.
Navigate aircraft to mission destination
Monitor environment for threats
Process model of Aircraft
Process Model of A/C Computer Systems
Process Model of Mission Environment
Process Model of PVI Systems
Pilot Vehicle Interface (Hardware and Software)
Provide control interface between pilot and aircraft systems.
Provide feedback for the pilot to control the aircraft systems.
Integrate sensor feedback to initiate warnings, cautions, and advisories.
Process model of Aircraft
Process Model of A/C Computer Systems
Aircraft Software Enabled Controllers
Flight Control Computer
Engine Controller
Mission Processor
Implement PIC commands.
Monitor Aircraft sensor feedback
Interpret sensor information.
Keep aircraft within operational flight envelope.
Process Model of Aircraft and subsystems
Process Model of Pilot Input
Aircraft Hardware
Mission Equipment
Flight Sensors
Electrical
Flight Actuators
Fuel
Engine
Communications
Control
Feedback

Figure 16. High-level conceptual architecture for the piloted aircraft

Figure 17 shows the high-level architecture for the PIC lead aircraft and the software-controlled aircraft. There is no assumption, in this initial conceptual architecture, that the piloted aircraft will control the software-controlled aircraft. As the conceptual architecture is refined using the results of STPA, such high-level control assumptions may change.

Different formations of the aircraft will be used in different phases of flight and circumstances. Some formations make the group more defensible while others increase fuel efficiency. Transition through all the phases of flight also requires different formations throughout a mission. In this example, the controller that implements the command to set a formation shape will vary. Two candidate conceptual architectures are considered here. In architecture one, the human pilot in command (PIC) determines the formation shape from the lead aircraft while the tethered aircraft are responsible for implementing the command by maintaining their position in the specified formation. In the alternative architecture, the tethered aircraft will determine the optimal formation shape based on present conditions and the current phase of flight.

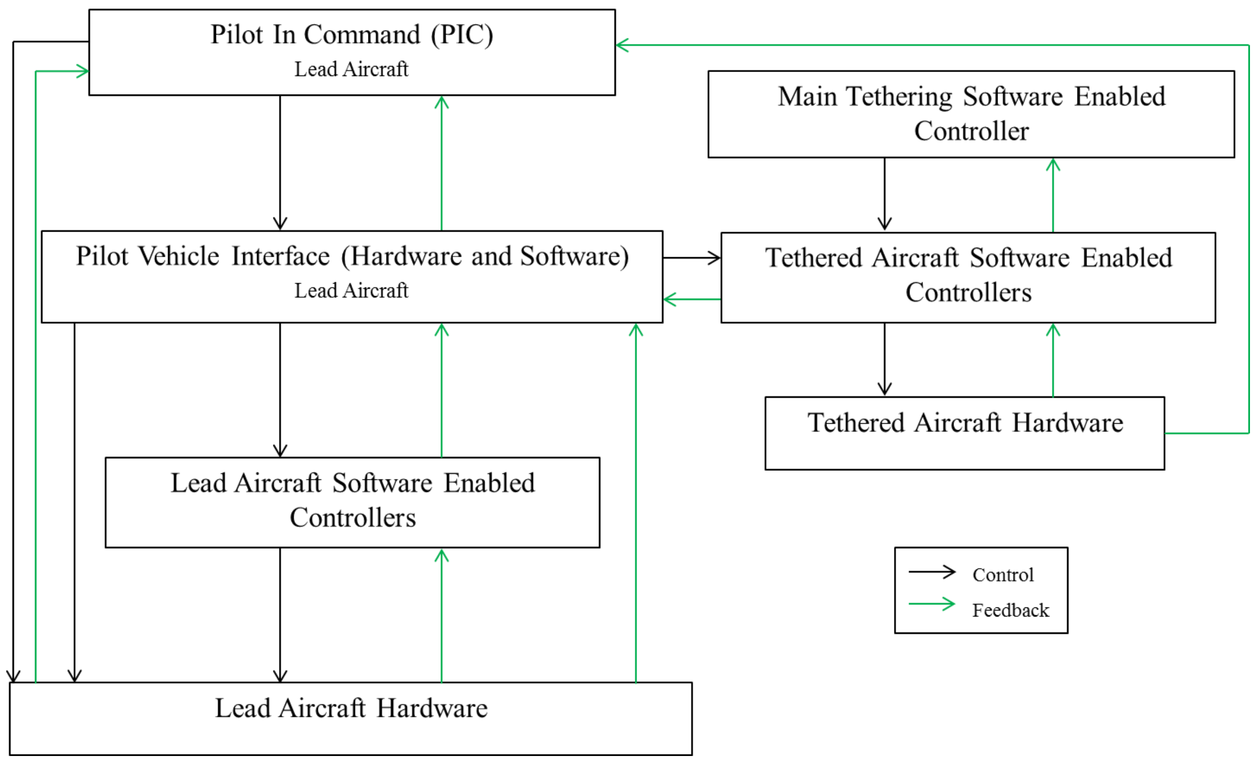

Pilot In Command (PIC)
Lead Aircraft
Main Tethering Software Enabled Controller
Pilot Vehicle Interface (Hardware and Software)
Lead Aircraft
Tethered Aircraft Software Enabled Controllers
Tethered Aircraft Hardware
Lead Aircraft Software Enabled Controllers
Control
Feedback
Lead Aircraft Hardware

Figure 17. Conceptual architecture for the piloted and tethered aircraft.

Without getting into details, the primary difference in these two alternatives is which controller is responsible for the control action "*Set Formation Shape*," the PIC in the Lead Aircraft or the tethered aircraft (the Main Tethering Software-Enabled Controller). This is, of course, a major architectural decision and will be difficult to change later in development. The conceptual architect would consider the unsafe control actions and causal scenarios for the two choices. While some of the causal scenarios are similar between the two potential architectures, others are unique. See David Horney's thesis for the detailed STPA analysis.

How should the decision about who to make responsible for the *Set Formation Shape* command be made? Horney notes that the architecture with the tethered aircraft setting the formation shape has two more unsafe control actions that can lead to the associated hazards. This result comes from the fact that responsibility resides in multiple controllers. Unless one of the tethered vehicles is specifically designated to choose the formation shape, the second architecture relies on the agreement of multiple entities to issue the control action. While this appears to provide some fault tolerance (redundancy), it also can lead to important control and safety problems, which are identified by STPA.

In addition, the lead PIC is still ultimately responsible for the safety of the formation and must be informed of the decision: The PIC must know how the formation plans to follow his lead in order to make appropriate piloting decisions. The presence of additional hazardous states does not necessarily mean that the second architecture is more dangerous than the first; it simply means that designers will have to consider additional factors during design to assure the safety of the system.

In some cases, an architecture can have a large number of hazards that can easily be mitigated. Other architectures with fewer hazards could still be more dangerous if the fewer paths to unsafe control cannot be easily prevented or the effects mitigated. In this case, delegating the responsibility to set the formation shape to the tethered vehicles could decrease the workload for the PIC and thus make the mission safer as long as the implementation is carefully thought through and the relevant hazards are addressed. Careful comparison and decision making will likely require considering the updating requirements for the process models of the system controllers, situational awareness, feedback requirements, security concerns, workload (particularly for the humans), software development requirements, etc.

The bottom line is that this complex decision making should occur in the conceptual architecture stage and not be performed at the same time as, or worse, after, the logical/physical architecture is being designed. This procedure allows a *separation of concerns*, i.e., deal with separate aspects of a difficult problem separately, where a structured decision process can lead to a better and more thoroughly considered design. Important decisions need to be disentangled from decisions about the implementation of those decisions, one of the basic principles of design presented earlier. Examples of important decisions needing to be made at the conceptual level are high-level system architectural tradeoffs, assignment of tasks to humans or to automation, where in the control structure decisions should be made and how to assign responsibilities to controllers, critical information that needs to be provided to each controller, mental/process models and preventing basic problems like mode confusion or general situational awareness problems, i.e., designing to assist humans in better decision making, and optimizing communication requirements.

Example 3: Manned-Unmanned Aircraft Teaming

The third example also involves teaming of manned and unmanned aircraft. The analysis was done by Jeremiah Robertson in his MIT Master's thesis.[15] I have reproduced here some of the modeling and analysis he produced. The previous examples were focused on showing how to create the conceptual architecture. The purpose of this example is to show the difference between starting with an STPA conceptual architecture versus an architecture that is not generated directly from specific system requirements.

Manned-Unmanned Teaming (MUM-T) describes a system where a controller (human, software, etc.) works together with unmanned systems. The primary motivation for such a platform is that missions can be completed by unmanned systems while the manned system provides guidance and oversight. The goal is to combine the strengths of manned and unmanned platforms to increase situational awareness in operations that include combat support and intelligence, surveillance, and reconnaissance (ISR) missions.

Several groups have provided generic architectures for MUM-T. Each define a decomposed set of functional components and their interactions. Differences exist between the functions and general data connections in these architectures. For example, one may use a central database to send and receive messages while others might use multiple servers that exchange temporary memory depending on a UVA's location. Each of the architectures was designed for specific and different payloads, mission sets, and communication capabilities. Therefore, trying to use one of these generic architectures for a particular MUM-T mission can result in greater costs and schedule delays than designing and building a MUM-T architecture from scratch. None was designed while considering safety and security requirements so verifying these properties may be very difficult. Here is where a conceptual MUM-T architecture can come into the picture.

It is useful to look at some of the MUM-T architectures (often called UAV Swarm architectures) that have been developed to see how they differ from a general MUM-T conceptual architecture.

[15] Jeremiah Robertson, Systems Theoretic Process Analysis Applied to Manned-Unmanned Teaming, Masters Thesis, Aeronautics and Astronautics Department, MIT, January 2019.

Johns Hopkins University Applied Physics Laboratory (JHU APL) developed an architecture for simple surveillance missions that allows the UAVs to take off and land, as well as select targets autonomously (Figure 18).

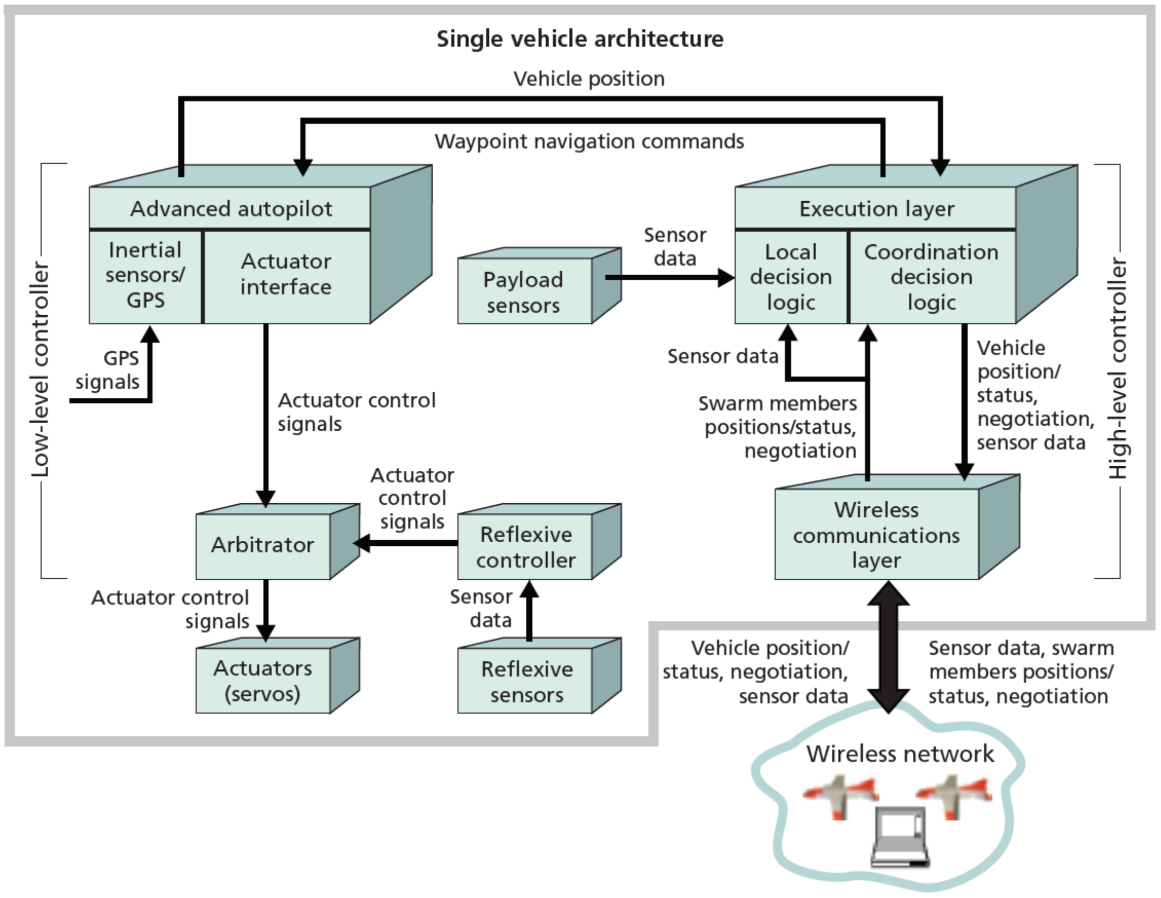


Figure 18. The JHU APL swarming modular architecture

MIT Lincoln Lab (MIT LL) developed software and data services for a UAV swarm with the goal of information sharing by UAVs for a variety of environments such as oceanic flight, international airspace, tactical operations, etc. (Figure 19).

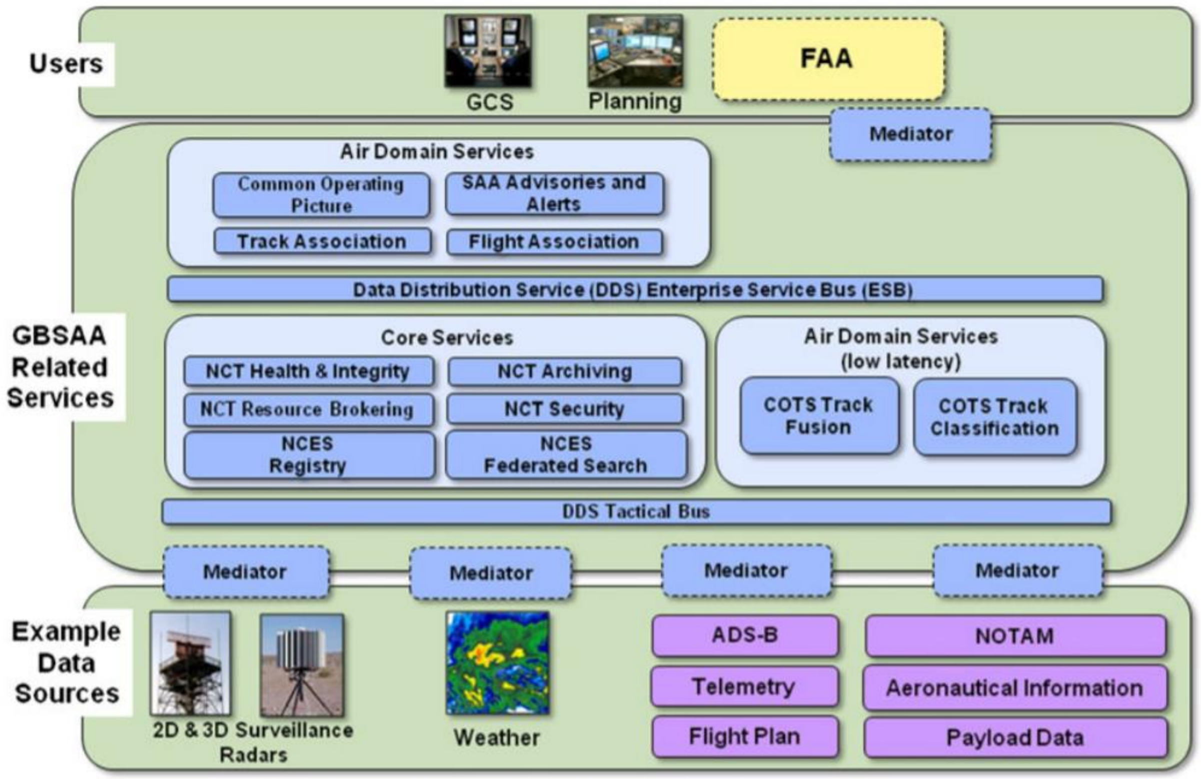


Figure 19. The MIT LL Reference Service-Oriented Architecture

A third example was produced by the Technical University of Catalonia in Barcelona, Spain. Their service-oriented architecture focuses on three primary services: flight services, mission/payload, and "awareness." The latter service monitors the surroundings and takes over the management of the flight when there are critical conditions, such as terrain and meteorological conditions. This architecture, shown in Figure 20, was instantiated for an example involving the mission of fighting wildfires.

Figure 21 shows the aerostack architecture created by a group headed by Molina at the University of Catalonia. A primary goal of this architecture was to generate a communication layer that is separate from the sensors, processing, etc. If the communication layer becomes corrupt, the other layers can still function independently.

As a final example, the Rand Corporation developed an architecture for UAVs to improve interoperability, provide greater autonomy to the UAVs, and allow for cooperative behaviors for teaming to perform complex missions in extreme environments. Rand analyzed the previously published architectures such as the ones developed by JHU APL and MIT LL in order to develop a set of essential modular components that would support multiple levels of autonomy.

None of these architectural designs started with a set of specific requirements. A set of assumptions about the swarm behavior was instead used, such as flying at varying altitudes, etc. According to Robertson, decisions were made on the basis of the designers' expertise and understanding of previous designs developed by other groups. In addition, each of these architectures applied to a specific mission scope and not all potential missions.

In contrast, Robertson applied STPA to create a generic MUM-T conceptual architecture to satisfy the requirements for the general design of a swarm architecture that is safe when coordinating with a manned aircraft.

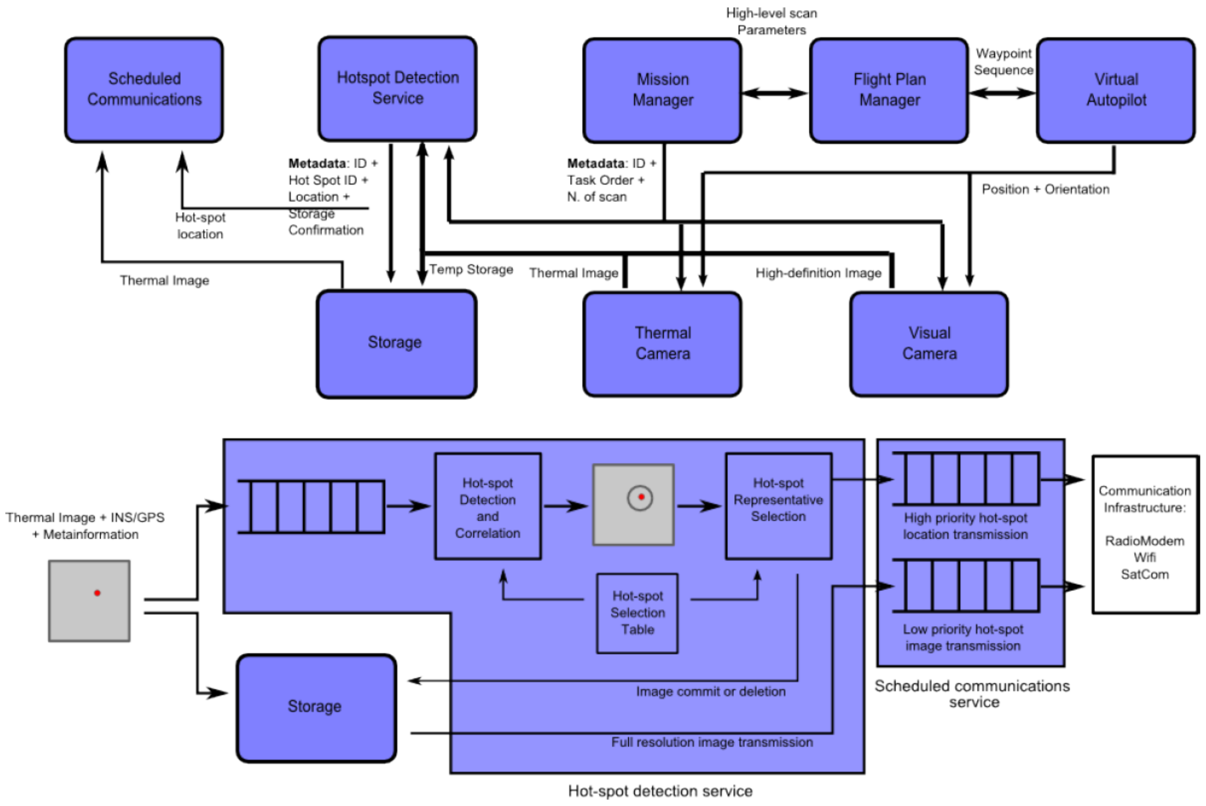

Scheduled Communications
Hotspot Detection Service
Mission Manager
Flight Plan Manager
Virtual Autopilot
High-level scan Parameters
Waypoint Sequence
Metadata: ID + Hot Spot ID + Location + Storage Confirmation
Metadata: ID + Task Order + N. of scan
Position + Orientation
Hot-spot location
Thermal Image
Temp Storage
High-definition Image
Storage
Thermal Camera
Visual Camera
Thermal Image + INS/GPS + Metainformation
Hot-spot Detection and Correlation
Hot-spot Representative Selection
Hot-spot Selection Table
High priority hot-spot location transmission
Low priority hot-spot image transmission
Communication Infrastructure: RadioModem Wifi SatCom
Scheduled communications service
Image commit or deletion
Full resolution image transmission
Hot-spot detection service

Figure 20. Example of University of Catalonia High-Level System Architecture for Wildfires

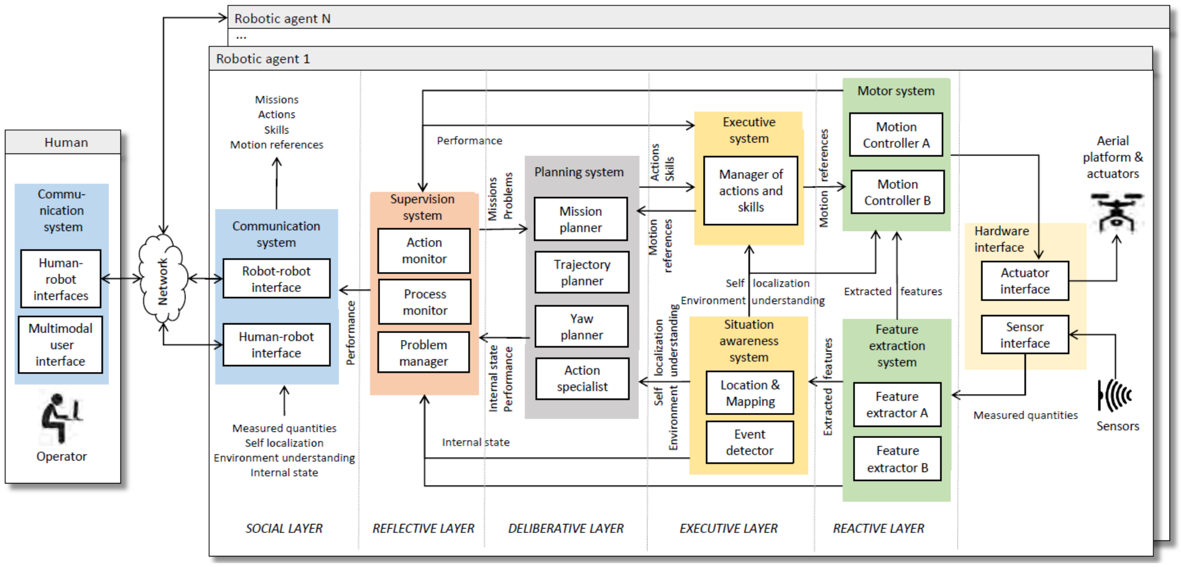


Figure 21. Main Components and Layers of the Aerostack Architecture

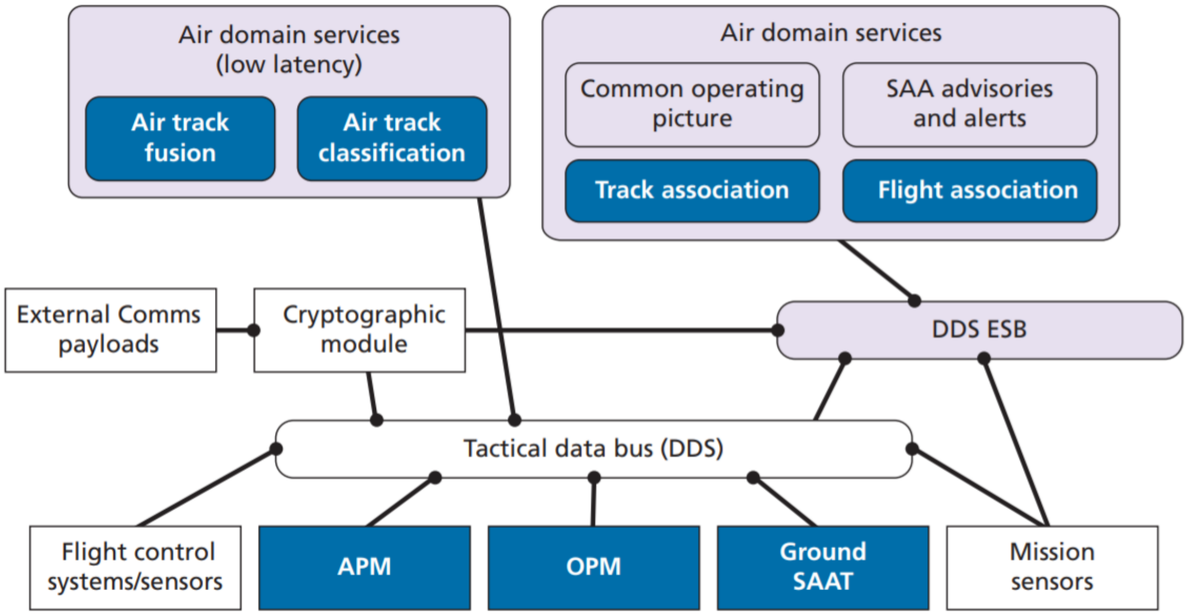


Figure 22. The RAND Corporation Recommended UAV Architecture

Robertson did not think of his analysis as producing a conceptual architecture but instead as producing general safety and security requirements for MUM-T. But the resulting model can be used in conceptual architecture development none the less. The very general requirements he generated can be refined into more specific requirements and constraints depending on the particular mission, etc. and used to create a conceptual architecture or perhaps multiple ones.

In contrast to the other generic swarm architectures described above, STPA is applied to MUM-T to identify safety and security requirements for an entire system including the software (automation), manned aircraft, ATC, a Ground Station, and more.

In the specific analysis performed by Robertson, a manned aircraft is used as the Team or Flight Lead. Few, if any, changes are needed to make the Team Lead an aircraft with the pilot on the ground instead of airborne, although this decision opens up the potential for more causal scenarios that would need to be considered in the conceptual architecture. The system includes a Crew Chief (CC), Air Traffic Control (ATC), Mission Planners (MP), an Autonomous Controller (AuC), and a Ground Station (GS). The Crew Chief acts as the head of tactical aircraft maintainers, coordinates the aircraft's care, and calls in specialists (like avionics or propulsion technicians) when a problem is found. Air Traffic Control provides coordination between aircraft to prevent air and ground collisions. The Mission Planners provide mission updates and prepare human pilots during the brief. Finally, the Ground Station provides tactical coordination for changing targets or locations and may also control unmanned systems when necessary using remote pilots.

The Ground Station is assumed to include remote pilots that can perform LOS (line of sight) or BLOS (beyond line of sight) control of the aircraft. There is no distinction between remote pilots at different locations because they are performing the same control actions. Finally, the Autonomous Controller actuates the UAV(s) hardware based on commands from the Team Lead or the Ground Station. For the STPA analysis at this point in development, the details of the autonomous controller are irrelevant. The location or decision-making capabilities do not change the early analysis. Whether it uses machine learning or not, whether it is on multiple aircraft or on one aircraft or on the ground does not make a difference. The basis for the early STPA analysis is that the autonomous controller is implementing commands from the Ground Station and Team Lead by actuating hardware on the UAV(s).

Because the analysis was done in response to a contract with AFRL, a military environment is assumed but need not be. Here are the basics (mishaps and hazards) that form the foundation of Robertson's analysis:

M1: Death or injury of a person (includes ground system personnel, Team Lead, civilians, friendly forces, etc.)
M2: Destruction or damage to aircraft
M3: Non-achievement of mission
M4: Ground property damage (either U.S. Air Force or civilian)

H1: Aircraft violates minimum separation from other aircraft or terrain [**M1, M2, M3, M4**]
H2: Aircraft loss of control (includes departure from stable flight) [**M1, M2, M3**]
H3: Aircraft does not execute planned operations [**M3**]
H4: Aircraft departs approved airspace (where approved airspace is defined by mission planning and ATC) [**M2, M3**]
H5: UAV fires at friendly forces [**M1, M2, M4**]
H6: Team Lead fires at friendly forces [**M1, M2, M4**]

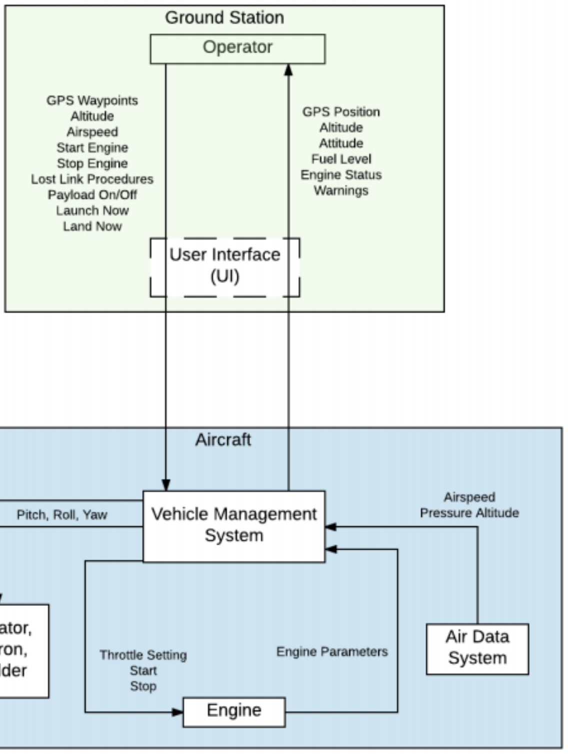
X	**X**	**X**	**X**	**H1**	**Aircraft violates minimum separation**
X	**X**	**X**		**H2**	**Aircraft Instability**
		X		**H3**	**Aircraft doesn't complete mission**
	X	**X**		**H4**	**Aircraft departs approved airspace**
X	**X**		**X**	**H5**	**UAV fires at friendly forces or civilians**
X	**X**		**X**	**H6**	**Team Lead fires at friendly forces or civilians**
M1	**M2**	**M3**	**M4**		
Loss of life	**Loss of aircraft**	**Loss of mission**	**Ground property damage**		

SC-1.1: Aircraft must satisfy minimum separation requirements from other aircraft and objects [**H1**]
SC-1.2: If aircraft violates minimum separation, then the violation must be detected, and measures taken to prevent collision [**H1**]
SC-2: Aircraft airframe integrity must be maintained under worst-case conditions [**H2**]
SC-3: Aircraft must satisfy mission parameters [**H5**]
SC-4.1: Aircraft should not depart approved airspace [**H6**]
SC-4.2: If aircraft violates approved airspace constraint, then the violation must be detected and measures taken to prevent encounters with enemy or law enforcement [**H6**]
SC-5.1: UAV must not fire at friendly assets or forces [**H3**]
SC-5.2: If a UAV violates the friendly firing constraint, then the violation must be detected and measures taken to prevent impact [**H3**]
SC-6: The Team Lead must not fire at friendly assets or forces [**H4**]

Figure 23 shows the difference between the conceptual architecture developed for the ground station and aircraft communication in STPA and the physical/logical architecture developed for the data link between the two, to again emphasize the difference between a conceptual architecture and a logical/physical one. The standard architectures start from identifying functions that are needed and then a connection logic. In contrast, the conceptual architecture starts from the system control requirements and identifies the requirements and constraints necessary for system safety, security, and other system properties that will need to be maintained by the physical/logical architecture. As in the TTPS example, the conceptual architecture can be optimized for these properties before a physical design is created.

Figure 24 shows the entire control structure included in Robertson's STPA analysis. One of the first things that can be seen without any analysis is that there are multiple controllers (e.g., the Team Lead and Ground Station) of the autonomous controller. Potential conflicting and unsafe control actions must be prevented in the conceptual architecture. Whereas the standard architecture development starts with identifying the functions that are to be included in the architecture, STPA starts from a model of the system as a whole that the architecture is being created to support and implement.

Robertson identifies what requirements are necessary for air-to-air combat or ISR missions involving manned and unmanned aircraft. Physical/logical architectures should be designed to implement those requirements and constraints in a top-down system engineering process. If the STPA analysis starts at a very high level, few changes may be needed for different systems, i.e., the conceptual architecture can be reused. If a different system is envisioned and under development, the entire analysis will not need to be redone.



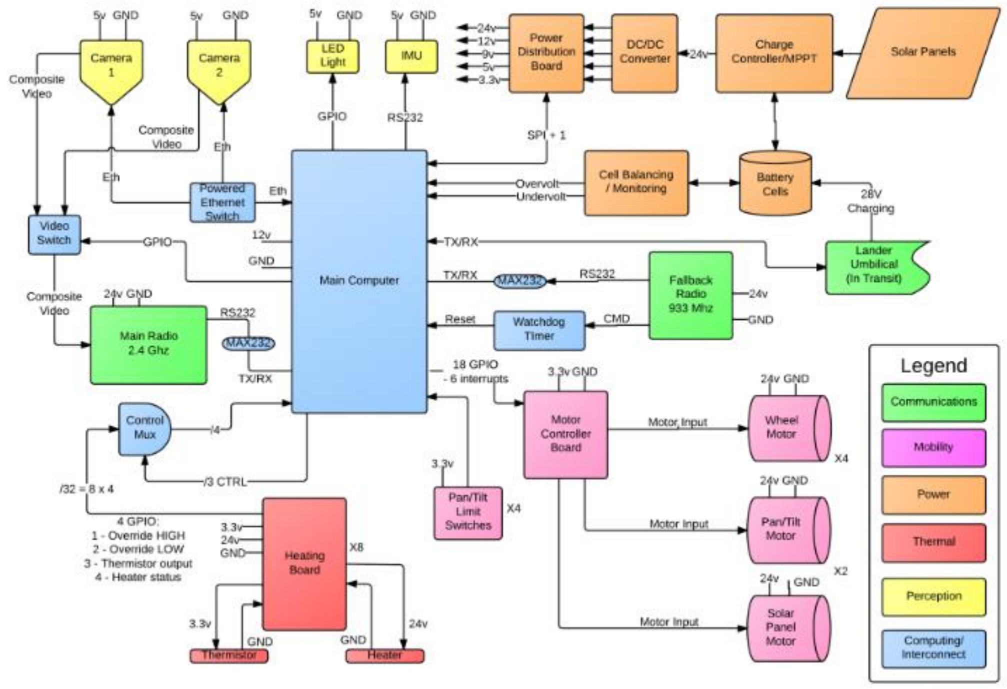


Figure 23. The top structure shows the conceptual architecture for the interaction between the ground station and the airport. The bottom shows the physical/logical architecture for the data link.

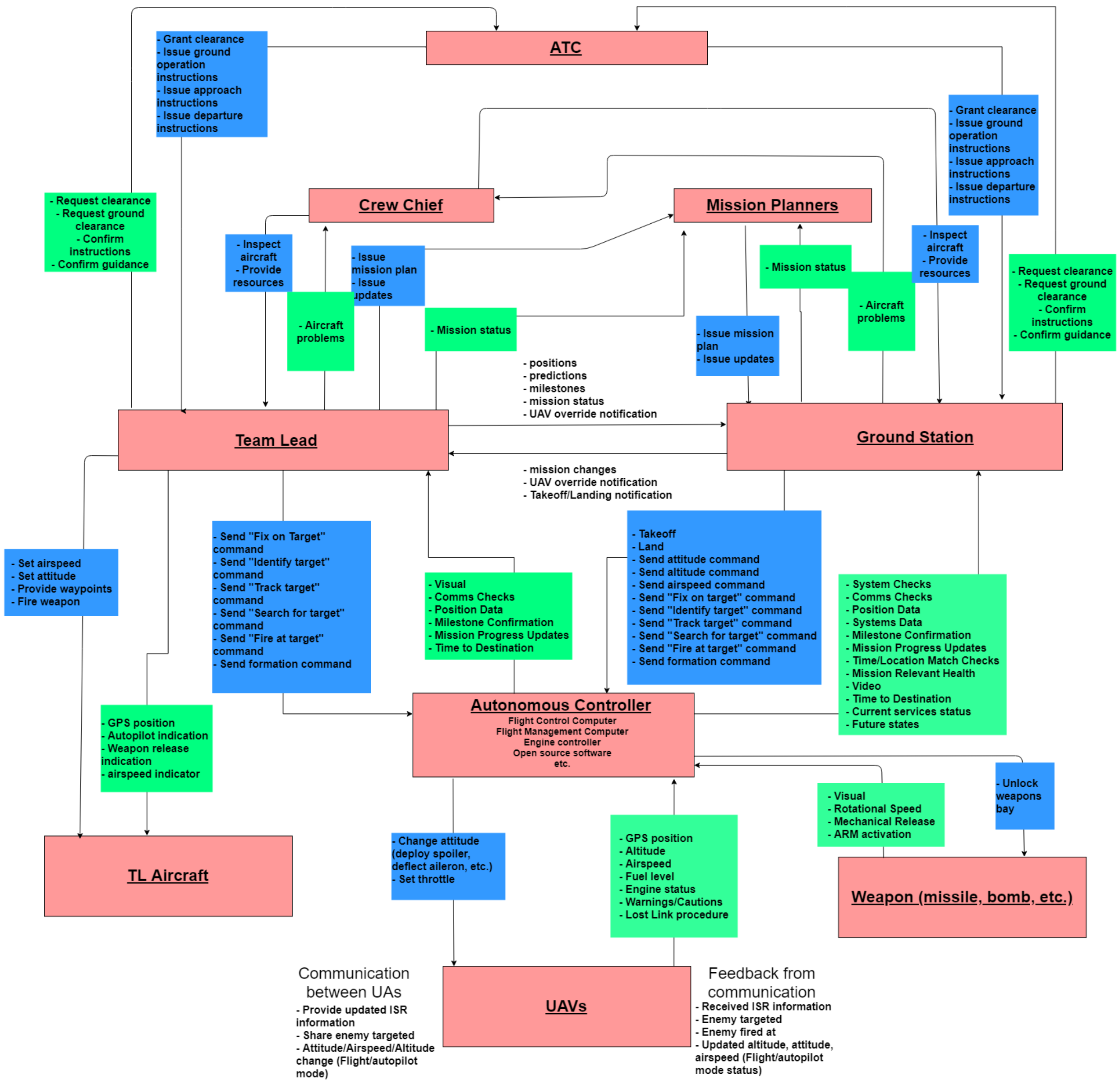


Figure 24. The high-level control structure (conceptual architecture) for MUM-T

An example UCA table for the landing command for the UAV where the Ground Station is assigned control over landing:

UAV Control Action	Not providing	Providing	Too early/Late	Applied too long/stopped too short
Land	Ground Station does not provide land command when the UAV is in a pattern and at minimum fuel (H1, H2)	Ground Station provides land command when the runway is not clear (H1)	Ground Station provides land command before the UAV completes the airfield arrival procedure (H1)	N/A
	Ground Station does not provide land command when the UAV is at the airfield and other aircraft are trying to enter a pattern (H1)	Ground Station provides land command when the UAV is above unstable terrain (H1)	Ground Station provides land command after the UAV has already entered a restricted air space (H6)	
	Ground Station does not provide land command when the UAV is ready to land (H1, H2)	Ground Station provides land command when the UAV is in a restricted landing area (H6)	Ground Station provides land command after the UAV is already out of fuel or is no longer over a safe landing area (H1, H2, H6)	
			Ground Station provides land command before the UAV(s) complete the mission (H5)	

The UCAs lead to some example constraints on the Ground Station and Team Lead:

- The Ground Station must provide a land command when the UAV is in a pattern at minimum fuel
- The Ground Station must provide a land command before a UAV enters restricted airspace
- The Autonomous Controller must provide attitude correction if the UAV is off course

More specific requirements, constraints, and conceptual architecture decisions will result from the scenarios created for the UCAs. As an example:

UCA: The Ground Station provides a land command before the UAV(s) complete the mission

Possible requirements that follow from the scenarios for this UCA include:

- The Ground Station must be able to verify the origin of a communication
- The Ground Station must verify the end of mission with the Team Lead

- The UAV(s) must not implement a post-mission procedure until provided with authorization from the Team Lead or Ground Station
- The Team Lead must provide verification to the Ground Station that the UAV(s) completed the mission
- The Team Lead must provide redundant feedback to the Ground Station to verify mission completion

These requirements not only assist in the design of a conceptual architecture, they can be used to generate verification and test requirements for later stages of development. Tracing information is omitted here, but in a complete analysis, these requirements would be traced back to the scenarios, UCA's, and hazards they were created to address.

Some of the scenarios and requirements may not apply for a particular system. By starting with a very general conceptual architecture, there is potential for reuse of the conceptual architecture in other systems and for enhanced interoperability. In fact, a particular physical/logical architecture may only implement a subset of the general conceptual architecture. Traceability of the analysis results to architecture features can assist with such partitioning.

Generating the Physical/Logical Architecture from the Conceptual Architecture

A detailed process for generating the physical/logical architecture from the conceptual architecture is a future research topic. Some steps are obvious. The essential difference in a process that includes a conceptual architecture development step versus the usual current approach is that analysis is performed to identify the requirements and best format for the physical/logical architecture rather than just jumping to one and assuming it will be effective. Basically, more systems analysis is performed before architectural decisions are made.

There are many benefits from a process that structures the detailed architecture from an initial conceptual architecture. It will ease the mapping from the problem to the architecture to be used to solve that problem. Consideration of critical system requirements and constraints, such as safety and security, during development of the conceptual architecture can significantly reduce the number of later required changes if deficiencies in the way safety and security have been handled are discovered. The system should be easier to review (validate and verify), to update and maintain/evolve over time, and to certify. It may also be possible to design systems that are easier to operate and to train operators.

Starting from a generic conceptual architecture, we may discover physical/logical architectures that are very different than today but have advantages throughout the system lifecycle.

Potential Role in Certification

One potential use for a conceptual architecture is in certification. Systems (aircraft, autos, etc.) today are very complex. Certifiers usually cannot understand all of the design details and the rationale for them. To get around this difficulty, current certification methods are based on probabilistic analysis to ease the certification effort. But another approach is possible that uses a conceptual architecture.

Some of my grad students and I were involved in the certification of TCAS, a very complex system for that time.[16] TCAS was certified before the use of probabilistic analysis in aircraft system certification became the standard approach. Engineers at MITRE performed a hazard analysis, but the analysis did not involve probabilities because the intensive software and human involvement in the system precluded deriving such probabilities. Qualitative scenarios were generated.

[16] Nancy Leveson, Mats Heimdahl, Holly Hildreth, and Jon Reese. Requirements Specification for Process-Control Systems, *IEEE Transactions on Software Engineering*, Vol 20, No. 9. September 1994, pp. 684-707. Downloadable from: http://sunnyday.mit.edu/papers/tcas-tse.pdf

In addition to the hazard analysis, a black box specification was developed using the format of the black box level of intent specifications and the commercial tools developed to implement intent specifications called SpecTRM. The language used, called RSML, has evolved over time to be much simpler and more straightforward, but it was satisfactory for the purpose at the time. The most recent form of the specification language is called SpecTRM-RL. SpecTRM-RL is formal but also easily reviewable, even by non-engineers, with minimum training (less than an hour). In addition, it is executable.

The government, MITRE, and my students were responsible for ensuring that the black box behavioral specification was safe, that is, that the scenarios identified by the fault tree analysis were eliminated or mitigated. The companies simply had to implement the black box specification in their own physical TCAS devices and convince the FAA that they had done sufficient testing and verification to ensure that the devices would behave in the same way as the official government behavioral specification. This type of verification is familiar to companies. The government, on the other hand, did not have to delve into the details of the various implementations and physical TCAS boxes.

The use of a conceptual architecture might make returning to such a certification approach practical while eliminating the serious limitations of probabilistic analyses in systems containing humans and software.[17]

TCAS is a long-lived system with changes in both itself and its environment over time. A conceptual architecture might make it easier to identify the impacts of changes in the environment, such as ADS-B and Reduced Vertical Separation Minimum, on safety. The changes in the conceptual architecture can be identified and, along with the traceability provided by STPA and the specification of design rationale (such as using Intent Specifications), it should be possible to recertify the system under the new conditions relatively easily.

How a Conceptual Architecture Based on Control Satisfies the Basic Design Principles

Use of a conceptual architecture assists in applying the design principles described earlier to systems where control is the primary concern. Dealing with complexity effectively requires being able to use top-down system design principles rather than bottom-up approaches. A conceptual architecture fills a big gap between requirements specification (as "shall" statements) and detailed design of the logical/physical system architecture to implement those requirements. We cannot completely understand very complex problems until we start to solve them. Generating a conceptual architecture provides a way of creating this understanding at the beginning of the problem solution phase. That is, the process involves determining "what" before "how."

A conceptual architecture provides views of the desired system behavior and encourages thinking of the system as a whole and not just a combination of parts. As Curtis and his colleagues found, a critical factor in successful projects is the ability to understand how the system being developed fits into the larger encompassing system as a whole. Those responsible for implementing the system components can see how their part fits into the whole without getting overwhelmed with details and losing the big picture.

While the process starts with the big picture, the identification of the control components and their subsequent analysis provides a form of "locality of Information" and the ability to consider and analyze components separately without having to simultaneously perform detailed design of all the system components at once.

[17] Nancy Leveson, Chris Wilkinson, Cody Fleming, John Thomas, and Ian Tracy, *A Comparison of STPA and the ARP 4761 Safety Assessment Process*, MIT Technical Report, Jun 2014, downloadable from http://sunnyday.mit.edu/STAMP/ARP4761-Comparison-Report-final-2.pdf

Abstraction is clearly involved in the conceptual architecture process. Separation of concerns is used along with stepwise refinement of design as the problem is better understood throughout the system development process. Top-down ordering of decision making is used, with details left until the overall high-level design is completed. Concurrent activities, such as development of test procedures can be performed in concert with conceptual architecture development.

Including human controllers in the conceptual architecture allows for integrated human-cyber-physical design and the ability to reduce common human–automation interaction problems, such as mode confusion. In essence, it allows integrating the design of the hardware and software with the design of human–automation interaction. All the types of engineers on the project—mechanical, software, and human factors—can use a common model for improved communication and coordinated design. The conceptual architecture for the human controllers also can be used in the design of operator controls, displays, procedures, training, etc.

A conceptual architecture, using the right specification language, is potentially executable although analysis ability and reviewability by experts should not be sacrificed for executability. Maintainability and evolution should be easier when the modifiers (who may not have been involved in the original design process) can understand the role of each component in the whole and how the components interact and work together to achieve the system requirements and maintain constraints on behavior. Companies might create proprietary conceptual architectures that apply to an entire product line and therefore reduce (reuse) the effort required for generating new products.

Conclusions

In many ways, there is absolutely nothing new in this paper. The techniques are all things that can and have been done when analyzing complex systems using STPA. But an assumption has been made in most of our previous papers that the control structure is created as an artifact for the STPA analysis only and that the results of the hazard analysis are just a set of usually English-language recommendations to be passed to the system designers. The control structure is not used further. We conceived of STPA and the control structure model as an “add on” that is done as a sideline to the regular development process.

What is new here is a recognition that the control structure can serve as a conceptual architecture at the beginning of the process of creating a logical/physical architecture, adding additional system analysis to the general system engineering process. The STPA process then becomes a critical step in the overall system engineering process, not a side activity. The control structure, the constraints created from the UCAs, and the recommendations arising from the causal scenarios serve as the guiding requirements and constraints on the generation of the system’s detailed logical/physical architecture. Multiple views of the system architecture may be generated and used, such as a model of data flow through the system, a model of the control flow as the system operates, and a model of the physical components and their physical connections. But they all should derive from and be consistent with the system’s conceptual control architecture.

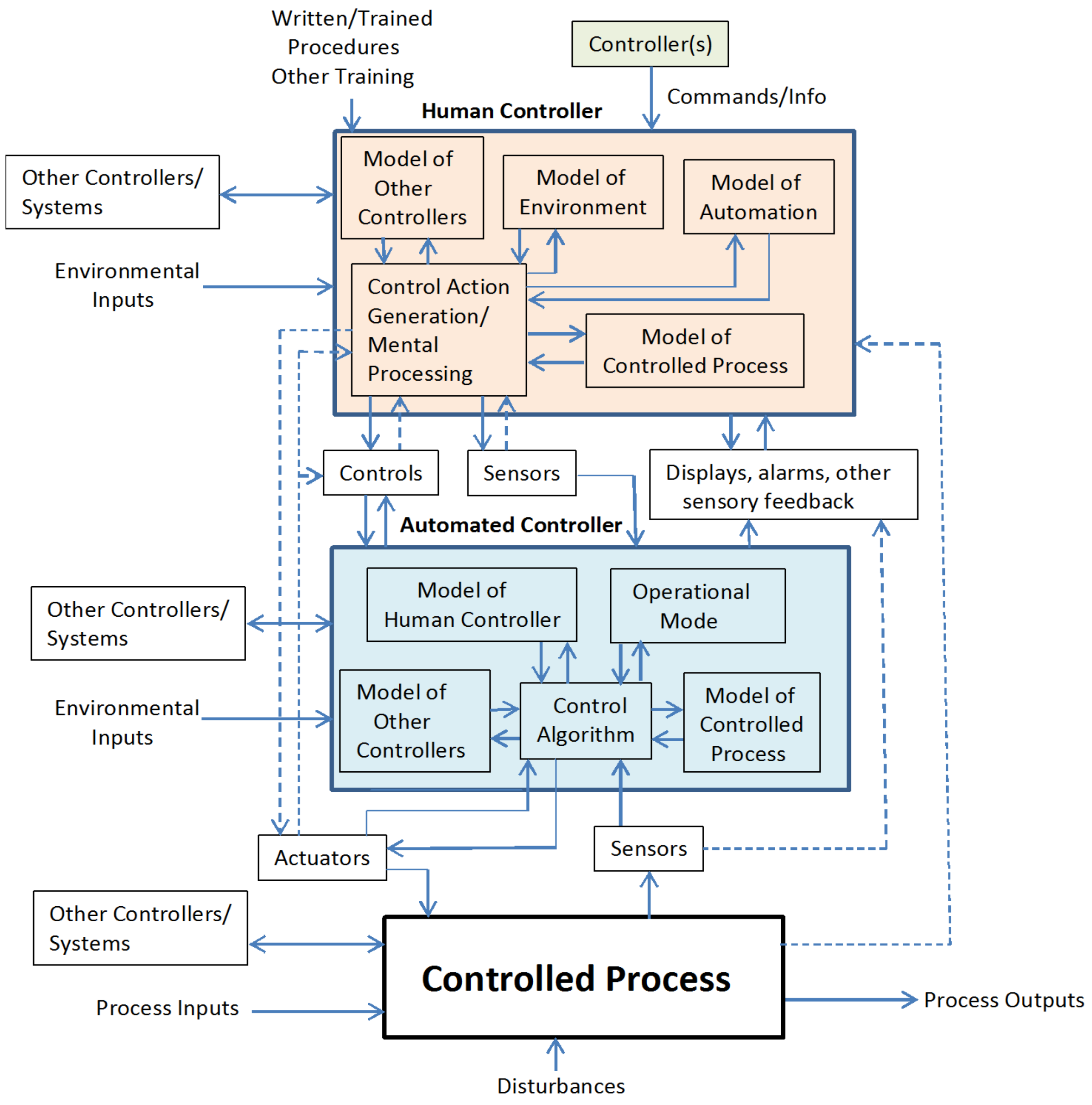
Currently, engineers jump right into detailed logical and physical design, often with incomplete understanding of the requirements and constraints, especially those related to safety and security, that the architecture needs to satisfy. If it is later determined that there are potential safety and security flaws in the architecture generated, changes to achieve these critical system properties are going to be either enormously expensive or may be infeasible and require operational controls of limited effectiveness and reliability. Some upgrades may be impossible or very expensive.

By creating conceptual architectures before concrete architectures are designed, the potential for designing more effective safety and security into systems from the beginning and enhancing the upgrade and maintenance processes is possible. Later changes simply start at the conceptual architecture and identify changes in requirements and constraints that can guide and limit the effort involved in the upgrade process.

This paper has argued that the general form of the architecture as a control structure is better for systems that are control-oriented vs. object-oriented. Solution structure should match the problem structure in order to make problem solving easier. Designing basic architectures as objects with functions assigned to them can lead to architectures for control systems that are difficult to assure and to change. Having the basic architecture designed as a control structure instead and using STPA for analysis of the evolving design and design decisions can lead to enhanced assurance and changeability.

Appendix A: The Contents of a Conceptual Architecture

Figure 10 shows a conceptual control model for a generic control system architecture. The figure is repeated here to reduce needed page flipping.


Written/Trained Procedures
Other Training
Controller(s)
Commands/Info
Human Controller
Other Controllers/ Systems
Model of Other Controllers
Model of Environment
Model of Automation
Environmental Inputs
Control Action Generation/ Mental Processing
Model of Controlled Process
Controls
Sensors
Displays, alarms, other sensory feedback
Automated Controller
Other Controllers/ Systems
Model of Human Controller
Operational Mode
Environmental Inputs
Model of Other Controllers
Control Algorithm
Model of Controlled Process
Actuators
Sensors
Other Controllers/ Systems
Controlled Process
Process Inputs
Process Outputs
Disturbances

This model is augmented from our past models of a control structure in order to include more facets of an architecture. The conceptual architecture for a specific system will particularize this generic model into a conceptual model (abstraction) to describe the solution for a specific problem. The generic components in Figure 10 will correspond to the parts of your model. A description is included here of each of the components of the generic model along with descriptions of some of the analyses that might be performed.

Controlled Process

A hazard is defined in terms of the state of the controlled process at the bottom of Figure 10, e.g., the attitude of the aircraft, the speed of the automobile, the position of the robot. States are composed of components or variables. As the goal of STPA is to identify how the controlled process could get into a hazardous state, then we need to look at the ways the controlled process can change state. Anything that can change that state may potentially be part of a causal scenario for a hazard.

Failures or Degradation over Time of the Controlled Process Components

Failures of the controlled process hardware can cause it to get into a hazardous state. Changes over time of that hardware, such as corrosion, might make it operate differently than designed. Controls such as hardware interlocks may also fail or degrade in such a way that a hazardous state is created. Remember, the goal is not to do a FMEA, that is, to look at all possible failures and their results. Instead, STPA starts from hazards and determine what conditions or events in the controlled process could contribute to that hazardous state. As an example, let’s assume that the hazard of concern to the causal analysis is the loss of pitch control of an aircraft. Failure of the aircraft pitch control devices, such as the slats, flaps, ailerons, and elevator, could lead to the hazard. If the hazard is a lack of effective braking (deceleration), that may be due to insufficient hydraulic pressure (pump failure, hydraulic leak, etc.). However, by starting from the hazard, STPA considers only the specific failures that could lead to the hazard, not all failures.

External Disturbances

Disturbances to the process can lead to a hazardous state, such as lightning affecting an aircraft’s electrical system. Or a bird is sucked into an engine and affects the power (thrust) of the aircraft. The national airspace may be disrupted by weather that increases the hazardous state for individual aircraft currently in that airspace. The environment may interfere with the proper execution of the controlled process, such as inadequate braking due to a wet runway (e.g., the wheels hydroplane).

Direct Inputs to the Controlled Process

Direct inputs to the process may affect its state. For an aircraft, loading and unloading passengers or cargo change the weight, which can affect the controllability of the aircraft. For the national airspace, inputs are aircraft that enter the airspace and outputs are those that leave it. If the components of the controlled-process hazardous state are affected by any inputs (or outputs), then those process inputs and outputs must be considered in causal scenario generation.

Controllers of the Controlled Process

Clearly, the controllers of a hazardous process can potentially contribute to a hazardous state. The model shows the primary controllers in the blue and tan boxes. In an aircraft, these may be the human pilots and automated flight control system(s). Note that the model does not imply that the pilot(s) are physically onboard the aircraft.

The model in Figure 10 shows another box to the left of the controlled process that represents other controllers or systems that can directly impact the state of the controlled process or its components. For example, maintenance activities or the lack of them can contribute to hazards in physical systems. The impact of other automated and human controllers is described separately below.

<u>Automated Controllers</u>

Only one automated controller is shown in the diagram (the blue box) although there may be several of these and they may be designed with more hierarchical levels. Because the model for automated and human controllers always have the same components, only one is shown here.

Except in very simple systems, automated controllers provide control actions to actuators, which change the controlled process state. Sensors provide feedback about the current process state.

Control Path from the Automated Controller to the Controlled Process through the Actuator

This control path transfers control actions from the automated controller to the controlled process. The control path may consist of a simple actuator, may involve a series of actuators, or it may transfer control actions through a complex network with switches, routers, satellites, or other equipment. No matter how it is implemented (which will be determined later in the physical/logical architecture design process), problems along this control path can lead to hazardous controlled system states when control actions needed to prevent a hazard are not executed, are improperly executed, or are delayed to the point where they become hazardous under some conditions. Examples include actuator failure, transmission problems along the control path, or delays in sending or executing the control action. For hazards related to security flaws, an adversary may impede a control action required to prevent a hazard or inject a control action that leads to a hazard in the controlled process. The basic STPA analysis on the generic architecture will provide information about how best to design the physical control path.

Feedback Path from the Controlled Process to the Automated Controller through Sensors

As with the actuator control path, flaws in the feedback path may stem from transmission problems including delays, measurement deficiencies and inaccuracies, sensor failure, or other flaws related to sensor design or operation. Again, an adversary might negatively impact the feedback path. If a flawed process model in the automated controller can contribute to an unsafe control action, then the feedback path should be considered in the causal scenario generation. For example, data may be delayed to the point where it no longer reflects the current state and the controller acts on incorrect information. The analyst should not stop with "flawed process model" in the causal analysis generation. It does not provide enough information to design safeguards against the unsafe control action. Instead, the analyst needs to determine why the process model could be flawed. Note that the safeguards, as they are designed, become part of the conceptual architecture.

Automated Control Algorithm

The automated control algorithm has two primary functions: (1) generate control actions and (2) maintain accurate information (models) about the state of the controlled process and external system components and environment that can impact the generation of control actions. Flaws in either of these functions can lead to UCAs. These two functions must be included in the later concrete software architecture. While more standard software architectures could be used, alternatives might be considered that directly reflect the conceptual automated controller design. Such a software design approach (and other changes in the software architecture to directly reflect the design of the conceptual architecture) could have important advantages in terms of verification of the "safety" of the software, i.e., consistency with the conceptual architecture and thus with the system safety requirements and constraints implemented by the generic architecture, maintainability, evolvability, certifiability, etc. While standard software designs might be used, alternatives that are closer to the conceptual architecture might be considered.

1. Generating Control Actions: The automated control algorithm may generate unsafe control actions due to flaws in the control algorithm itself, unsafe commands provided to the automated controller from an external source, or flaws in the models that the automated controller has of the controlled process and its environment. The causes of the latter are described in the next section.

   STPA generates the safety requirements and constraints for the automated control algorithm. These must, of course, be verified to be correct. A causal factor here, then, is inadequate communication and verification of the operational safety requirements and constraints for the control algorithm.

2. Maintaining Internal Models of the Contextual Factors Associated with UCAs: The safety of decision making about what control actions to perform are impacted by the correctness of the information that the automated controller uses to make those decisions, even if the actual automated algorithms satisfy the safety requirements and constraints. The automated controller uses four types of information in that decision making: (1) the state of the controlled process and critical information about its environment, (2) the state of other controllers of the controlled process, (3) the operational mode of the automation, and (4) the state of the human controller (in some sophisticated new systems). Various types of flaws in these models can lead to unsafe control actions by the automated controller. The context for the UCAs will contain the information about what flaws in the model can lead to the UCA. The scenarios will include why an unsafe control action might be given in that context.

Model of the Controlled Process

The model of the controlled process is the state that the automated controller thinks that the controlled process is in. This model may include information about the environment in which the controlled process is operating. The model is updated by the controller; this updating process is defined as a function implemented by the control algorithm in the conceptual architecture. Refinements of the control algorithm abstraction may separate it into separate functions, depending on the system requirements and constraints such as reliability and fault tolerance concerns.

The automated control algorithm gets direct information about the controlled process state from sensors that are designed to measure controlled process variables. There may also be designs where the process model is automatically updated without going through the control algorithm. An example is the loading of specific flight data right before launch of the Space Shuttle. Missile systems also do this. Appendix B of Engineering a Safer World details a very costly accident arising from errors (typos) in a software flight data load.

If the control algorithm is separated into several pieces (such as handling of regular braking and automated braking), there are various ways that updating of the controlled process model may be flawed through unknown or unconsidered interactions among the updating actions. These cases should be considered when identifying scenarios that lead to UCAs. Consolidating the updating actions in one module in the conceptual architecture may simplify the analysis process and reduce the number and type of UCA causal scenarios related to updating the controlled process model. The next section of this paper provides examples.

In updating the controlled-process model, the control algorithm may receive indirect information about the state of a controlled process or make assumptions about it. One common assumption is that when the control algorithm issues a control action, it may assume the action is executed and update its process model. For example, the Electronic Flight Control System on an aircraft issues a control command for an actuator to move the elevator a certain number of degrees. It could then assume both that the elevator has moved and that it has moved by the specified amount. Reasons for the commanded movement not occurring could involve flaws in the communication links to the actuators, inadequate actuator operation, and failures or faults in the controlled process (the elevator itself).

The UCA causal scenarios will include potential missing feedback about the non-execution of the command, including feedback not in the original conceptual architecture, or included but not received or not processed by the automated controller. Various types of technical design issues may also be involved such as latency (for example, the time between when an input is received and when it is processed, which will be impacted by software architecture design decisions such as the use of interrupts or polling). Data age problems have also led to hazards, that is, data is used that is no longer valid or output commands are delayed in their execution to the point where they are hazardous. The latter can occur because of delays in the actuation path or because of conditions in the controlled process. For example, on one military aircraft, a maintenance worker was working on the bomb bay door and activated a mechanical interlock to prevent the door from closing while he was in its path. At the same time, other people working in the cockpit issued a command to close the bomb bay door. The command did not execute (because of the mechanical interlock) until several hours later when the maintenance worker released the mechanical interlock. The worker was killed. By separating the development of the conceptual architecture from the physical/logical architecture, STPA can identify the potential causes of these flaws (in this case, flaws in the controlled-system process model) and the physical/logical architecture can later be optimized to prevent them. Essentially, harking back to the general design principles described earlier, separation of concerns can be used along with stepwise refinement of the design as the problem is better understood through the system development process.

There are many other reasons for inconsistencies between the process model and the actual process state (and between models in various controllers). One possible reason is delays in updating process models, which can lead to inconsistencies and unsafe control actions. Preventing such delays needs to be a consideration in the later design of the physical/logical architecture. In initial startup, designers may have assumed default values that are not correct in some cases. For example, some aircraft automation is supposed to be powered up and initialized before takeoff. Default values are often provided for that case. However, if the device is started up later than takeoff under unusual circumstances, the default values may not hold.

TCAS, for example, assumes that it has been started while the aircraft is on the ground and initializes the system with values corresponding to being on the ground. The same is true of process models for devices that are shut down for maintenance and restarted with perhaps an assumption that the controlled process state is the same as when the system was shut down. Another example is that human controllers may decide to restart a device when it does not appear to be working correctly. TCAS, for example, allows pilots to reboot TCAS in the air. The device, however, restarts with initialization values that may not be correct at that point in the flight. Input to update the process model may also arrive before the device is powered up, after shut down, or while the device is disconnected (off-line) and therefore may be ignored. Again, all of these considerations can be identified and handled in the development of the conceptual architecture and optimized in the design of the physical/logical architecture.

Model of the Operational Modes

The model of the operational modes may also be related to the context defined in the UCAs. There are four types of modes that must be considered: (1) the controlled process mode, (2) the automation mode, (3) the supervisory mode, and (4) the display mode.

1. Controlled Process Mode: The controlled process mode identifies the operating mode of the controlled process. For example, an aircraft (controlled process) may be in takeoff mode, landing mode, or cruise mode. The requisite modes to model will depend on the contexts identified for the UCAs. For example, if the aircraft is in cruise mode (rather than in landing mode), operation of the reverse thrusters can be unsafe. That context should be identified in the UCA context table (i.e., operation of the reverse thrusters when the aircraft is not in landing mode). The reasons (causal scenarios) for the UCA will be at least partially related to the automation being confused about the current mode of the controlled process.
2. Basic Automation Operating Mode: In addition to the controlled process, the automation itself (automated controller in Figure 10) has modes of operation. Those modes will often be used in the logic of its control algorithm and affect its outputs. Examples include nominal (normal) behavior mode, shutdown mode, and fault-handling mode. As another example, the automation may be in partial shutdown mode after detecting a fault in itself or in the controlled process.

   In the Air France 447 loss, the autopilot shut itself off (went into shutdown mode) when a pressure probe (pitot tube) on the outside of the aircraft iced over and the autopilot could no longer tell how fast the aircraft was moving. The fly-by-wire system, at the same time, continued operating but switched into a mode where it no longer provided protection against aerodynamic stall. On modern Airbus aircraft, operational modes of the electronic flight control system include normal, alternate, direct, and mechanical laws, each of which changes the behavior of the automated flight control system. STPA will identify the scenarios involving these modes that can lead to hazards and that therefore need to be considered in the conceptual architecture development.

   Partial shutdown modes in automated controllers can also be very confusing to human controllers, which is covered later under the things that can go wrong with human controllers. Mode confusion arises when the mode of the controlled system is different than the controller thinks it is. Mode confusion has contributed to many accidents and must be considered when generating causal scenarios by STPA and in the design of the conceptual architecture.

   In fault-handling mode, the automated controller may change its behavior because it believes there is a fault in the controlled process or in itself and different behavior is necessary to provide safe control.
3. Supervisory Mode: This mode identifies who or what is controlling the automation at any time, when multiple supervisors may be in control of (provide control commands) to the automated controller. Basically, the supervisory mode allows coordination of control action implementation among multiple supervisors. For example, a flight guidance system in an aircraft may be issued direct commands by a human pilot or may receive commands from other system components, which could be either human or automation. Another example is a tethered UAV, which might receive commands from a pilot in the lead aircraft or from ground controllers such as ATC. While Figure 10 abstracts all automated controllers into one box for generality, there may be and usually is more than one automated controller. One controller may control multiple others in a hierarchical arrangement, or multiple controllers may operate in parallel, or both. This, of course, can get very complicated, e.g., the multiple controllers may each be supervised by different controllers, etc. All of this needs to be sorted out in the development of the conceptual architecture before concerns such as actual physical realization of the architecture are tackled.
4. Display Mode: This mode specifies what type of information should be displayed to a human controller of the automated controller. The current display mode will affect both the information provided as well as how the user will interpret it. One of the components of the model of a human controller in Figure 10 is the current display mode he/she is seeing. If these display modes (in the human and the automation) become inconsistent, then serious problems can result, i.e., the

behavior on the part of one or both can lead to a system hazard. This type of mode confusion is called "Interface Interpretation Mode Confusion" and is described further in Chapter 9 of Engineering a Safer World.

Model of the Human Controller

In some systems today, a model of the state of the human controller, such as drowsiness or inattention, is used by the automated control algorithm to determine its behavior. Increasingly, sensors are being used to gauge the state of human controllers. For example, cars try to detect whether the driver is intoxicated or inattentive through various types of sensors that provide input to the automated controller. A dotted line from the sensor to the human controller is shown in Figure 10 to denote a design where the human gets feedback about whether the sensor is working or what the state of the sensed variables are (e.g., the driver's intoxication level).

Model of Other Controllers

If the automation has multiple potential controllers, then the automation may need to have a model of important state variables in those controllers.

Environmental Inputs

There may be inputs from the environment that go directly to the automated controller and not through the human controller. For example, some automation today gets direct information about location from GPS. GPS location information is calculated from signals in the environment (e.g., through antennas) and goes into the automated controller, not through the human controller. In the newest aircraft, there is something called ADS-B (Automatic Dependence Surveillance), which can be used to communicate information between different aircraft, again without going through the human controller. Note that this does not include information about the state of the aircraft itself (e.g., input from pitot tubes) which is sent via sensors in or on the controlled process and is part of the model of the controlled process.

Other controllers/systems

In some complex systems today and certainly increasingly in future systems, there may be multiple controllers of the automation. For example, there may be the usual human control of the aircraft automation with perhaps some control of the aircraft automation outside the aircraft itself (usually on the ground but it could also reside on another aircraft to which the automated aircraft is tethered), which may be human or automated. The control responsibilities may be shared by different controllers.

Figure 10 shows only one controller (in tan) and the possibility of others in a box connected to the automation as each of the controllers will have the same internal details.

As mentioned when describing the operational mode, the automation may need to know what controller is controlling it at any time. With shared control comes the possibility of conflicting commands. Some confusion may result from inconsistencies between the models in the automation and the models in its controllers. Such confusion can lead to UCAs. The possible types of inconsistencies and confusion can provide important contributions to causal scenario generation and design of the conceptual architecture.

Transmission of information between the automation and its supervisor(s)

Human controllers transmit control commands via various types of controls and get feedback from the automated controller through displays. Once again, flawed transmission of information between the automated controller and its controllers will be an important (although straightforward) part of the causal scenario and conceptual architecture generation processes.

Direct changes to the automated controller that do not go through the control algorithm

In some systems today, and certainly increasingly in future systems, the control algorithm and other software, as well as all the internal models used by the automation's control algorithm may be directly changeable from the outside without the human controller knowing about it. For example, on some aircraft, new software may be updated on an aircraft over the internet without the pilot or even the airline's maintenance personnel being involved (for example, the changes may originate with an OEM). The changes may affect both the control algorithm itself and the models used by it, such as maps and other information in the automated controller and indirectly (perhaps through displays) by human controllers. There are clearly potential safety and especially security issues here. Such changes must be considered when generating the contextual factors in a UCA and when designing the conceptual architecture.

<u>Human Controllers</u>

Human controllers are the most complex component of the model in Figure 10 (although system designers are quickly increasing the complexity of automated controllers) and thus provide some of the most important parts of the causal scenario and conceptual architecture generation processes. France has defined an extended model of human controllers.[18] All the same components are in this model, but they are connected differently than France did. An important role for this part of the conceptual architecture development is to allow interaction and collaboration between human factors experts and hardware/software engineers. While the human controller model is obviously not "implemented" in the physical/logical system architecture, the model provides important information for the concrete architecture. It also serves as a conceptual architecture for Human–Machine Interaction and augments communication between the hardware/software engineers and the human factors experts so that integrated system design, including operators and physical system designers, can be achieved. Implications of identifying human controller hazardous scenarios will also impact the physical human-machine interface design and operator training.

There are many relevant components of the model that should be included in the conceptual architecture, including control action generation and mental processing and the mental models related to these functions.

Control Action Generation/Mental Processing

Although these two functions could be modeled as different components as France did, they have important interactions that can be lost in such a separation, such as inadequate updating of the mental models because of distraction or overload caused by other human processing functions. Therefore, they are combined here to assist in the generation of the conceptual architecture based on the STPA-generated causal scenarios for human-controller UCAs about why the mental models may not be properly updated. This component thus has two basic functions: generating control actions and updating mental models.

1. <u>Generate control actions</u>: Control actions are generated using information from all the models in the human controller noted in Figure 10 (environment, automation, and controlled process) as well as goals, training, written procedures, and experience. External factors will have an impact such as time pressure. Control action generation will be affected by commands from other controllers with which

[18] Megan France, Engineering for Humans: A New Extension to STPA, M.S. thesis, Aeronautics and Astronautics Department, MIT, June 2017.

the human operator interacts. For example, a pilot is partially controlled by control actions or information received from the airline operations center or from other controllers, such as Air Traffic Control or onboard instruments and systems such as TCAS (Traffic Alert and Collision Avoidance System).[19]

When there is unclear delineation of responsibilities for control, coordination challenges increase as do the causal scenarios for unsafe control actions. For example, the FAA defines the relationship between pilots and airline operations centers as 50/50 with respect to making some specific decisions related to aircraft control. Without careful delineation of responsibilities (which may be ultimately the responsibility of the Director of Operations for the airline), confusion about who is making the decisions can lead to hazards and accidents. This has happened.[20]

There are, of course, a very large number of incorrect behaviors possible during control action generation. The UCA generation process (the results of which are documented in the UCA tables) to identify the context in which control actions are unsafe, along with the process used by the human controller to identify the current context, will serve to limit the number of causal scenarios generated. During the scenario generation process, the analyst will need to determine how the identified context associated with the UCA could occur and why the controller might be confused about the current context. All of this information will be used in the design of the conceptual architecture.

The human controller can in some systems interact directly with the controlled process (shown by dotted lines between the human controller and the actuators and sensors and the controlled process) without going through the automation.

2. Update mental models: Besides generating control actions, a major responsibility of human controllers is to update their mental models. Initial mental models may be created through training, documentation, or other experiences. Mental models are always updated through processing by the human mind, although some of this processing may be subconscious. Thus, there are two-way arrows in the model—humans both update and use information in their mental models and things can go wrong in both functions.

   The information the human uses in this updating is partially obtained through the displays, partially through inputs from its controllers or other controllers with which it interacts (e.g., information transmitted from air traffic control or the airline operations center), and partially from direct sensory input to the controller (e.g., looking out the window and seeing a tornado approaching or feeling turbulence on an aircraft). For example, a pilot receives information (such as NOTAMs) sent from the airline operations center or from other controllers, such as Air Traffic Control. In some systems, the human operators have the ability to directly observe the actuators operating, which is depicted in Figure 10 by a dotted line from the actuator to the human controller.

   For example, in some plants, operators may be able to walk up to the actuators and valves and watch them open and close to check for proper operation. As another example, pilots usually do a walk-around before taking off or during a pre-flight inspection they may move the controls and

[19] While TCAS could be considered to be an automated controller (in the middle of the hierarchical control structure in Figure X), pilots are required to follow TCAS resolution advisories (control commands) and thus in some respects TCAS controls the pilots. However, pilots can input settings for TCAS to control how it generates resolution advisories. As more and more functions are automated on aircraft and other systems, various types of "partnerships" between humans and automation will be necessary and coordination challenges will increase. Updates to the general conceptual architecture, shown in Figure X, may be required.

[20] Shem Malmquist, Nancy Leveson, Gus Larard, Jim Perry, and Darren Straker. Increasing Learning from Accidents: A Systems Approach Illustrated by the UPS Flight 1354 CFIT Accident, May 2019. Downloadable from sunnyday.mit.edu/UPS-CAST-Final.pdf

directly watch the aircraft ailerons to ensure they move in the correct direction. Human controllers may also receive input from the state of the controls, for example, the driver may see the steering wheel move on its own and assume that the automation has issued a control command related to steering.

Model updating may also be affected by the training and documentation that the human operator (controller) receives as well as whether any inputs or feedback are correctly perceived and interpreted [France]. Another subtle type of input may come from the reaction of the controls when the pilots operate them (shown as a dotted line in Figure 10 from the controls to the human controller). Operators may make assumptions about the state of the automated controller and controlled process based on the movement of controls, such as the movement of the control column on an aircraft or the steering wheel on an automated car.

3. <u>Interactions between updating models and generating control actions</u>: As with automated controllers and their updating of their process models, there can be important interactions between the control generation process and the mental models of human controllers similar to those that were described for automated controllers. For example, if a human controller issues a command to "Do X," human controllers are likely to assume that the "Do X" command has been executed by the Automated Controller and the actuators for the Controlled Process and unconsciously update their mental models about the state of the automation or the controlled process. If, in reality, the "Do X" command is never executed, then the process models of the human controller will be inconsistent with the actual state of the automation or controlled process. There can be many reasons why the command may not be executed, for example, it may be blocked and ignored if the automation considers the command to be unsafe or not possible at that point in time or there can be faults or failures in the controlled process.

   This same type of flawed updating of process models was described in the section on the automated control algorithm. But in the case of the automation, the algorithm can be checked to ensure that this problem does not occur. It is more difficult to preclude this from human mental processing.

   Feedback about the non-execution of the command may be provided to the human controller, but the human may not notice it because of distraction or other reasons.

Mental Models used by Human Controllers

Four types of mental models are identified in Figure 10 for the human controller: a model of the environment, a model of the state of the automation, a model of the controlled process (including beliefs about how the system will behave in a particular mode or stage of operation), and a model of other controllers. Any of these models may be inconsistent with the actual state and thus lead to unsafe control actions. These models have similar content and use as the same models in the automated controller, although the reasons for their being flawed may be quite different. The STPA-generated causal scenarios must include potential reasons for (causes of) dangerous inconsistencies that can give rise to the contextual conditions in the UCAs and these scenarios must be considered in the design of the conceptual architecture.

Because the human controller is usually controlling both the automation (directly) and the controlled process (indirectly but in some cases directly), humans need models of the states of both.

In order to supervise automation, humans need to have basic understanding of how the automation operates. Confusion about this can lead to unsafe control and losses.

There are a large number of reasons why human mental models may be incorrect. Some of the same reasons given earlier for models being incorrect in automated controllers may apply. But human processing presents many other opportunities. We highly recommend that human factors experts be part of the causal scenario generation involving human controllers. Here is a list of some (but only some) factors that should be considered:

- Mode confusion is a common cause of accidents in mode-rich systems where the human is confused about the mode of the automation or the automation may be confused about the current mode of the controlled process. Mode confusion may be caused by incorrect updating in process models, inputs may be incorrect or delayed, updating may be delayed or the human controller may be informed of the mode change but does not notice or process that information. When automation can change the mode of the controlled process without being directed to do so by the human controller, mode confusion and potential unsafe control actions can result.
- "Situation awareness" is commonly cited as the cause of accidents without a careful definition of what that term means. A large number of errors can fall into this category. Note that the modeling used in STPA can identify situation awareness errors as they simply mean that the human controller mental models do not match the real state. The conceptual architecture should be designed to minimize such errors.
- Humans can easily be confused by automation that is nondeterministic or acts one way most of the time but in a few special cases behaves differently. Such automation also greatly increases training difficulties. Careful design of the conceptual architecture should be able to reduce such confusion as well as the other types of errors described here.
- Some automation is programmed such that the logic can result in side effects that are not intended or known by human controllers. If such side effects cannot be eliminated, then they need to be part of pilot training if they could lead to hazardous behavior.
- While the design of the system may include feedback to humans, there are many reasons why that feedback may not be noticed such as distraction or lowered alertness when monitoring something that rarely changes or is rarely incorrect.
- Complacency and overreliance on automation by humans is increasingly becoming a problem in automated systems today.
- Automation may fail so gracefully that the human controller does not notice the problem until late in the process. Humans may also think that automation is shut down or has failed when it has not. This type of problem has arisen when robots and humans must work in the same areas. The logout/tagout problem, where humans think energy is off but is actually on, leads to a large number of accidents in the workplace.

This list is only meant to indicate that there are many causes that must be considered when humans are part of a system. It is far from exhaustive. Working with a human factors expert is recommended if causal scenarios need to get to this level of detail in order to be eliminated or controlled in the conceptual architecture.

The design of the human controller conceptual architecture and its use in both the design of the human–automation interaction and the human–automation controls, training, etc. is so important and complex that another whole paper on this part of the conceptual architecture may be needed.