Systems Security Engineering for Mission Assurance

System-Theoretic and Technical Operational Risk Management (STORM)

Tyson Brooks[1], Shiu-Kai Chin[2], Erich Devendorf[3], and William Young[4]

[1]Department of Defense
[2]Syracuse University
[3]US Air Force Research Laboratory
[4]USAF 53^{rd} Electronic Warfare Group

Copyright © 2018 Tyson Brooks, Shiu-Kai Chin, Erich Devendorf, and William Young

Contents

List of Tables

List of Figures

Chapter 1

Executive Summary

STORM (System-Theoretic and Technical Operational Risk Management) is a methodology that is completely consistent with existing paradigms such as the Risk Management Framework, the NIST Cyber Security Framework, and the NIST 800-160 Systems Security Engineering guidelines [30]. While the NIST guidelines are a good start, implementation has been hampered by legacy thinking emphasizing the use of checklists and compliance as the means for assuring both Information Communications Technology Infrastructure and the supported missions. STORM provides a set of tools and methodologies that allow leaders and technical staff responsible for both the mission and the technology to apply a secure-systems engineering approach to both frame and address risk.

This report describes in detail how STORM works and what STORM produces. STORM assures missions and manages risk by

1. Devising and validating a CONOPS (Concept of Operation) is right for the mission, and
2. Formally verifying the mission CONOPS satisfies mission requirements and constraints.

STORM uses a rigorous process starting with mission statements, unacceptable losses, and functional process models to derive validated CONOPS. The derivation depends on determining the control actions necessary to avoid losses and preserve security. This is done by generating causal scenarios resulting in losses and refining the scenarios into constraints and behavioral requirements. These requirements start from informal descriptions, are refined using scenarios that illustrate and inform safety and security constraints, which are then translated into linear temporal logic (LTL) formulas.

STORM takes validated CONOPS and refines them by devising mission-specific functions for authentication and authorization. STORM incorporates an access-control logic to formally describe and reason about access-control decisions, authentication, authorization, delegation, and trust. The access-control logic is a propositional modal logic with Kripke semantics.

STORM takes validated CONOPS with authentication, authorization, next-state, and next output functions as parameters, and expresses the CONOPS as secure state machines. The goal is the prove the CONOPS as secure state machines satisfy:

1. mission constraints and requirements, and
2. *complete mediation*, i.e., actions are taken if and only if the actions are authenticated and authorized, with no exceptions.

Proofs are done using higher-order logic (HOL) and the HOL theorem prover.

As an illustration, we apply STORM to a Unmanned Aerial Vehicle (UAV) Payload Controller used in an air interdiction mission. We start with a top-level description of the mission and go through all the steps to produce a formally verified secure state machine description of the Payload Controller that is formally verified to meet mission security and safety constraints.

Elements of STORM have been taught successfully to students within the DoD and within US universities at both the undergraduate and graduate levels. The products of STORM are easily reproduced and verifiable by independent third parties.

Chapter 2

Introduction

"Mission assurance requires systems that behave with predictability and proportionality."
– General Michael Hayden, Former NSA and CIA Director, 29 October 2009 at Syracuse University

Systems-Theoretic and Technical Operational Risk Management—STORM—is systems engineering with integrity, safety, and security at the forefront. STORM's purpose is to assure missions and manage risk. STORM's focus is *required behavior*. By focusing on behavior with the mission always in mind, we end up focusing on mission-essential functionality and mission-essential constraints. This enables us to separate "must-haves" from "nice-to-haves" and keeps first things first.

STORM has the following characteristics:

- ▷ STORM is *implementation agnostic*. It does not favor or disfavor one implementation or component over others.
- ▷ STORM is *parametric*. It does not assume specific constraints, policies, behaviors, or particular definitions of security.
- ▷ STORM is *higher-order*. This means behavior-specifying functions, such as next-state functions, next-output functions, and functions for authentication and authorization are parameters.
- ▷ STORM is *tailorable and scalable* because security is defined within the context of a mission, is higher-order, and behavioral properties are proved *for all functions specifying secure system behavior*.
- ▷ STORM is well-suited for the *Internet of Things (IoT)* because STORM focuses on mission-essential behavior, as opposed general-purpose components.
- ▷ STORM offers *rigorous assurance* as a result of:
  - A rigorous derivation of behavioral specifications from mission owners describing mission outcomes, unacceptable losses, hazards, and functional models of behavior that lead to constraints, policies, and rules of engagement.
  - Formal verification, using higher-order logic, of behavioral descriptions claiming to meet behavioral requirements, policies, and rules of engagement.
  - Easy and rapid reproduction of all assurance claims by independent third parties by virtue of the tools and methods employed by STORM.

STORM's characteristics are well-suited for serving mission-assurance needs. It starts with mission needs, rigorously develops formal requirements, and formally verifies these requirements are satisfied.

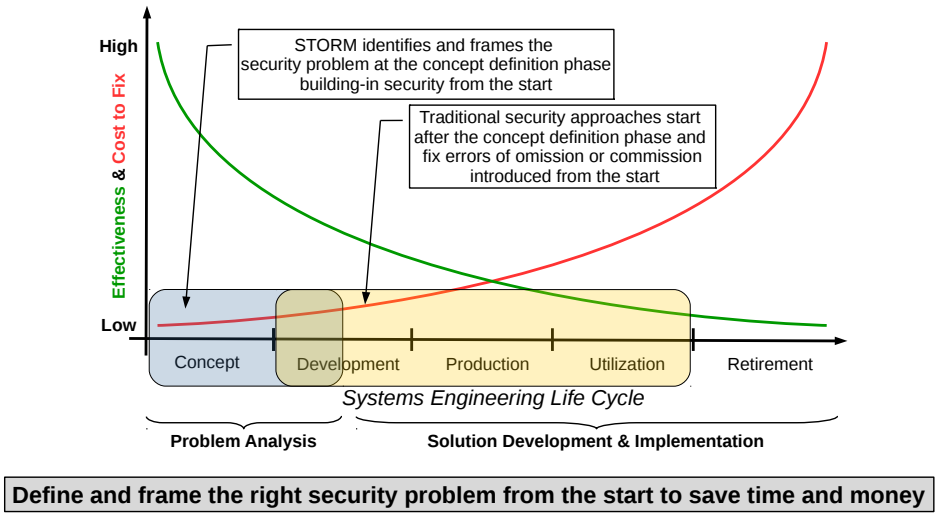


Figure 2.1: Framing the right security problem from the start saves time and money

2.1 Conceptual Motivations of STORM

"Begin with the end in mind."
– Stephen R. Covey

STORM is motivated by the accumulated experience that maximizing mission success while minimizing cost requires: (1) beginning with the end in mind, and (2) investing effort at the beginning to correctly define security requirements, constraints, and policies. When we say "security" we mean security in its broadest sense—properties describing confidentiality, integrity, and availability. What security means and its associated requirements are *mission dependent*. For example, confidentiality is of paramount importance when conveying battle plans electronically, whereas integrity outweighs confidentiality when electronically controlling ordinance seconds away from impact.

STORM's approach to maximizing mission assurance is based on promoting one outcome by avoiding two unacceptable ones.

Desired outcome:	The right product is built right
Unacceptable outcome:	The wrong product is built
Unacceptable outcome:	The right product is built wrong

2.2 Building the Right Product

"Building the right product requires systematically and relentlessly testing that vision to discover which elements of it are brilliant, and which are crazy."
– Eric Ries

Experienced planners and engineers know that the longer conceptual errors of omission or commission persist in the systems engineering life cycle, the costlier they are to fix and the more they endanger mission success. Time and effort invested up front in the conceptualization stage that rigorously eliminates inconsistencies, misconceptions, and mistakes, reduce risk and reduce cost throughout the entire systems engineering life cycle. Figure 2.1 illustrates the associated cost and benefits over time.

Fixing errors in the initial conceptual phase is quick and economical. Correcting mistakes in the design and development phases prior to production potentially risks significant amounts of time and resources to

Instance	Description	Cost
1985–1987: Therac 25	Radiation therapy device fatally irradiates patients because of flawed control software	5 dead
1994: Pentium Bug	Intel Pentium processor released with flaw in floating point division	$475 million
1996: Ariane 5 Explosion	Software error due to converting 64-bit floating point number into a 16-bit signed integer that exceeded the maximum representable number	$500 million ($7 billion spent in development)
1998: Mars Climate Orbiter Crash	Units mix-up in software: Lockheed produced results in pounds, NASA expected results in newtons	$328 million
2014: Heartbleed flaw	Security flaw in OpenSSL crypto library	$500 million estimated
2018: Spectre and Meltdown	Memory isolation in commercial microprocessors compromised due to optimization techniques at the hardware level	Not yet determined

Table 2.1: The High Cost of Design Flaws

execute. When fixing mistakes prior to production is infeasible, the price is paid for in terms of added mission risk or reduction in mission scope. Fixing mistakes in the production and utilization phases is logistically daunting. Communicating, tracking, and verifying fixes are needed and done require major efforts beyond installing the fixes alone. The price is paid in terms of added risk to missions, reduction in mission scope, lost resources, and loss of confidence.

Security approaches that seek to *bolt-on* security after a system is conceptualized or designed, incur all the risks and losses associated with remediating security flaws in behavioral requirements during the conceptualization or initial design phases. Building the wrong product for the mission squanders resources and denies resources for the mission or other purposes. France's Maginot Line shows how the wrong product—regardless of implementation quality—fails operationally, and leads to mission loss and defeat.

STORM focuses on stakeholder-defined unacceptable losses. It frames security losses as control problems. STORM defines secure behavior and formally verifies its properties, which implementations must satisfy.

2.3 Building the Product Right

"Smart, Secure Everything—where devices are getting smarter, everything's connected, and everything must be secure."

– From Synopsys company description; Synopsys is a $2.7 billion electronic design automation company

Having validated requirements for the right product is necessary but insufficient for mission assurance. Building the product right is essential. Mistakes are costly. Table 2.1 tabulates some well-publicized flawed systems with a brief description and estimated cost.

Of particular interest is the Intel floating point division error and the changes it caused in the semiconductor industry. Here is a rough timeline.

1. 1994: The floating-point division error was reported in Intel's Pentium processor. It cost Intel $475 million, [19].
2. 1996: Dill and Rushby in *Acceptance of Formal Methods: Lessons from Hardware Design*, [17], reports on companies developing formal verification capabilities in hardware, including AT&T, Cadence, Hewlett-Packard, IBM, Intel, LSI Logic, Motorola, Rockwell, Texas Instruments, and Silicon Graphics.
3. 1998: Intel's Cornea-Hasegan reports on *Proving the IEEE Correctness of Iterative Floating-Point Square Root, Divide, and Remainder Algorithms*, [16].

4. 2005: Intel's Harrison reports on *Floating-point verification* using higher-order logic and interactive theorem provers, [19]

5. 2018: Synopsys.com, a leading company in Electronic Design Automation (EDA) with $2.7 billion in revenue, describes its vision and what it does in Synopsis: About Us, as follows:

> From www.synopsis.com/company.html (bolding added):
>
> SMART, SECURE EVERYTHING—FROM SILICON TO SOFTWARE
>
> Synopsys technology is at the heart of innovations that are changing the way we live and work. The Internet of Things. Autonomous cars. Wearables. Smart medical devices. Secure financial services. Machine learning and computer vision. These breakthroughs are ushering in the **era of Smart, Secure Everything—where devices are getting smarter, everything's connected, and everything must be secure**.
>
> Powering this new era of technology are advanced silicon chips, which are made even smarter by the remarkable software that drives them. Synopsys is at the forefront of Smart, Secure Everything with the world's most advanced tools for silicon chip design, **verification**, IP integration, and **application security** testing. Our technology helps customers innovate from Silicon to Software, so they can deliver Smart, Secure Everything.

   Under formal methods tools, Synopsys includes under its Next-Generation Formal Verification tools for system on a chip (SoC) design:

   - ▷ Property Verification (FPV),
   - ▷ Auto Extracted Properties (AEP),
   - ▷ Coverage Analyzer (FCA),
   - ▷ Sequential Equivalence Checks (SEQ),
   - ▷ Register Verification (FRV),
   - ▷ Formal Testbench Analyzer (FTA), and
   - ▷ Security Verification (FSV).

The importance of Synopsys' vision and self-description is this: Synopsys and other EDA companies offer technology and tools that amount to a modern-day *Aladdin's Lamp*. EDA for design and verification gives us what we ask for. In a world where we get what we ask for, the difference between heaven and hell is asking for the right thing and *knowing that it is right before we ask for it*. What the right properties are for mission assurance is mission-dependent. Using formal methods to describe systems and their properties integrates well with existing SoC design flows that increasingly include formal verification of system properties.

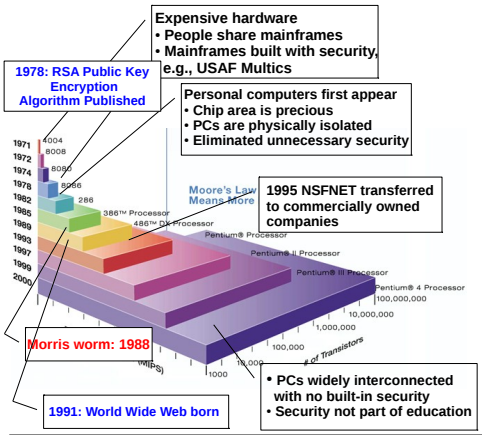
STORM describes system behavior, requirements, control policies, constraints, and proved properties using natural language, functional models, linear temporal logic, propositional modal logic, structural operational semantics, and higher-order logic

2.4 STORM Education

"I understand Mission first and People always."
– US Army Cadet Creed

Tools without people to use them are useless. Figure 2.2 juxtaposes Moore's Law, time, COTS (commercial off-the-shelf systems), and the virtual disappearance of security in university curricula.

In the 1970s, mainframe computers dominated. As mainframes were intended to be shared by many users, security was built into their design from the start. The Multics system [5][6][27], is one such example. Virtual



Commercial off the Shelf (COTS) Dominates Critical Infrastructure

- Single-user, sole physical access, costly chips leads to elimination of security capabilities
- COTS used to contain costs and add features
- Internet's rise: vulnerable PCs no longer isolated
- 1970 → 2000: 1.5 generations to forget how to design-in security

Security disappears from the curriculum

Application-specific integrated circuits are becoming pervasive in the IoT

- Incentives for custom design to achieve reduced power & cost with increased functionality

Will we repeat the mistakes of the PC era?

How can we expect mission assurance when engineers cannot design secure systems?

Figure 2.2: The Path from Security to Insecurity Despite our Best Intentions

memory, memory segmentation, descriptors, access-control lists, protection rings, etc., were the means to achieve process isolation and sharing [31][32]. These techniques were part of a computer architect's and hardware designer's knowledge base.

The microelectronics revolution spanning the mid 70s through the 90s, saw the development and widespread use of VLSI (very large-scale integrated) circuit design, [15]. VLSI technology led to personal computers (PCs), the Internet, and the World Wide Web.

At the beginning of the PC era—prior to the Internet—relatively few transistors could be fabricated on chips. Understandably, this led to shedding unnecessary functionality to contain costs. Security was not critical because the very concept of what made a PC different was it was owned and operated by individuals. The original conception of PCs did not include sharing and hence isolation was not an issue.

The focus during the 80s and 90s was increased functionality at lower cost, as exemplified by the steady release every year of new microprocessors with increased size and functionality. By 2000, the power of PCs rivaled that of mainframes.

What happened concurrently with PC development was the development of the Internet and the World Wide Web. PCs, with relatively few security capabilities, were connected to the Internet. The 30 years spanning the widespread use of mainframes with security to PCs with little or no security, saw the disappearance of security in most engineering and computer science curricula.

One of the major challenges in computer science and engineering education is to re-introduce security into the curricula in ways that enable students to routinely design-in security into their systems. Another challenge is to give them the tools to make the case that their designs are trustworthy.

An important consideration for STORM is how easily people are educated and trained in its theory and practice. The major components of STORM are Systems Theoretic Process Analysis for Security (STPA-Sec) and Certified Security by Design (CSBD). At a high level, the STPA-Sec portion of STORM supports the disciplined derivation and validation of system requirements. The CSBD portion of STORM is the formal verification that implementations satisfy system requirements.

To support education and training in STORM, there are textbooks, online asynchronous learning modules, computer-assisted reasoning tools, and numerous examples in the form of homework and projects. These materials are the result of over a decade of research, development, and educational experimentation.

STORM is taught routinely in the Air Force and in Syracuse University's graduate and undergraduate degree programs.

Chapter 3

Systems Security Engineering Using STORM

"This whole economic boom in cybersecurity seems largely to be a consequence of poor engineering."

– Carl Landwehr, *Communications of the ACM*, February 2015

Systems security engineering (SSE) is:

A discipline to achieve stakeholder objectives for the protection of assets, by means of

applying systems and security principles, analysis, and tools, in order to

produce outcomes that

1. prevent and control asset loss and associated consequences, and
2. substantiate security and trustworthiness claims using evidence-based reasoning.

Figure 3.1, taken from NIST Special Publication 800-160 [30], illustrates the nine principal tasks necessary for systems security engineering. These nine tasks collectively constitute *System Security Analysis*, and are defined and described in detail in [30]. The analysis makes explicit what concepts, principles, means, methods, processes, practices, tools, and techniques are used to solve the right problem in the right way with demonstrable evidence of trustworthiness. It frames the tasks within three categories:

1. Problem Definition
   (a) Define Security Objectives
   (b) Define Security Requirements
   (c) Define Success Measures
   (d) Define Life Cycle Security Concepts
2. Solution
   (a) Define Security Aspects of the Solution
   (b) Refine the Security Aspects of the Solution
   (c) Produce Evidence for Security Aspects of the Solution
3. Trustworthiness
   (a) Develop Assurance Case for Acceptable Security
   (b) Demonstrate Assurance Case is Satisfied

3.1 Concepts of Operation (CONOPS)

"Design is a funny word. Some people think design means how it looks. But of course, if you dig deeper, it's really how it works."

– Steve Jobs

SSE and STORM focus on Concepts of Operation (CONOPS). CONOPS are a means of describing how a system accomplishes its purpose or mission. There are at least two relevant definitions of CONOPS.

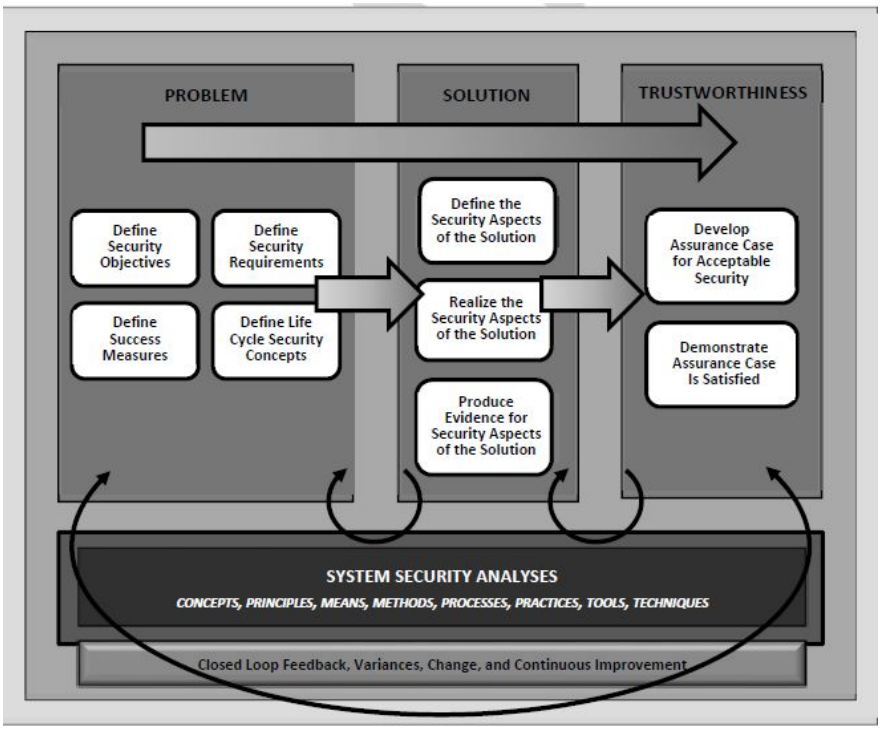


Figure 3.1: Systems Security Engineering Framework

1. The Institute of Electrical and Electronics Engineers (IEEE) Standard 1362 [1] defines a CONOPS as an expression of the "characteristics for a proposed system from a user's perspective. A CONOPS also describes the user organization, mission, and objectives from an integrated systems point of view."

2. The US military has a similar definition of CONOPS in Joint Publication 5-0, Joint Operational Planning [2]. For military leaders planning a mission, a CONOPS describes "how the actions of components and organizations are integrated, synchronized, and phased to accomplish the mission."

Simply put, a CONOPS describes who does what, when, where, and why. A CONOPS describes the flow of command, control, communications (C3) and actions taken by system components to accomplish a mission. CONOPS are used to describe systems at various levels of detail.

1. A high-level CONOPS might define a system's behavior in terms of roles and the actions taken by each role, e.g., commanders and their orders combined with operators and their actions.

2. A mid-level CONOPS might refine a high-level CONOPS by adding people and processes authorized to act in the roles defined at the high level.

3. A low-level CONOPS might include the cryptographic keys and cryptographic functions used to authenticate people and processes acting in their assigned roles.

Figure 3.2 shows a diagram of a simple CONOPS. Here is its interpretation.

1. The flow of command and control in this figure is from left to right. Alice issues a command by some means (speaking, writing, electronically, telepathy, etc.). This is symbolized by

$$Alice \text{ says } \langle command_1 \rangle.$$

2. The box in the center labeled **Bob** shows Bob receiving Alice's command on the left. Inside the box are the things Bob "knows", i.e., the context within which he attempts to justify acting on Alice's

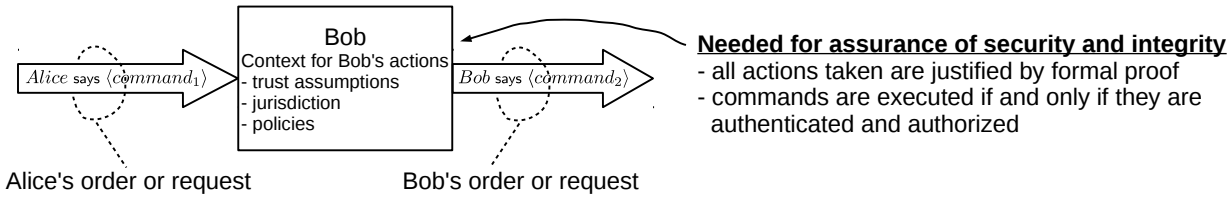


Figure 3.2: Flow of Command and Control (C2) for a Simple CONOPS

command. The context might include a policy that if Bob receives a particular command, such as *go*, then he is to issue another command, such as *launch*. Typically, before Bob acts on Alice's command, his operational context includes statements or assumptions such as Alice has the authority, jurisdiction, or is to believed on matters related to the command she has made.

3. The arrow coming from the right hand side of the box shows Bob's statement or command, which is symbolized by

$$Bob \text{ says } \langle command_2 \rangle.$$

4. What Figure 3.2 shows is one C2 sequence starting from left to right. Bob gets an order from Alice. Bob decides based on Alice's order and what he knows (the statements inside the box), that it is a good idea to issue $command_2$. This is symbolized by

$$Bob \text{ says } \langle command_2 \rangle.$$

Regarding the comment in Figure 3.2, for assurance what we want is a logical justification of the actions Bob takes given the order he receives and the context within which he is operating. For us, logical justifications are *proofs in mathematical logic*.

Security vulnerabilities often result from inconsistencies among CONOPS at various levels of abstraction. Supervisors might assume only authorized operators are able to launch an application, whereas the application itself might incorrectly trust that all orders it receives are from authorized operators and never authenticate the inputs it receives.

Any *design for assurance* methodology must address authentication and authorization in order to *avoid vulnerabilities* due to unauthorized access or control. Rigorous assurance requires mathematical models and proofs. Our intent is to provide a rigorous methodology to achieve security by design.

3.1.1 CONOPS and System Security Engineering

SSE as described by NIST SP 800-160 [30], states the importance of CONOPS in problem definition and context. Footnote 27 in [30] states:

> *"The term life cycle security concept refers to all processes and activities associated with the system throughout the system life cycle, with specific security considerations. The term is an extension of the notion of concept of operation including, for example: processes and activities related to development; prototyping; analysis of alternatives; training; logistics; maintenance; sustainment; evolution; modernization; disposal; and refurbishment. Each life cycle concept has security considerations and constraints that must be fully integrated into the life cycle to ensure that security objectives for the system can be met."* – NIST SP 800-160, page 23

CONOPS, as defined in JP 5-0 [2] include the actions of components, people, and organizations. Essentially, CONOPS are models used to describe behavior. If described formally, CONOPS give us the means to reason about their properties, including properties that *emerge* when components are composed to form systems. Security is an emergent property. To the extent possible, we use CONOPS to formally describe behavior so we can use formal analysis tools to verify security or remediate insecure behavior.

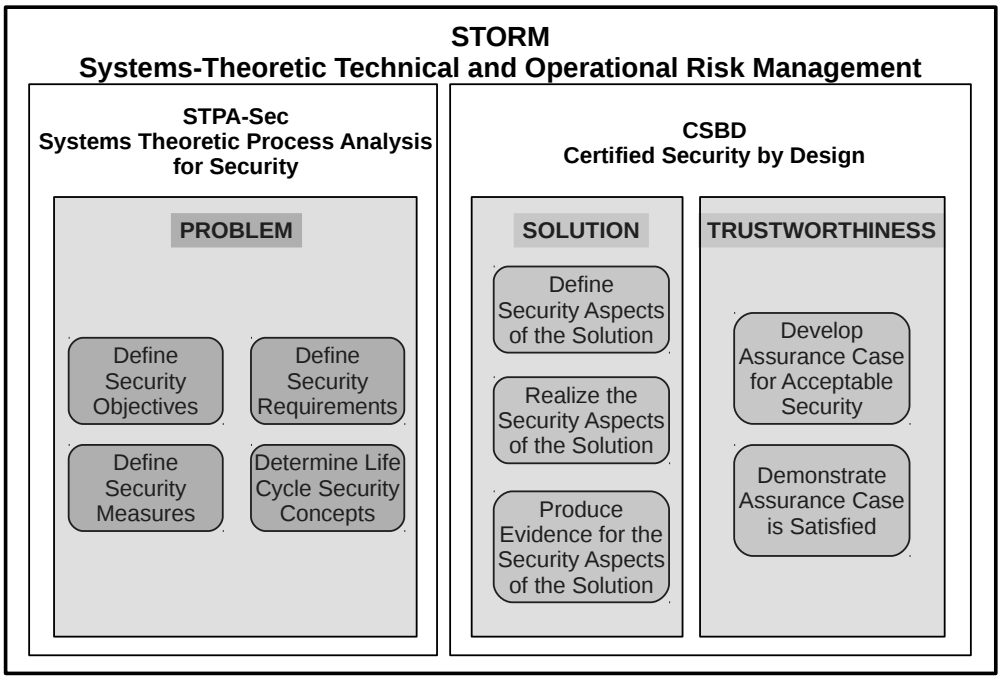


Figure 3.3: STORM's Principal Components and Coverage of Systems Security Engineering Tasks

Put another way, system security is a design problem focused on providing satisfactory controls and constraints that provide comprehensive security across the system. CONOPS model system behavior and enable us to reason about the adequacy of security controls.

3.1.2 CONOPS and STORM

System Security as a Design Problem

"Providing satisfactory security controls in a computer system is in itself a system design problem. A combination of hardware, software, communications, physical, personnel and administrative-procedural safeguards is required for comprehensive security. In particular, software safeguards alone are not sufficient." – The Ware Report, Defense Science Task Force on Computer Security, 1970

STORM focuses on CONOPS as a means to specify and describe desired *behavior*. STORM first focuses on identifying the unacceptable losses associated with a mission, then, in conjunction with functional system models, devises controls, constraints, and policies to avoid those losses. Safety and security properties are stated as requirements.

At a more detailed level, CONOPS modeled as transitions among various system configurations. CONOPS as transition systems are formally described and verified to have the desired safety and security properties.

3.2 STORM's Principal Components and Coverage of SSE Tasks

"Build the right product and build the product right."
– A product design aphorism

STORM has two major components:

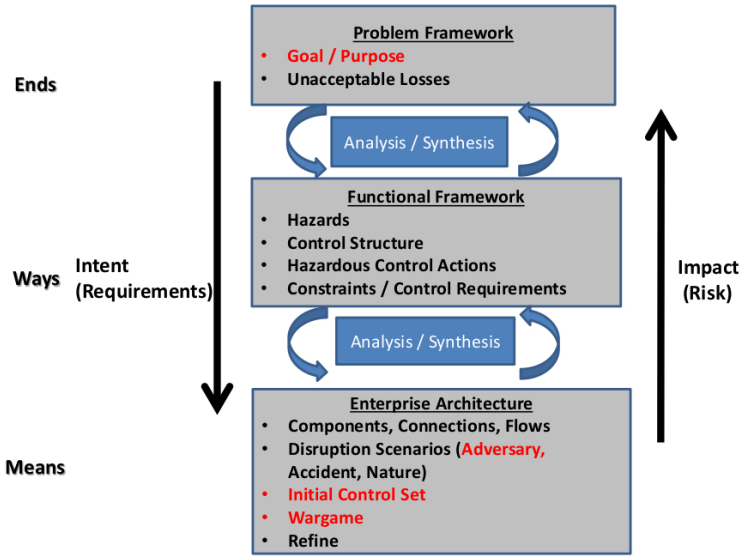


Figure 3.4: System Theoretic Process Analysis for Security

1. Systems Theoretic Process Analysis for Security (STPA-Sec), which is used to develop the right mission CONOPS, and

2. Certified Security by Design (CSBD), which is used to formally describe and verify system behavior modeled as a transition system using structural operational semantics combined with a command, control, and communication (C3) calculus with Kripke semantics all embedded within the Higher Order Logic theorem prover.

3.2.1 STPA-Sec Component of STORM

The STPA-Sec component of STORM addresses the "whys" of a system. As shown in Figure 3.3, STPA-Sec does the following to address the problem-definition phase of SSE:

1. define security objectives,

2. define security requirements,

3. define security measures, and

4. determine life cycle security concepts.

Figure 3.4 is a block diagram of STPA-Sec. It has three foci: ends, ways, and means, as reflected by the three components of STPA-Sec:

1. Problem framework: the system's goal/purpose,

2. Functional framework: identified hazards, control structure, hazardous control actions, and constraints/control requirements, and

3. Enterprise architecture: components, connections, flows, disruption scenarios, initial control set, war games, and refinements.

The elements of STPA-Sec that are security focused are: goal/purpose, adversary-based disruption scenarios, initial control set, and war games. STPA-Sec starts with the following framing question to define the system purpose and goal

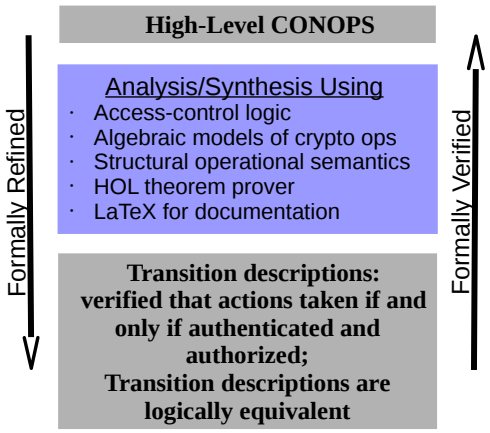


Figure 3.5: Certified Security by Design

A system to do $\{What = Purpose\}$ by means of $\{How = Method\}$ to contribute to $\{Why = Goals\}$.

STPA-Sec follows with identifying unacceptable losses, e.g., hazardous chemicals released into environment due to cyber-attack on plant's industrial control system. Next, it develops a functional control structure (functional security architecture) of the process being controlled to execute the identified mission. Hazardous control actions are identified, e.g., starting a chemical process or opening/closing water valves at the wrong time where an explosion or inadvertent release might occur, followed by a structured analysis to determine the effects when control actions are:

(a) missing,

(b) provided,

(c) incorrectly timed or ordered, or

(d) too short or too long in duration.

These analyses lead to required high-level functional constraints that inform a system's security requirements and security context. These outputs provide the initial inputs required to begin CSBD.

3.2.2 CSBD Component of STORM

The CSBD component of STORM addresses the "hows" of a system. As shown in Figure 3.3, CSBD does the following to address the solution and trustworthiness phases of SSE:

1. define security aspects of the solution,
2. realize security aspects of the solution,
3. produce evidence for the security aspects of the solution,
4. develop assurance case for acceptable security, and
5. demonstrate assurance case is satisfied.

Figure 3.5 is a block diagram of CSBD. A complete illustration of CSBD appears in [9]. CSBD starts with a high-level CONOPS describing:

(a) the principals—roles, personnel, authorities, tokens, and keys (derived from STPA-Sec control structure),

(b) the actions, commands, or requests (STPA-Sec control actions) made by principals,

(c) how commands or requests are authenticated, and

(d) the security context for determining authorizations, i.e., delegations, assignments to roles, jurisdiction, rules of engagement, policies, and trust assumptions.

CONOPS typically describe command, control, and communications (C3) use cases. CONOPS are refined horizontally by adding details at the same level of abstraction, e.g., describing C3 first in terms of roles only, then assigning personnel to roles, and next using cryptographic keys for authentication. CONOPS are refined vertically by adding details at lower levels of abstraction, e.g., adding data structures for inputs, public-key certificates, etc., and their interpretation in the access-control logic. We prove the logical equivalence of transition relations at differing levels of abstraction.

The sequential nature of a CONOPS is described as a transition system. Each CONOPS phase or state is described by a configuration. The sequential behavior of a CONOPS is defined by next-state and next-output functions combined with labeled transition relations, where the transition labels correspond to discarding, executing, or trapping commands. Unauthenticated commands are discarded, authenticated and authorized commands are executed, and authenticated but unauthorized commands are trapped. At higher levels, CONOPS correspond to C3 protocols. At lower levels, CONOPS are (finite or infinite) state machines and instruction-set architectures.

CSBD incorporates authentication and authorization into transition system descriptions. Configurations have an interpretation or meaning in the access-control logic. Commands and requests are considered within a configuration's interpretation in the access-control logic. If the request is sound within a configuration's context, then the action is both authenticated and authorized. This satisfies the property of complete mediation, i.e., the action or transition is taken if and only if the action is authenticated and authorized within the authentication and authorization context set forth in the current configuration. CSBD relies on the formally verified implementation of the access-control logic, algebraic models of cryptographic operations, and the semantics of transition systems with structural operational semantics in the HOL-4 higher-order logic theorem prover. CSBD takes full advantage of HOL's higher-order nature. Our secure state machines and configurations are parameterized in terms of input, output, and state types, next-state functions, next-output functions, authentication functions, and security contexts. This avoids state explosion. The definitions and theorems for transition systems are parameterized; specific mission values are instantiated into their corresponding parameters.

3.3 A Focus on Reproducibility

"If you've got the truth you can demonstrate it. Talking doesn't prove it."
– Robert A. Heinlein, Stranger in a Strange Land

A key feature of STORM is reproducibility. Without reproducibility, mission assurance and SSE does not scale and trustworthiness cannot be verified. STORM uses computer-assisted reasoning tools, XSTAMPP [3] and HOL [18]. These tools allow independent third parties to reproduce STORM's results.

The STPA-Sec portion of STORM is supported by XSTAMPP (eXtensible STAMP Platform). It refines safety and security requirements into Linear Temporal Logic (LTL) formulas. XSTAMPP starts with top level descriptions and guides users through a safety analysis, which results in requirements expressed as LTL formulas. These formulas formally express safety and security properties that must be satisfied by the mission CONOPS.

XSTAMPP enables people writing the requirements to link control structures with process models to control actions, constraints, hazards, and accidents. XSTAMPP functions similar to spreadsheet programs documenting and performing a numerical analysis.

The CSBD portion of STORM is supported by the HOL-4 (Higher Order Logic) theorem prover. A propositional modal logic with Kripke semantics is used the reason about the command, control, and communications (C3) portion of mission CONOPS with respect to authentication and authorization. Structural operational semantics is used to express the transitional nature of mission CONOPS combined with authentication and authorization.

Both the propositional modal logic of C3 and the structural operation semantics of CONOPS is implemented as conservative extensions to the HOL logic. By so doing, HOL's logical soundness is preserved. HOL is used to formally prove that a CONOPS satisfies the security and safety properties specified by STPA-Sec and XSTAMPP.

Independent third parties can easily recompile all mission-related theories in HOL and pretty-print the theories in LaTeX. HOL has the capability to generate LaTeX macros that typeset all formulas in theories, which eliminates errors introduced by manual typesetting. This greatly eases report generation and maintenance.

Chapter 4

Just in Time Mission Composition

"You can ensure the safety of your defense if you only hold positions that cannot be attacked."
– Sun-Tzu

Much of current critical infrastructure and military systems rely on COTS (commercial off the shelf) components and systems. In particular, most systems rely on the same operating systems, underlying chip sets, and protocols, e.g., Windows, iOS, Linux, x86-based architectures, and TCP/IP.

The consequence of widespread homogeneity is shared vulnerability across a wide spectrum of systems, including military systems. This is equivalent to having fixed defenses in the physical world. Adversaries have years to study, develop, and exploit vulnerabilities in systems, from the hardware level up through and including operating systems, software applications, and network protocols.

Current military systems are homogeneous, time invariant, and vulnerable.
Continuing to rely on COTS amounts to surrendering the initiative to adversaries.

4.1 Just in Time Mission Composition

"It is not easy to make a computer system secure, but neither is it impossible. The greatest error is to ignore the problem."
– Roger Schell

Our goal is to create a maneuver capability in cyberspace that is mission driven. Instead of the conceptual equivalent of fixed defenses in cyberspace using homogeneous COTS hardware and software, we are developing capabilities to:

- ▷ Achieve heterogeneity through one-time-use mission entities
- ▷ Instantiate entities that perform only a mission-essential functions
- ▷ Shrink the window of vulnerability of mission-critical systems from years to hours

This is an achievable goal given the EDA infrastructure developing to support the Internet of Things, as exemplified by EDA companies, such as Synopsys. EDA companies recognize the IoT as a market will reward those who are capable of delivering smart, secure, special purpose, full-custom designs delivering mission-essential functions that use minimal resources. Delivering such devices depends on the critical capability to quickly generate special-purpose systems with assurance, or *"Smart, Secure Everything—From Silicon to Software,"* as envisioned by Synopsys.

This critical capability is consistent with current trends toward custom hardware realized by reconfigurable computing coupled with automatic and autonomous software generation. The continuing and critical role of people is that of *concept and requirements creators, validators, and verifiers*, i.e., **creators of CONOPS, determining required properties, and providing credible assurance**. This critical capability is part of what defines SSE (Systems Security Engineering) [30].

The capability to generate systems on an as-needed, just-in-time mission-by-mission basis, with assurances of integrity, safety, and security, is a critical requirement for maneuverability in cyberspace. Figure 4.1 illustrates one way this capability can be achieved.

The are three components to Just-in-Time Mission Composition (JiT MC),

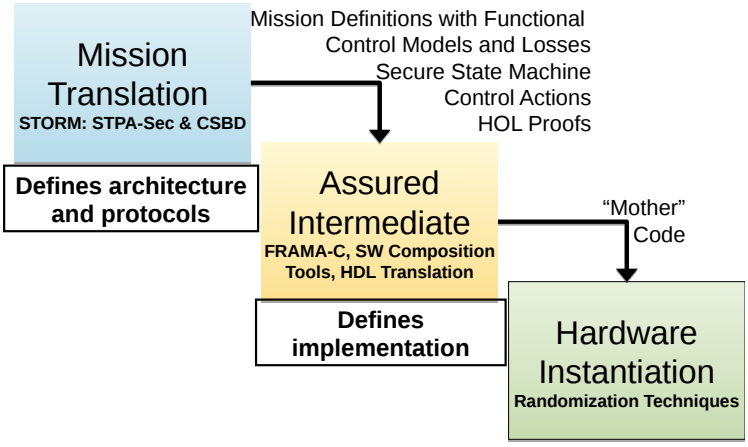


Figure 4.1: Just-in-Time Mission Composition

Mission Translation takes mission descriptions with functional control models and specifications of unacceptable losses and produces architectures and protocols in the form of secure state machines with formally verified security properties.

Assured Intermediate takes secure state machine descriptions with proved properties and produces an implementation description in a hardware description language (HDL).

Hardware Instantiation takes an HDL description and instantiates it behaviorally by randomly selecting among hardware parameters such as word length, architecture/instruction set, and instruction encoding.

JiT MC's mission specific and implementation agnostic approach is crucial to reducing vulnerability from years to hours. JiT MC does not rely on massive and vulnerable general-purpose COTS components, i.e., COTS microprocessors and operating systems. In fact, given the desire to reduce attack surface by implementing only mission-essential functions, the result is systems that are much smaller, economical, mission-specific, and feasible to assure, when compared with implementations built using components that have significant mission-irrelevant functionality, which adds risk and vulnerability.

Maneuver in cyberspace requires provably secure system design, instantiated in heterogeneous hardware that exists solely for the lifetime of a mission.

4.2 Mission Translation

Mission translation is the domain of STORM. The remainder of this report illustrates how mission definitions and enumerations of unacceptable losses, together with functional control models, are refined into secure state machines verified to satisfy mission safety and security requirements.

STORM employs both rigor and formality to accomplish mission translation. Validation is done rigorously by answering a series of questions increasing in detail and specificity. The questions start with stating the whats, hows, and whys of a mission, proceed to identifying unacceptable losses, and then looks at scenarios leading to those losses. All of this is translated into specific constraints and control actions expressed as linear temporal logic formulas.

Mission-validated CONOPS takes the form of functional behavioral models expressed as state machines with security constraints. It is the system engineer's task to devise specific functions for authentication, authorization, and next-state and next-output calculations. Once these mission-specific functions are devised, the mission-specific secure state machine (SSM) is formally verified in HOL to satisfy mission requirements.

The verified SSM is the output of Mission Translation and the input to the Assured Intermediate process.

4.3 Assured Intermediate

The input to the Assured Intermediate process are formally verified transition-systems expressed as SSMs. The outputs are descriptions in a hardware description language, such as VHDL. One candidate means to translate SSMs into assured VHDL descriptions is to use an intermediate language, such as FSMLanguage (Finite State Machine Language) [4].

FSMLanguage programs describing finite-state machines are compiled into VHDL and C. FSMLanguage was created with FPGA (field programmable gate array) hardware in mind.

FSMLanguage programs have 10 sections. The following is taken directly from [4].

State Names: internal names for FSM state variables

Generics: compile time variables

Ports: inputs/outputs from/to the outside world

Connections: permanent connections of output ports to FSM signals

Memories: internal/external memory blocks

Channels: FIFO ports to the outside world

Signals: internal FSM state

Initial: initial state definition for the FSM

Transitions: logic/behavior for an FSM

VHDL: optional section for linking in libraries of native VHDL constructs

FSMLanguage is implemented in Haskell. A SSM to FSMLanguage translation should be conceptually straightforward and likely amenable to interpretation or compilation.

4.4 Hardware Instantiation

Hardware instantiation can be done using hardware randomization techniques. Randomization techniques including instruction-set randomization, e.g., randomizing original executable code using a key to get randomized executable code, which is derandomized just prior to execution by the hardware, [24]. Instruction-set randomization can be implemented with FPGA hardware support for existing instruction sets [28]. This approach, combined with FSMLanguage support for implementing SSM descriptions, is one possible path to mission-specific randomized hardware instantiation.

Chapter 5

Using STPA-Sec to Validate a Payload Controller CONOPS

"Be careful what you ask for because you might just get it"
– Anonymous

Knowing what to ask for, with the capability of recognizing whether or not what we have is what we want, is essential for mission assurance. The STPA-Sec portion of STORM is focused on getting the right CONOPS for a mission. Getting the CONOPS right using STPA-Sec entails developing the following information.

1. We have a succinct description of the system and its goals.
2. We have a statement on unacceptable losses or accidents to be controlled against.
3. We have enumerated the hazards leading to accidents and their corresponding safety/security constraints.
4. We have developed scenarios around unsafe control actions with associated hazards based on a functional model of the system with safety/security constraints.
5. We have formal descriptions of behavioral properties of our system expressed in linear temporal logic (LTL).

As an illustration of the Mission Translation process using STORM, we work through the details of a *payload controller* on a UAV as part of an Air Interdiction (AI) mission.

The rest of this section is organized as follows.

1. For background, we give a brief description of AI and the operational assumptions.
2. We show the use of STPA-Sec to
   (a) Specify the mission and frame the security problem.
   (b) Perform secure systems analysis based on the STAMP (System-Theoretic Accident Model and Processes) model [33].
   (c) Develop STPA-Sec outputs to CSBD: accidents, losses, hazards, hazardous control actions, and safety/security properties expressed rigorously and in LTL (linear temporal logic). These constitute a validated CONOPS for the mission.

5.1 Background

Air interdiction is

> *"... a flexible and lethal form of air power that can be used in various ways to prosecute the joint operation. ... [I]t does not require detailed integration with friendly forces. Detailed integration requires extensive communications, comprehensive deconfliction procedures, and meticulous planning. AI is inherently simpler to execute in this regard. Therefore, if the enemy surface force presents a lucrative target, AI conducted before friendly land forces make contact can significantly*

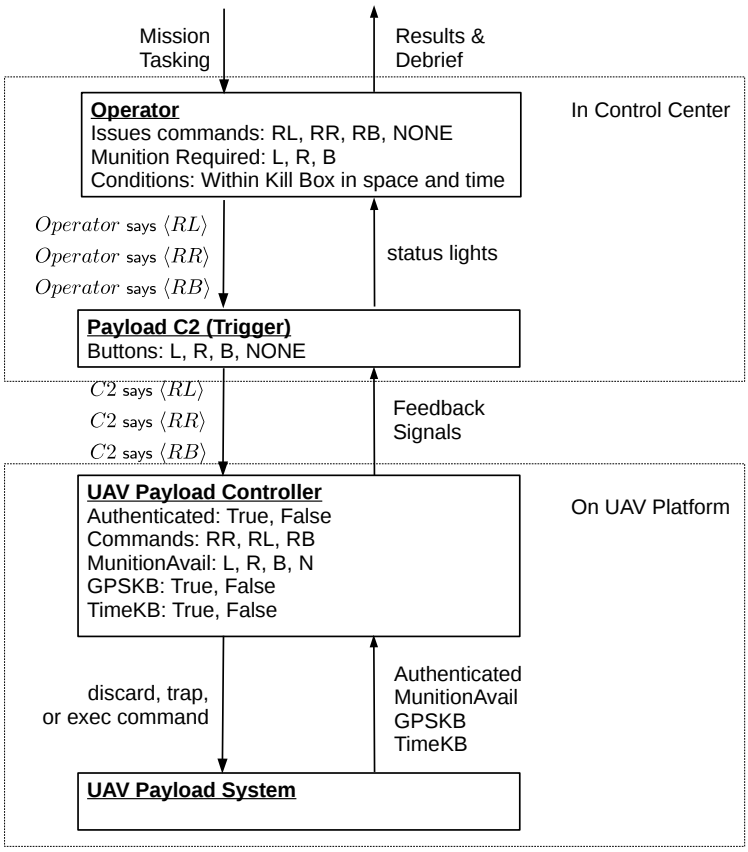


Figure 5.1: High-Level Functional Payload Control Structure for Air Interdiction Mission

> *degrade the enemy's fighting ability and limit the need for close air support (CAS) when the two forces meet in close combat."*
> – ANNEX 3003 COUNTERLAND OPERATIONS; AIR INTERDICTION, 17 March 2017

The example mission supported here is an unmanned aerial vehicle (UAV) flying within a closed, hot, established Blue Kill Box, where the following definitions apply

Kill Box: a three dimensional permissive fire support coordination measure with an associated airspace coordinating measure; lethal attack against surface targets allowed without further coordination

Blue Kill Box: air-to-surface fires effects are permitted

Hot: Fires or effects of fires are permitted without further coordination

Closed: manned aircraft are restricted from operating within kill box confines

With the above background in mind. the following assumptions are made:

- ▷ The GPS signal received and processed is a trusted source.
- ▷ The wireless command and control (C2) channel is trusted.
- ▷ Commands to and from the system do not suffer signal loss.
- ▷ The payload systems release as intended.
- ▷ The implementation implementation hardware is trusted.

Figure 5.2: STPA-Sec Process

Figure 5.1 shows the high-level functional control structure for an air interdiction mission using an unmanned aerial vehicle (UAV) with two pylons, left and right, carrying up to two weapons. At this high level, the CONOPS for payload control is as follows.

1. The *Operator*, acting on mission tasking, issues commands *RL,RR, RB* (release left, release right, release both).
2. The *Operator* receives feedback from the UAV sensors indicating if munitions are available, if the UAV is within the kill box, and if the UAV is within the time of the mission.
3. The command-and-control (C2) system relays the *Operator's* commands to the UAV *Payload Controller*.
4. The payload controller either discards, traps, or executes the instruction depending on if the command is authenticated, authorized, and the UAV is within the mission kill box in time and space. Unauthenticated commands are *discarded*. Authenticated but unauthorized commands are *trapped*. Authenticated and authorized commands are *executed*.

In the following sections, we define the mission, frame the security problem, state unacceptable losses, define a functional control structure, and state hazards and constraints.

5.2 Defining the Mission and Framing the Security Problem

A top-level block diagram of the STPA-Sec process is in Figure 5.2. The process starts by defining the mission and framing the security problem. Defining the mission starts by instantiating the *what, how, and why* in the following sentence.

> The **payload controller** is a system
>
> - to do {**what**},
> - by means of {**how**},
> - in order to contribute {**why**}.

We define our mission as follows:

> The **payload controller** is a system
>
> ▷ to **release a weapon within a kill box within mission timing**,
>
> ▷ by means of **transmitting, receiving and executing a valid release command**,
>
> ▷ in order to contribute to **accomplishing an air interdiction mission.**

5.3 Identify Unacceptable Losses

The next step is to define the unacceptable losses with respect to the system and the mission. A loss is defined as follows in the *STPA Handbook*, [25]:

ID	Unsafe Condition	Related Accidents
H-1	Payload released over non-target	A-1
H-2	Payload not released over legitimate target	A-2
H-3	Payload released outside of defined time window	A-1, A-3
H-4	Incorrect payload munitions dropped	A-1, A-2, A-3
H-5	Operator fails to assess collateral damage of drop before firing	A-1
H-6	Operator cannot confirm the effect of payload drop	A-1, A-2, A-3

Table 5.1: Payload Controller Hazard Specifications

ID	Safety Constraint	Related Hazard
SC0.1	Payload must not be released over non-target	H-1
SC0.2	Payload must be released over legitimate target	H-2
SC0.3	Payload must not be released outside of defined time window	H-3
SC0.4	Only specified payload munitions are to be dropped	H-4
SC0.5	Operator must correctly assess collateral damage of drop before firing	H-5
SC0.6	Operator must be able to confirm the effect of payload drop	H-6

Table 5.2: Payload Controller Safety Constraints

Definition: A *loss* involves something of value to stakeholders. Losses may include a loss of human life or human injury, property damage, environmental pollution, mission, reputation, or any other loss that is unacceptable to the stakeholders.

Unacceptable losses for the payload controller (due to its actions or inactions) result in the following:

A-1 Payload release causes unacceptable collateral damage

A-2 Payload release fails to achieve desired effects against target

A-3 Payload release destroys UAV

5.4 Identify System Hazards and Constraints

5.4.1 Hazards

Once unacceptable losses are identified, the paths to those losses are described in terms of *hazards* [25], where:

Definition: A *hazard* is a system state or set of conditions that, together with a particular set of worst-case environmental conditions, will lead to a loss.

Hazard specifications have three components:

1. System,
2. Unsafe condition, and
3. Associated losses linked to the hazard

Table 5.1 lists the six payload controller hazard specifications. Each hazard specification has an identifier, unsafe condition, and related accidents.

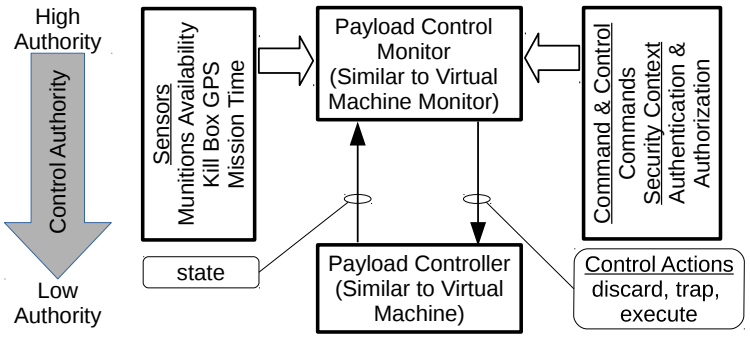


Figure 5.3: Control Structure for Payload Controller

5.4.2 Constraints

Once hazards are identified, *constraints* are identified by simply *inverting* the condition. The *STPA Handbook* [25] defines *constraints* as follows.

> **Definition:** A system-level constraint specifies system conditions or behaviors that need to be satisfied to prevent hazards (and ultimately prevent losses)

Constraints, similar to hazards, have three components:

1. System,
2. Condition to enforce, and
3. Associated hazards linked to constraint

Table 5.2 identifies system level constraints corresponding to the hazards identified in Table 5.1.

5.5 Create Functional Control Structure

The next step in the STPA-Sec process is to devise the Functional Control Structure. This is shown in Figure 5.3. Conceptually, the functional control structure is identical to that of a virtual machine monitor (VMM) controlling a virtual machine (VM), [29].

The Payload Control Monitor exercises the following *control actions*:

- ▷ Any C2 command that cannot be authenticated is *discarded* outright.
- ▷ Any C2 command that is authenticated but unauthorized (either because of of lack of authority or due to environmental conditions, e.g., outside the kill box) is *trapped*.
- ▷ Only C2 commands that are both authenticated and authorized are *executed*.

The control exercised by the Payload Control Monitor is consistent with classical systems theory, as stated by Checkland in page 87 of [7].

> Control is always associated with the imposition of constraints, and an account of a control process necessarily requires our taking into account at least two hierarchical levels.

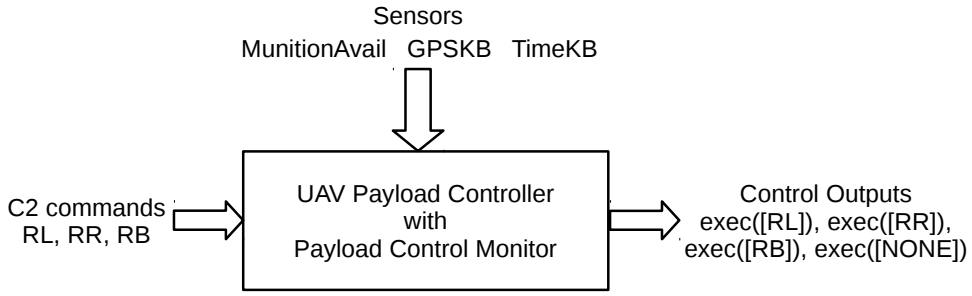


Figure 5.4: Payload Controller Block Diagram

5.6 Identify Hazardous Control Actions

The next step in the STPA-Sec process is to identify hazardous control actions. Until this point, we have not provided any details of the UAV. Now the details matter. The envisioned UAV can carry two payloads, one each on a *left pylon* and a *right pylon*. The UAV has three sensors:

Munitions Available sensor that reports on the munitions available in terms of whether or not each pylon is loaded or empty. If *both* pylons are loaded, the sensor reports B. If the left is loaded and the right is empty, the sensor reports L. If the right is loaded and the left is empty, the sensor reports R. If both pylons are empty, i.e., nothing is available, the sensor reports N.

GPSKB sensor that reports if the UAV is within the Kill Box or not, i.e., true (T) if within the Kill Box, or false (F) if not.

TimeKB sensor reports if the the UAV is within the mission timeframe or not, i.e., true (T) if so, and false (F) if not.

Based on the physical characteristics of the UAV, we have three commands that are requested by operators and relayed to the UAV via the C2 channel, all related to release of ordnance carried by the UAV's left and right pylons.

RL: release payload on left pylon

RR: release payload on right pylon

RB: release payload on both pylons

Figure 5.4 is a block diagram of the Payload Controller. Notice that the controller outputs include all of the hazardous control actions with one additional action:

NONE: essentially a NOP, or take no action.

The NONE action is used when it is appropriate for no action to be taken that can result in any kind of payload release.

Given the above commands *RL, RR,* and *RB* with the addition of *NONE*, we have ten control actions exercised by the Payload Control Monitor. These control actions are listed in Table 5.3.

The behavior of the Payload Controller in conjunction with the Payload Control Monitor is shown in Figure 5.5 as the secure-state machine uavSSM0. We go into more detail later as to what constitutes a secure-state machine (SSM). Essentially, SSMs support *complete mediation*, as defined by Saltzer and Schroeder [31].

Complete mediation: Every access to every object must be checked for authority.

Control Action	Constraints
discard command	input (command and sensor data) are dropped if authentication fails, where $command \in \{RL, RR, RB\}$
trap command	input (command and sensor data) are trapped if input is authenticated but authorization fails, where $command \in \{RL, RR, RB\}$
execute command	input (command and sensor data) are authenticated and authorized, where $command \in \{RL, RR, RB\}$
execute NONE	essentially a NOP, usually associated when an input is discarded or trapped

Table 5.3: Control Actions and Constraints

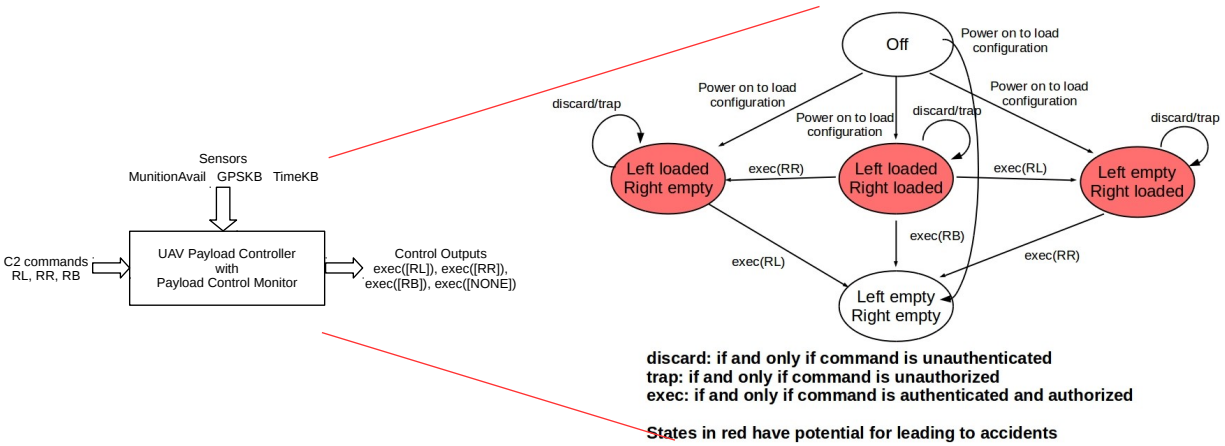


Figure 5.5: Top-Level UAV Secure State Machine uavSSM0

In the case of the Payload Controller, only the UAV Operator—within the Kill Box, mission timeframe, and munition availability—has the authority to access/deploy UAV payloads. Inputs and commands that fail authentication are discarded. Authenticated commands that fail authorization are trapped. In both these cases, no state change occurs.

When *RL, RR,* and *RB* commands are authenticated and authorized, they are executed. Authorization is intended to check not only if the UAV is within the Kill Box and mission timeframe, but also if there is consistency between the physical configuration of the UAV's payload and the state of the Payload Controller. If there are any discrepancies, the UAV may be faulty.

5.7 Generate Causal Scenarios

The next step in the STPA-Sec process is to generate causal scenarios leading to hazards. The *STPA Handbook* [25] defines loss scenarios as follows.

> **Definition:** A loss scenario describes the causal factors that can lead to the unsafe control actions and to hazards.

In other words, what has to happen (or not) in order for a hazard to occur? STPA-Sec uses a rigorous and structured approach to generating hazardous scenarios linked to control actions by asking, for each control action, what if the control action is

1. missing,
2. incorrectly applied,
3. occurs at the wrong time or the wrong order, or

Control Action	Not providing causes hazard	Providing incorrect causes hazard	Wrong timing or order causes hazard	Stopped too soon or applied too long
discard RL	Might allow unauthenticated command to be executed: H-1, H-2, H-3, H-4	Might cause failure to act on legitimate command: H-2, H-3	-	-
trap RL	Might allow unauthorized command to be executed: H-1, H-2, H-3, H-4	Might cause failure to act on legitimate command: H-2, H-3	-	-
execute RL	Failure to act on legitimate command: H-2, H-3	Might allow unauthorized command to be executed: H-1, H-2, H-3, H-4	If too late, UAV might exit Kill Box or exceed mission time: H-1, H-2, H-3	-

Table 5.4: Unsafe Control Actions Case Analysis

4. stopped too soon or applied too long (for continuous control actions, not discrete control actions)?

Answering the above questions are useful for informing possible scenarios leading to hazards.

For the case of the combined Payload Controller and Payload Control Monitor, answering the above questions leads to approximately 100 basic scenarios and associated safety constraints. Space limitations do not allow us to list them all. Nevertheless, the examples we cover below conceptually cover almost all the kinds of hazards contained in the basic scenarios.

Analyzing Unsafe RL Control Actions As a representative example, and to keep things simple, we focus on the control actions applied to the *RL* (release left payload) command. Table 5.4 summarizes the unsafe control action case analysis for discarding, trapping, and executing *RL*. A case-by-case description is as follows.

discard RL

Not providing causes hazard In this case a *RL* command unauthenticated and *discard RL* is not ordered. This runs the risk of allowing *RL* to be executed, which leads to hazards H-1 through H-4.

Providing incorrectly causes hazard In this case *RL* is authenticated but discarded. This runs the risk of a legitimate *RL* command being discarded and leads to hazards H-2 and H-3.

trap RL

Not providing causes hazard In this case an authenticated *RL* should be trapped because it is unauthorized and *trap RL* is not ordered. This runs the risk of allowing *RL* to be executed, which leads to hazards H-1 through H-4.

Providing incorrectly causes hazard In this case *RL* is authenticated and authorized but trapped. This prevents execution of a legitimate *RL* and leads to hazards H-2 and H-3.

execute RL

Not providing causes hazard In this case *RL* is authenticated and authorized but *exec RL* fails to be issued. This prevents execution of a legitimate *RL* and leads to hazards H-2 and H-3.

Notice that all the cases for control actions being stopped too soon or applied too long are empty. Given that SSMs are state machines, the outputs of the Payload Controller are *exec RL* or *exec NONE* as specified by the next-output function. The properties of the next-state and next-output functions are verified using the CSBD portion of STORM.

5.8 Mitigations and Controls

The final step in STPA-Sec is to generate the Linear Temporal Logic formulas corresponding to the constraints and controls against hazards. These formulas are informed by the previous work generating causal scenarios.

SC2.82	Constraint	The discard command must be provided when Authenticated is False
SSR1.82	LTL Formula	$\Box((Authenticated = False) \supset (controlAction = discard))$
SC2.83	Constraint	If outside Kill Box then control action is trap
SSR1.83	LTL Formula	$\Box((GPSKB = False) \supset (controlAction = trap))$
SC2.124	Constraint	If inside Kill Box, within mission time, Munition Available = L, Authenticated, and Input = RL, then execute RL
SSR1.124	LTL Formula	$\Box(((TimeKB = True) \wedge (GPSKB = True) \wedge (MunitionAvail = L) \wedge (Authenticated = True) \wedge (Input = RL)) \supset (controlAction = exec(RL)))$

Table 5.5: Refined Safety Constraint Examples and Their LTL (Linear Temporal Logic) Formulas

Considering all the unacceptable outcomes, hazards, constraints, hazardous control actions, and causal scenarios, we devise precise and formal statements for mitigations and controls. Building from Tables 5.1, 5.2, 5.3, and 5.4, the Payload Controller control structure as shown in Figure 5.3 and the top-level UAV secure state machine in Figure 5.5, we get the refined safety constraint examples and their corresponding LTL (Linear Temporal Logic) formulas, as shown in Table 5.5. Note: the LTL formula $\Box\varphi$ means the formula φ holds *globally*, i.e., in all states.

At this point, we have gone through the STPA-Sec process and have developed the following:

Validated CONOPS in the form of functional control models, control structures, control actions, constraints, and behavioral requirements expressed as LTL formulas, all derived from mission goals and unacceptable losses. The CONOPS are refined further and verified to have the required behavioral properties using CSBD.

Chapter 6

Using CSBD to Verify a Payload Controller CONOPS

"Do not trust security to technology unless that technology is demonstrably trustworthy, and the absence of demonstrated compromise is absolutely not a demonstration of security."

– Roger Schell

The Certified Security by Design (CSBD) portion of STORM focuses on verifying the properties of CONOPS implementation descriptions. CSBD is parametric and scalable with a focus on *complete mediation*—a command is executed *if and only if* it is authenticated and authorized [31]. CSBD is well-suited for application-specific systems, systems using the IoT (Internet of Things) [9], and for mission assurance.

CSBD is intended for project engineers responsible for satisfying technical specifications and for program managers responsible for devising system requirements satisfying mission objectives. CSBD addresses persistent worries of engineers and program managers, such as:

- ▷ Have I missed something?
- ▷ Is my design sufficient to satisfy the requirements?
- ▷ How do I know something bad won't happen because of a design flaw?
- ▷ Will my design do the right thing in all cases?
- ▷ Is my design secure?

CSBD address the above concerns by providing: (1) conceptual models focused on primary concerns, (2) computer-assisted reasoning tools to manage problems of scale, repeatability, and correctness, and (3) libraries of theories and examples.

CSBD has four components.

1. A command and control (C2) calculus for justifying actions while accounting for authentication and authorization. The rules of the C2 calculus are the inference rules of an access-control logic implemented as a propositional modal logic with Kripke semantics [13].

2. Algebraic models of idealized cryptographic operations.

3. Transition systems described as secure state machines (SSMs) with parameters for authentication and authorization. Transition systems are defined using structural operational semantics [22].

4. The HOL (Higher Order Logic) theorem prover [18], where the above components are conservative extensions of HOL, which preserve HOL's logical soundness.

The CSBD portion of STORM takes the outputs of STPA-Sec in the form of CONOPS expressed as functional control models, control structures, control actions, constraints, and behavioral requirements expressed as LTL formulas, all derived from mission goals and unacceptable losses. The output of CSBD are secure state machines formally proved in HOL to have the required properties required by STPA-Sec.

The SSMs produced by CSBD are CONOPS formally verified to have the required properties specified by STPA-Sec. This is the foundation of credible claims of trustworthiness as required by System Security Engineering in Figure 3.1.

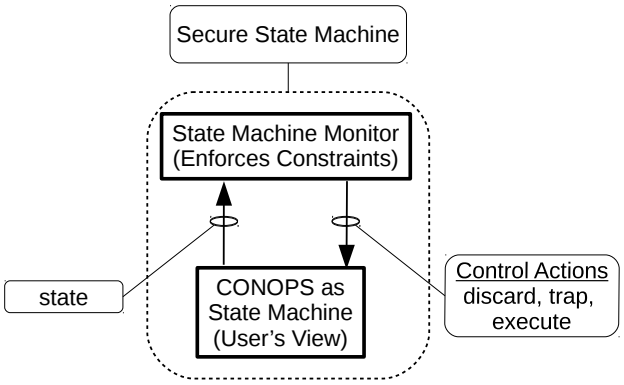


Figure 6.1: Secure State Machines and Their Components

The CONOPS produced by STPA-Sec satisfies the assurance case for acceptable security. The CONOPS formally verified by CSBD demonstrates the assurance case is satisfied.

6.1 Secure State Machines: A High Level Overview

Figure 6.1 is a high-level block diagram of SSMs. Notice that conceptually SSMs are consistent with STPA-Sec's hierarchical control structure. At the bottom is the user's view of the CONOPS expressed as a state machine (SM), where the number of states is either finite or infinite. The dynamic nature of CONOPS is modeled by changes in state. State changes are prompted by commands evaluated within the context of current state and the environment defined by sensor input values. A state machine monitor (SMM) enforces safety, security, and integrity constraints. We see later that these constraints are parameters of SSMs.

The SSMs we use for the Payload Controller have three control actions: discard, trap, and execute.

Discard: A SSM input is *discarded* if it fails authentication.

Trap: A SSM input is *trapped* if it is authenticated by fails authorization.

Execute: A SSM input is *executed* if it is authenticated and authorized.

Notice that this is consistent with the control actions, constraints, and unsafe control action case analysis developed using STPA-Sec in Tables 5.3 and 5.4.

6.1.1 Secure State Machine Structure

SSMs have a superset of state machine parameters. Typical state machines are characterized by five components:

1. A set of states $S = \{s_0, \cdots, s_{n-1}, \cdots\}$,
2. A set of inputs $I = \{i_0, i_1, \cdots, i_k, \cdots\}$,
3. A set of outputs $O = \{o_0, \cdots, o_j, \cdots\}$,
4. A next-state transition function $\delta : S \to I \to S$, and
5. An output function $\lambda : S \to I \to O$

Secure State Machine Inputs The inputs to SSMs are polymorphic, i.e., use type variables. For specific instances of SSMs, type variables are instantiated to the particular types used for the mission. Our general theory of SSMs supports parallel or concurrent inputs, e.g., commands or instructions from multiple sources with data coming from multiple sensors.

Payload Controller Inputs In the case of the Payload Controller, we have three sensors. These were listed in Section 5.6 with their associated outputs and interpretations.

MunitionAvail with outputs *R, L, B,* and *N*, indicating if the left, right, both, or neither pylons are carrying munitions.

GPSKB with outputs *T* or *F* indicating if the UAV is inside or outside the Kill Box.

TimeKB with outputs *T* or *F* indicating if the UAV is within the mission timeframe, or not.

In addition to the sensor inputs, we have the actions/commands requested by the UAV operator. These commands come via the C2 channel. The commands were described in Section 5.6 as:

RL: release payload on left pylon

RR: release payload on right pylon

RB: release payload on both pylons

Payload Controller Inputs Defined in HOL as Types Within the HOL theorem prover, we define the type *muniAvail* to be *R, L, B,* and *N*

```
muniAvail = N | L | R | B
```

The HOL type *ctrlAct* are the commands issued by the UAV operator and relayed by the C2 channel.

```
ctrlAct = RL | RR | RB
```

The values provided by sensors GPSKB and TimeKB are Booleans, which are defined in HOL. With all of this, we define the type *c3input* to be all the kinds of sensor and C2 inputs received by the UAV.

```
c3input = CMD ctrlAct | MA muniAvail | KBL bool | KBT bool
```

c3input is constructed from the types *muniAvail, ctrlAct,* and *bool*. The type constructors *CMD, MA, KBL,* and *KBT* map the C2 commands and sensor values from MunitionAvail, GPSKB, and TimeKB into the *c3input* type.

The combination of command and sensor inputs to the Payload Controller are modeled as *lists* of C2 and sensor statements coming from the C2 channel, and the MunitionAvail, GPSKB, and TimeKB sensors. Time is not an input value. SSM state-transition behavior is based on an underlying model of *clock periods*, i.e., inputs are sampled each clock period and state changes are made in response to the input and state values.

Modeling Values Coming From the C2 Channel and Sensors The property of **complete mediation**—statements are acted upon if and only if statements are authenticated to come from a known source or *principal* and that principal is either trusted or authorized on the statement—requires all commands and sensor inputs have the logical form and interpretation

Principal says ⟨*statement*⟩

Informally, principals are the *subjects and objects* in the system. Principals can be people, cryptographic keys, roles, certificate authorities, sensors, and C2 channels.

Payload Controller Principals The component types that constitute the overall type of principals for the Payload Controller include

staff: Alice, Bob, and Carol

role: Commander and Operator

authority: certificate authority identified by number

public keys: cryptographic keys for digital signatures for staff and certificate authorities

C2: the UAV C2 channel

MunitionAvail: the UAV sensor for available munitions

GPSKB: the UAV sensor indicating if the UAV is physically within the Kill Box

TimeKB: the UAV sensor indicating if the UAV is within the mission timeframe

In HOL, we principals and their components are algebraic types.

```
authority = ca num

principal =
    Staff staff
  | Authority authority
  | Role role
  | KeyS (staff pKey)
  | KeyA (authority pKey)
  | C2
  | MunitionAvail
  | GPSKB
  | TimeKB

role = Commander | Operator

staff = Alice | Bob | Carol
```

Payload Controller Statements The statements made by the Payload Controller principals are fully defined by the type *c3input*, with one exception: what if the sensor or the C2 channel says *nothing*? By *nothing* we include situations such that there is no command issued by the C2 channel or sensor, or the channel or sensor is disconnected. Essentially, what we need to do is add one more value to the *c3input* type corresponding to *nothing* or *nothing useful*.

This is easily done by the use of *option* types in HOL. The *option* type is defined in HOL as follows.

```
option = NONE | SOME 'a
```

where `'a` corresponds to a *type variable*, (all type variables start with a single quote. The properties of option types is given below by the theorem *option_CLAUSES*.

```
[option_CLAUSES]

⊢ (∀x y. (SOME x = SOME y) ⟺ (x = y)) ∧
  (∀x. THE (SOME x) = x) ∧ (∀x. NONE ≠ SOME x) ∧
  (∀x. SOME x ≠ NONE) ∧ (∀x. IS_SOME (SOME x) ⟺ T) ∧
  (IS_SOME NONE ⟺ F) ∧ (∀x. IS_NONE x ⟺ (x = NONE)) ∧
  (∀x. ¬IS_SOME x ⟺ (x = NONE)) ∧
  (∀x. IS_SOME x ⇒ (SOME (THE x) = x)) ∧
  (∀x. option_CASE x NONE SOME = x) ∧
  (∀x. option_CASE x x SOME = x) ∧
  (∀x. IS_NONE x ⇒ (option_CASE x e f = e)) ∧
  (∀x. IS_SOME x ⇒ (option_CASE x e f = f (THE x))) ∧
```

```
(∀x. IS_SOME x ⇒ (option_CASE x e SOME = x)) ∧
(∀v f. option_CASE NONE v f = v) ∧
(∀x v f. option_CASE (SOME x) v f = f x) ∧
(∀f x. OPTION_MAP f (SOME x) = SOME (f x)) ∧
(∀f. OPTION_MAP f NONE = NONE) ∧ (OPTION_JOIN NONE = NONE) ∧
∀x. OPTION_JOIN (SOME x) = x
```

To add the additional value `NONE` to the values in *c3input*, the type of C2 channel and sensor statements is *c3input option*. The C2 statements that can be made in *c3input option* are

- ▷ `SOME (CMD RL)`
- ▷ `SOME (CMD RR)`
- ▷ `SOME (CMD RB)`
- ▷ `NONE`

In the case of C2 statements, `NONE` denotes no C2 statement is made, because nothing was said, the channel is disconnected, etc.

Corresponding statements and interpretations hold for all the sensors. For example, for the GPSKB sensor that indicates if we are within the physical confines of the Kill Box, we have,

- ▷ `SOME (KBL T)`
- ▷ `SOME (KBL F)`
- ▷ `NONE`

Example Payload Controller Inputs An example of an input to the Payload Controller is

[Name C2 says prop (SOME (CMD RL))]; Name MunitionAvail says prop (SOME (MA L));
Name GPSKB says prop (Some (KBL T)); Name TimeKB says prop (SOME (KBT T))]

Informally, what the above input conveys is that the C2 channel is relaying a RL command, there is a munition available on the left pylon, and we are physically and temporally within the Kill Box and mission timeframe.

6.1.2 Secure State Machine Configurations

Secure state machines are described using *configurations*. Configurations of SSMs in HOL have the following form.

CFG *elementTest stateInterp context inputStream outputStream*, where

- ▷ *elementTest* is an authentication function mapped across each input element received.
- ▷ *stateInterp* is a function returning authorizations when applied to the state of a SSM and its input.
- ▷ *context* is a function returning authorizations when applied to a SSM's input.
- ▷ *inputStream* is a list of inputs, typically a *list* of input lists corresponding to SSM inputs received concurrently for each clock cycle.
- ▷ *state* is the current state of a SSM
- ▷ *outputStream* is a list of outputs

6.1.3 Secure State Machine Transitions

Rule-Based Behavior Defined Inductively

Inductive relations are a convenient means to describe rule-based behavior. For example, we define the set of *even* numbers with the following rules.

1. 0 is *even*.
2. If n is *even* then $n + 2$ is *even*.
3. *even* is the smallest set satisfying rules (1) and (2).

Defining the *even* relation inductively on natural numbers in HOL produces the following theorems automatically.

```
[even_rules]
 ⊢ even 0 ∧ ∀n. even n ⇒ even (n + 2)

[even_induction]
 ⊢ ∀even'.
     even' 0 ∧ (∀n. even' n ⇒ even' (n + 2)) ⇒
     ∀a0. even a0 ⇒ even' a0

[even_cases]
 ⊢ ∀a0. even a0 ⇔ (a0 = 0) ∨ ∃n. (a0 = n + 2) ∧ even n
```

The above theorems capture the three informally stated rules defining *even*. HOL theorem `even_rules` describes rules (1) and (2). The first conjunct of `even_rules` is rule (1). The second conjunct of `even_rules` is rule (2).

HOL theorem `even_induction` captures rule (3) because any numbers satisfying relation *even'* also satisfy *even*. HOL theorem `even_cases` states there are two forms of even numbers.

Secure State Machine Transition Rules Defined Inductively

Secure state machine behavior is described in HOL using *inductively* defined labeled transition relations among configurations. The transition labels correspond to the three kinds of *control actions* shown in Figure 6.1: *discard, trap,* and *execute*. Informally, we define the labeled transition relations $Config \xrightarrow{exec\ (inputList\ x)} Config_e$, $Config \xrightarrow{trap\ (inputList\ x)} Config_t$, and $Config \xrightarrow{discard\ (inputList\ x)} Config_d$ among configurations with the following rules.

Let *inputList x* be a list of commands and sensor data.

1. If *inputList x* is authenticated and authorized, then $Config \xrightarrow{exec\ (inputList\ x)} Config_e$, where $Config_e$ is the result of executing *inputList x*. The results of executing *inputList x* in current state *s* for given next-state and next-output functions *NS* and *Out* are

   Next state: *NS s (exec (inputList x))*
   Next output: *Out s (exec (inputList x))*

2. If *inputList x* is authenticated but not authorized, then $Config \xrightarrow{trap\ (inputList\ x)} Config_t$, where $Config_t$ is the result of trapping *inputList x*. The results of trapping *inputList x* in current state *s* are

   Next state: *NS s (trap (inputList x))*
   Next output: *Out s (trap (inputList x))*

3. If the input *inputList x* is not authenticated, then $Config \xrightarrow{discard\ (inputList\ x)} Config_d$, where $Config_d$ is the result of discarding *inputList x*. The results of discarding *inputList x* in current state *s* are

Next state: *NS s (discard (inputList x))*
Next output: *Out s (discard (inputList x))*

The above rules depend on authenticating and authorizing SSM inputs. This is described in below.

Input Authentication Recall that SSM inputs each clock cycle are given by a list of expressions of the form P says x, i.e., a statement x made by a principal P, where P is a sensor or source of instructions. Recall, too, the example Payload Controller input given earlier:

> [Name C2 says prop (SOME (CMD RL))]; Name MunitionAvail says prop (SOME (MA L));
> Name GPSKB says prop (Some (KBL T)); Name TimeKB says prop (SOME (KBT T))]

For authentication purposes, we want to know that for any given list of input values that *all* input values are authenticated to come from known sources. As the nature of inputs, statements, and principals varies from mission to mission, and application to application, we make the authentication function *elementTest* is a parameter, where *elementTest* applied to an input element, say `Name C2 says prop (SOME (CMD RL))` returns either `T` or `F`. If the result of applying *elementTest* to all input elements in the list that is the current input is `T` (true), then the input is authenticated. Otherwise, the input has failed to be authenticated.

The function defined in HOL for SSM mapping *elementTest* across all input elements and taking the and-reduction of the resulting list of Boolean values is *authenticationTest*, which take as arguments the function *elementTest* and a list of input elements x. It applies *elementTest* to all elements of x using `MAP` and then applies `FOLDR` $\lambda p\ q.\ p \wedge q$ `T` to the result, which is the Boolean value corresponding to the and-reduction of the list of Booleans.

```
[authenticationTest_def]
 ⊢ ∀ elementTest x.
     authenticationTest elementTest x ⟺
     FOLDR (λ p q. p ∧ q) T (MAP elementTest x)
```

Extracting Commands and Statements For authenticated inputs, we want to extract the commands and statements from the input list. Again, recall that all authenticated input elements are of the form P says x. What we want is to extract x from P says x. We define *extractCommand* to do this on input elements that match the expected form.

```
[extractCommand_def]
 ⊢ extractCommand (P says prop (SOME cmd)) = cmd
```

Given the definition of *extractCommand*, we define the function *inputList* to return a list of commands and statements from an authenticated list of input elements.

```
[inputList_def]
 ⊢ ∀ xs. inputList xs = MAP extractInput xs
```

Input Authorization Recall from Section 6.1.2 that SSM configurations included parameters *stateInterp* and *context* that returned authorizations when applied to the SSM state and input and SSM input, respectively. Authorizations based on state and input handle cases such as allowing the execution of privileged commands if the machine is in a superuser state. Authorizations based on context handle cases where people are acting in assigned roles with specific authority.

The function *CFGInterpret* evaluates whether or not a list of commands and sensor data given by *inputList* x are authorized or not based on *stateInterp* and *context*. If authorized, then *CFGInterpret* will determine *inputList* x is justified under the circumstances and should be executed. If unauthorized, *CFGInterpret* will determine that nothing (`NONE`) is justified, and *inputList* x should be trapped.

CFGInterpret is defined in Section 6.2.4 using an access-control logic with Kripke semantics and its associated C2 calculus, which we describe in more detail in Section 6.2. For now, it is sufficient to know that

the function *CFGInterpret* (M, O_i, O_s) applied to a configuration *Config* gives us the means to determine if *inputList* x should be executed or trapped, where (M, O_i, O_s) denotes the Kripke structure M and partial orders O_i, and O_s for integrity and security levels that are part of the Kripke semantics of the access-control logic described in Section 6.2.

SSM Transition Rules in HOL With the logical infrastructure of configurations, input lists, accessor functions, and interpretation functions for configurations, the transition behavior of SSMs is described in rule-based form as follows.

SSM behavior is defined inductively by three rules.

$$\textit{Execute} \quad \frac{(\texttt{authenticationTest}\ \textit{elementTest}\ x) \quad (\texttt{CFGInterpret}\ (M, O_i, O_s)\ \textit{Config})}{\textit{Config} \xrightarrow{\texttt{exec (inputList}\ x\texttt{)}} \textit{Config}_e}$$

$$\textit{Trap} \quad \frac{(\texttt{authenticationTest}\ \textit{elementTest}\ x) \quad (\texttt{CFGInterpret}\ (M, O_i, O_s)\ \textit{Config})}{\textit{Config} \xrightarrow{\texttt{trap (inputList}\ x\texttt{)}} \textit{Config}_t}$$

$$\textit{Discard} \quad \frac{\neg(\texttt{authenticationTest}\ \textit{elementTest}\ x)}{\textit{Config} \xrightarrow{\texttt{discard (inputList}\ x\texttt{)}} \textit{Config}_d}$$

where,

$$\begin{aligned}
\textit{Config} &= \texttt{CFG}\ \textit{elementTest}\ \textit{stateInterp}\ \textit{context}\ (x :: \textit{ins})\ s\ \textit{outs} \\
\textit{Config}_e &= \texttt{CFG}\ \textit{elementTest}\ \textit{stateInterp}\ \textit{context}\ \textit{ins} \\
&\qquad (NS\ s\ (\texttt{exec}\ (\texttt{inputList}\ x)))\ (\textit{Out}\ s\ (\texttt{exec}\ (\texttt{inputList}\ x)) :: \textit{outs}) \\
\textit{Config}_t &= \texttt{CFG}\ \textit{elementTest}\ \textit{stateInterp}\ \textit{context}\ \textit{ins} \\
&\qquad (NS\ s\ (\texttt{trap}\ (\texttt{inputList}\ x)))\ (\textit{Out}\ s\ (\texttt{trap}\ (\texttt{inputList}\ x)) :: \textit{outs}) \\
\textit{Config}_d &= \texttt{CFG}\ \textit{elementTest}\ \textit{stateInterp}\ \textit{context}\ \textit{ins} \\
&\qquad (NS\ s\ (\texttt{discard}\ (\texttt{inputList}\ x)))\ (\textit{Out}\ s\ (\texttt{discard}\ (\texttt{inputList}\ x)) :: \textit{outs})
\end{aligned}$$

The actual theorems in HOL are similar to the rule-based definitions shown above, with the following translation of notation.

$\xrightarrow{\texttt{exec (inputList}\ x\texttt{)}}$ is denoted by $\texttt{TR}\ (M, O_i, O_s)\ (\texttt{exec}\ (\texttt{inputList}\ x))$ in HOL, where (M, O_i, O_s) is due to the Kripke semantics of the access-control logic used to define `CFGInterpret`.

$\xrightarrow{\texttt{trap (inputList}\ x\texttt{)}}$ is denoted by $\texttt{TR}\ (M, O_i, O_s)\ (\texttt{trap}\ (\texttt{inputList}\ x))$ in HOL.

$\xrightarrow{\texttt{discard (inputList}\ x\texttt{)}}$ is denoted by $\texttt{TR}\ (M, O_i, O_s)\ (\texttt{discard}\ (\texttt{inputList}\ x))$ in HOL.

The HOL theorems `rule0, rule1`, and `rule2` corresponding to each of the inductively defined rules for *execute, trap,* and *discard* is described below.

The *execute* rule for SSMs is given by `rule0`. Note that the conditions above the horizontal line in a rule are conjoined together. Note, too, that the horizontal line in a rule separating the assumptions from the conclusion corresponds to an implication $\Rightarrow$ in HOL.

```
[rule0]
 ⊢ ∀ elementTest NS M Oi Os Out s context stateInterp x ins outs.
     authenticationTest elementTest x ∧
     CFGInterpret (M,Oi,Os)
```

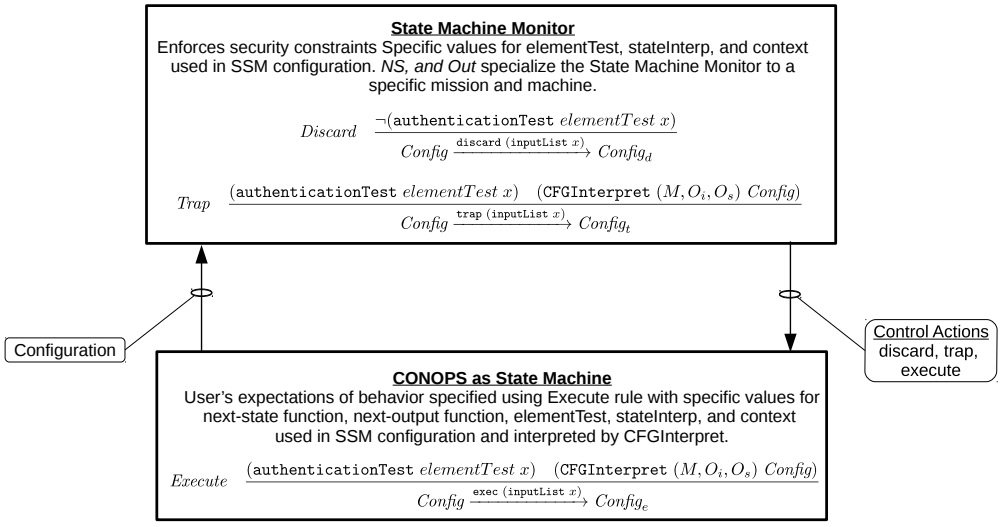


Figure 6.2: Refined Secure State Machine Descriptions and Their Components

```
  (CFG elementTest stateInterp context (x::ins) s outs) ⇒
TR (M,Oi,Os) (exec (inputList x))
  (CFG elementTest stateInterp context (x::ins) s outs)
  (CFG elementTest stateInterp context ins
     (NS s (exec (inputList x)))
     (Out s (exec (inputList x))::outs))
```

The *trap* rule for SSMs is given by `rule1`.

```
[rule1]
 ⊢ ∀ elementTest NS M Oi Os Out s context stateInterp x ins outs.
     authenticationTest elementTest x ∧
     CFGInterpret (M,Oi,Os)
       (CFG elementTest stateInterp context (x::ins) s outs) ⇒
     TR (M,Oi,Os) (trap (inputList x))
       (CFG elementTest stateInterp context (x::ins) s outs)
       (CFG elementTest stateInterp context ins
          (NS s (trap (inputList x)))
          (Out s (trap (inputList x))::outs))
```

The *discard* rule for SSMs is given by `rule2`.

```
[rule2]
 ⊢ ∀ elementTest NS M Oi Os Out s context stateInterp x ins outs.
     ¬authenticationTest elementTest x ⇒
     TR (M,Oi,Os) (discard (inputList x))
       (CFG elementTest stateInterp context (x::ins) s outs)
       (CFG elementTest stateInterp context ins
          (NS s (discard (inputList x)))
          (Out s (discard (inputList x))::outs))
```

Figure 6.2 refines the SSM block diagram in Figure 5.4. The transition relation $\xrightarrow{\texttt{exec (inputList } x\texttt{)}}$ in the *Execute* rule describes the user's expectations of machine behavior, and is part of the CONOPS as state machine. The *discard* and *trap* rules enforce constraints and are part of the state machine monitor.

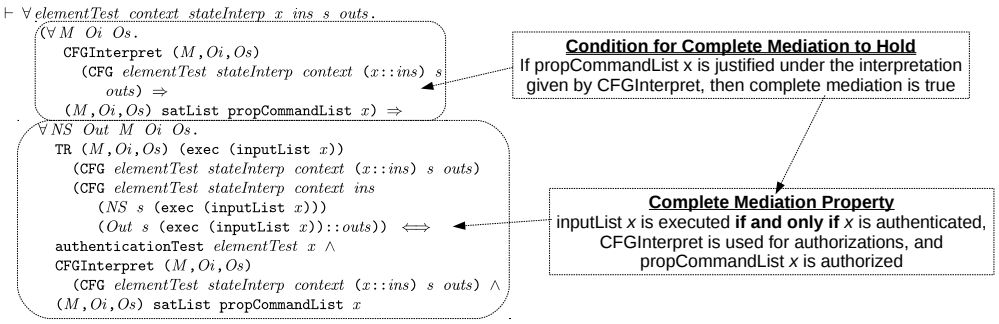


Figure 6.3: Execute Command Rule with Complete Mediation for Secure State Machines

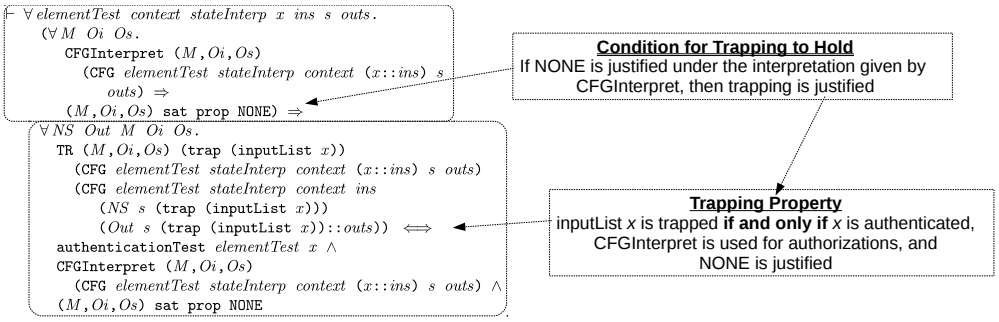


Figure 6.4: Trap Command Rule with Complete Mediation for Secure State Machines

6.1.4 Secure State Machine Complete Mediation Theorems

The important properties of secure state machine transition relations are shown in Figures 6.3, 6.4, and 6.5. Figure 6.3 shows the conditions under which complete mediation of instruction execution holds. If (M, O_i, O_s) `satList propCommandList` x is derivable from

$$\texttt{CFGInterpret}\ (M, O_i, O_s)\ (\texttt{CFG}\ elementTest\ stateInterp\ context\ (x :: ins)\ s\ outs),$$

where

```
[propCommandList_def]
 ⊢ ∀x. propCommandList x = MAP extractPropCommand x

[extractPropCommand_def]
 ⊢ extractPropCommand (P says prop (SOME cmd)) = prop (SOME cmd)
```

then for any and all next-state and next-output functions NS and Out, Kripke structures M, and integrity and security partial orderings O_i and O_s, `inputList` x is executed if and only if `inputList` x is authenticated and authorized.

Figure 6.4 shows the conditions under which complete mediation of instruction trapping holds. If (M, O_i, O_s) `sat prop NONE` is derivable from

$$\texttt{CFGInterpret}\ (M, O_i, O_s)\ (\texttt{CFG}\ elementTest\ stateInterp\ context\ (x :: ins)\ s\ outs),$$

then for any and all next-state and next-output functions NS and Out, Kripke structures M, and integrity and security partial orderings O_i and O_s, `inputList` x is trapped if and only if `inputList` x is authenticated and `NONE` is authorized.

Figure 6.5 shows that instructions are discarded if and only if they fail to be authenticated. Notice discards depend only on failed authentication and do not involve authorization at all.

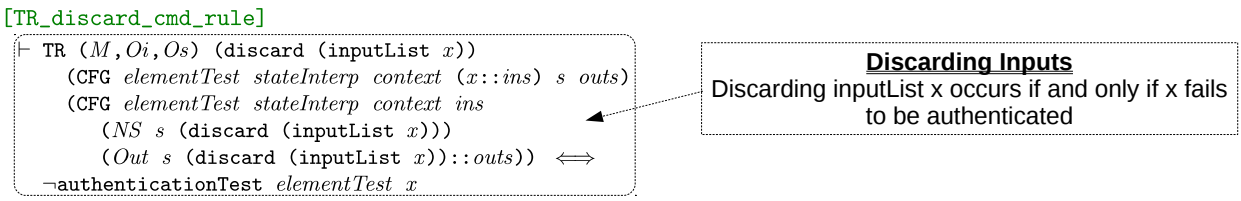


Figure 6.5: Discard Command Rule for Secure State Machines

CONOPS Statement	Formula
If φ_1 is true then φ_2 is true (typical of policy statements)	$\varphi_1 \supset \varphi_2$
Key associated with Alice	$K_a \Rightarrow Alice$
Bob has jurisdiction (controls or is believed) over statement φ	Bob controls φ
Alice and Bob together say φ	$(Alice \,\&\, Bob)$ says φ
Alice quotes Bob as saying φ	$(Alice \mid Bob)$ says φ
Bob is Alice's delegate on statement φ	Bob reps $Alice$ on φ
Carol is authorized in Role on statement φ	$Carol$ reps $Role$ on φ
Carol acting in Role makes statement φ	$(Carol \mid Role)$ says φ

Table 6.1: CONOPS Statements and Their Representation in the Access-Control Logic

`Theorem TR_discard_cmd_rule` is proof that Secure State Machines satisfy Refined Safety Constraint SC2.82 and its associated LTL formula SSR1.82 in Table 5.5.

6.2 C2 Calculus Overview

We have defined everything except `CFGInterpret` and the reasons for using Kripke structures M along with partial orders O_i and O_s. To understand the definition of `CFGInterpret`, we need to summarize the syntax, semantics, and inference rules of an access-control logic whose inference rules constitute a C2 (command and control) calculus.

The access-control logic is a propositional modal logic with Kripke semantics whose purpose is to describe and reason about authentication, authority, statements, delegations, policies, and trust assumptions [9] [13]. The access-control logic is our means to specify and verify security properties integrated with SSM transitions. This is done by `CFGInterpret`, which maps every SSM configuration into access-control logic formulas. By so doing, we are able to determine if a SSM input should be discarded, trapped, or executed.

6.2.1 Access-Control Logic Syntax

Recall that the access-control logic has principals as subjects and objects, and statements made by principals in the form of logical formulas. The syntax of principal expressions *Princ* is as follows.

$$\textbf{Princ} \quad ::= \quad \textbf{PName} \,/\, \textbf{Princ} \,\&\, \textbf{Princ} \,/\, \textbf{Princ} \mid \textbf{Princ}$$

" & " is pronounced "with"; " | " is pronounced "quoting". The type of principal expressions is composed of principal names, e.g., Alice, cryptographic keys, and userid with passwords. Compound expressions are created with & and | .

The syntax of logical formulas in the logic is defined as follows.

$$\begin{aligned} \textbf{Form} \quad ::= \quad & \textbf{PropVar} \,/\, \neg\, \textbf{Form} \,/ \\ & (\textbf{Form} \vee \textbf{Form}) \,/\, (\textbf{Form} \wedge \textbf{Form}) \,/ \\ & (\textbf{Form} \supset \textbf{Form}) \,/\, (\textbf{Form} \equiv \textbf{Form}) \,/ \\ & (\textbf{Princ} \Rightarrow \textbf{Princ}) \,/\, (\textbf{Princ}\ \textsf{says}\ \textbf{Form}) \,/ \\ & (\textbf{Princ}\ \textsf{controls}\ \textbf{Form}) \,/\, \textbf{Princ}\ \textsf{reps}\ \textbf{Princ}\ \textsf{on}\ \textbf{Form} \end{aligned}$$

$$
\begin{array}{rcl}
\mathcal{E}_{\mathcal{M}}[\![p]\!] & = & I(p) \\
\mathcal{E}_{\mathcal{M}}[\![\neg\varphi]\!] & = & W - \mathcal{E}_{\mathcal{M}}[\![\varphi]\!] \\
\mathcal{E}_{\mathcal{M}}[\![\varphi_1 \wedge \varphi_2]\!] & = & \mathcal{E}_{\mathcal{M}}[\![\varphi_1]\!] \cap \mathcal{E}_{\mathcal{M}}[\![\varphi_2]\!] \\
\mathcal{E}_{\mathcal{M}}[\![\varphi_1 \vee \varphi_2]\!] & = & \mathcal{E}_{\mathcal{M}}[\![\varphi_1]\!] \cup \mathcal{E}_{\mathcal{M}}[\![\varphi_2]\!] \\
\mathcal{E}_{\mathcal{M}}[\![\varphi_1 \supset \varphi_2]\!] & = & (W - \mathcal{E}_{\mathcal{M}}[\![\varphi_1]\!]) \cup \mathcal{E}_{\mathcal{M}}[\![\varphi_2]\!] \\
\mathcal{E}_{\mathcal{M}}[\![\varphi_1 \equiv \varphi_2]\!] & = & \mathcal{E}_{\mathcal{M}}[\![\varphi_1 \supset \varphi_2]\!] \cap \mathcal{E}_{\mathcal{M}}[\![\varphi_2 \supset \varphi_1]\!] \\
\mathcal{E}_{\mathcal{M}}[\![P \Rightarrow Q]\!] & = & \begin{cases} W, & \text{if } \hat{J}(Q) \subseteq \hat{J}(P) \\ \emptyset, & \text{otherwise} \end{cases} \\
\mathcal{E}_{\mathcal{M}}[\![P \text{ says } \varphi]\!] & = & \{w | \hat{J}(P)(w) \subseteq \mathcal{E}_{\mathcal{M}}[\![\varphi]\!]\} \\
\mathcal{E}_{\mathcal{M}}[\![P \text{ controls } \varphi]\!] & = & \mathcal{E}_{\mathcal{M}}[\![(P \text{ says } \varphi) \supset \varphi]\!] \\
\mathcal{E}_{\mathcal{M}}[\![P \text{ reps } Q \text{ on } \varphi]\!] & = & \mathcal{E}_{\mathcal{M}}[\![(P \mid Q \text{ says } \varphi) \supset Q \text{ says } \varphi]\!]
\end{array}
$$

Figure 6.6: Kripke Semantics of Access-Control Logic Formulas

Table 6.1 is a translation of sample CONOPS statements into formulas in the access-control logic.

6.2.2 Access-Control Logic Semantics

The semantics of the access-control logic uses *Kripke structures*. A **Kripke structure** $\mathcal{M}$ is a three-tuple $\langle W, I, J \rangle$, where:

- W is a nonempty set, whose elements are called *worlds*.
- $I : \mathbf{PropVar} \to \mathcal{P}(W)$ is an *interpretation* function that maps each propositional variable p to a set of worlds.
- $J : \mathbf{PName} \to \mathcal{P}(W \times W)$ is a function that maps each principal name A into a relation on worlds (i.e., a subset of $W \times W$).

The semantics of principal expressions *Princ* involves J and its extension $\hat{J}$. We define the extended function $\hat{J} : \mathbf{Princ} \to \mathcal{P}(W \times W)$ inductively on the structure of principal expressions, where $A \in \mathbf{PName}$.

$$
\begin{array}{rcl}
\hat{J}(A) & = & J(A) \\
\hat{J}(P \,\&\, Q) & = & \hat{J}(P) \cup \hat{J}(Q) \\
\hat{J}(P \mid Q) & = & \hat{J}(P) \circ \hat{J}(Q).
\end{array}
$$

Note: $R_1 \circ R_2 = \{(x, z) \mid \exists y.(x, y) \in R_1 \text{ and } (y, z) \in R_2\}$.

Each Kripke structure $\mathcal{M} = \langle W, I, J \rangle$ gives rise to a **semantic function**

$$\mathcal{E}_{\mathcal{M}}[\![-]\!] : \mathbf{Form} \to \mathcal{P}(W),$$

where $\mathcal{E}_{\mathcal{M}}[\![\varphi]\!]$ is the set of worlds in which φ is considered true.

$\mathcal{E}_{\mathcal{M}}[\![\varphi]\!]$ is defined inductively on the structure of φ, as shown in Figure 6.6. Note, in the definition of $\mathcal{E}_{\mathcal{M}}[\![P \text{ says } \varphi]\!]$, that $\hat{J}(P)(w)$ is simply the image of world w under the relation $\hat{J}(P)$.

6.2.3 The C2 Calculus—Access-Control Logic Inference Rules

An inference rule in the C2 calculus has the form

$$\frac{H_1 \quad \cdots \quad H_k}{C,}$$

where $H_1 \cdots H_k$ is a (possibly empty) set of *hypotheses* expressed as access-control logic formulas, and C is the *conclusion*, also expressed as an access-control logic formula. Whenever all of the hypotheses in an

$$P \text{ controls } \varphi \stackrel{\text{def}}{=} (P \text{ says } \varphi) \supset \varphi \qquad P \text{ reps } Q \text{ on } \varphi \stackrel{\text{def}}{=} P \mid Q \text{ says } \varphi \supset Q \text{ says } \varphi$$

$$\textit{Modus Ponens}\ \frac{\varphi \quad \varphi \supset \varphi'}{\varphi'} \qquad \textit{Says}\ \frac{\varphi}{P \text{ says } \varphi} \qquad \textit{Controls}\ \frac{P \text{ controls } \varphi \quad P \text{ says } \varphi}{\varphi}$$

$$\textit{Derived Speaks For}\ \frac{P \Rightarrow Q \quad P \text{ says } \varphi}{Q \text{ says } \varphi} \qquad \textit{Reps}\ \frac{Q \text{ controls } \varphi \quad P \text{ reps } Q \text{ on } \varphi \quad P \mid Q \text{ says } \varphi}{\varphi}$$

$$\textit{\& Says (1)}\ \frac{P \,\&\, Q \text{ says } \varphi}{P \text{ says } \varphi \wedge Q \text{ says } \varphi} \qquad \textit{\& Says (2)}\ \frac{P \text{ says } \varphi \wedge Q \text{ says } \varphi}{P \,\&\, Q \text{ says } \varphi}$$

$$\textit{Quoting (1)}\ \frac{P \mid Q \text{ says } \varphi}{P \text{ says } Q \text{ says } \varphi} \qquad \textit{Quoting (2)}\ \frac{P \text{ says } Q \text{ says } \varphi}{P \mid Q \text{ says } \varphi}$$

$$\textit{Idempotency of} \Rightarrow\ \frac{}{P \Rightarrow P} \qquad \textit{Monotonicity of} \Rightarrow\ \frac{P' \Rightarrow P \quad Q' \Rightarrow Q}{P' \mid Q' \Rightarrow P \mid Q}$$

Figure 6.7: Inference rules for the access-control logic

Access-Control Logic Formula	**HOL Syntax**
$\langle jump \rangle$	`prop jump`
$\neg\langle jump \rangle$	`notf (prop jump)`
$\langle run \rangle \wedge \langle jump \rangle$	`prop run andf prop jump`
$\langle run \rangle \vee \langle stop \rangle$	`prop run orf prop stop`
$\langle run \rangle \supset \langle jump \rangle$	`prop run impf prop jump`
$\langle walk \rangle \equiv \langle stop \rangle$	`prop walk eqf prop stop`
Alice says $\langle jump \rangle$	`Name Alice says prop jump`
Alice & *Bob* says $\langle stop \rangle$	`Name Alice meet Name Bob says prop stop`
Bob | *Carol* says $\langle run \rangle$	`Name Bob quoting Name Carol says prop run`
Bob controls $\langle walk \rangle$	`Name Bob controls prop walk`
Bob reps *Alice* on $\langle jump \rangle$	`reps (Name Bob) (Name Alice) (prop jump)`
Carol $\Rightarrow$ *Bob*	`Name Carol` speaks_for `Name Bob`

Table 6.2: CONOPS Formulas and Their Representation in HOL

inference rule are present in a proof, then the rule states it is permissible to include the conclusion in the proof, too.

The meaning of *sound* depends on the the definition of *satisfies* in the access-control logic. A Kripke structure $\mathcal{M}$ **satisfies** a formula φ when $\mathcal{E}_{\mathcal{M}}[\![\varphi]\!] = W$, i.e., φ is true in all worlds W of $\mathcal{M}$. We denote $\mathcal{M}$ satisfies φ by $\mathcal{M} \models \varphi$.

A C2 calculus inference rule is **sound** if, for all Kripke structures $\mathcal{M}$, whenever $\mathcal{M}$ satisfies all the hypotheses $H_1 \cdots H_k$, then $\mathcal{M}$ also satisfies C, i.e., if for all $\mathcal{M}$: $\mathcal{M} \models H_i$ for $1 \leq i \leq k$, then it must be the case that $\mathcal{M} \models C$.

All the inference rules presented here and in [13] are proved to be logically sound. Figure 6.7 are the core inference rules of the access-control logic.

6.2.4 The Access-Control Logic and C2 Calculus in HOL

Syntax and Semantics

The syntax of access-control logic formulas and principals are defined as the HOL types *Form* and *Princ* in *aclFoundation Theory* in Appendix A.1. The semantic function $\mathcal{E}_{\mathcal{M}}[\![-]\!]$ defined in Figure 6.6 is defined in HOL as the function *Efn*. Its definition, *Efn_def*, is part of *aclsemantics Theory*, which is in Appendix A.2.

Table 6.2 shows examples of access-control logic formulas and their representation in HOL.

Inference Rules

Recall in Section 6.2.3 that $\mathcal{M} \models \varphi$ denoted $\mathcal{E}_{\mathcal{M}}[\![\varphi]\!] = W$, i.e., φ is true for all worlds in $\mathcal{M}$. Inference rules are sound if $\mathcal{M}$ satisfies all the hypotheses $H_1 \cdots H_k$, and satisfies the conclusion C as well.

In our HOL implementation, we say Kripke structure M with partial orders O_i and O_s on integrity and security labels, respectively, satisfies an access-control logic formula f whenever the HOL semantic function `Efn`, applied to M, O_i, O_s, and f equals the universe of worlds in M. The definition of `sat` in HOL is as follows.

[sat_def]
$\vdash \forall M\ Oi\ Os\ f.$ `(`M`,`Oi`,`$Os`) sat` $f \iff$ `(Efn` Oi Os M f `= U(:'world))`

(M, O_i, O_s) satisfies a formula f if and only if the set of worlds in which f is true is the universe, i.e., the set containing all worlds.

An inference rule in the C2 calculus of the form

$$\frac{H_1 \cdots H_k}{C}$$

has a corresponding theorem in HOL

$$\vdash \forall M\ O_i\ O_s.(M, O_i, O_s)\ \texttt{sat}\ H_1 \Rightarrow \cdots \Rightarrow (M, O_i, O_s)\ \texttt{sat}\ H_k \Rightarrow (M, O_i, O_s)\ \texttt{sat}\ C,$$

where $\Rightarrow$ corresponds to logical implication in HOL.

Definition of `CFGInterpret` in HOL

Given the details of the access-control logic as implemented in HOL, we define the Secure State Machine (SSM) configuration interpretation function, `CFGInterpret`, as follows.

[CFGInterpret_def]
$\vdash$ `CFGInterpret (`M`,`Oi`,`$Os`)`
`(CFG` *elementTest stateInterp context* `(`x`::`*ins*`)` *state*
outStream`)` $\iff$
`(`M`,`Oi`,`$Os`) satList` *context* $x \wedge$ `(`M`,`Oi`,`$Os`) satList` $x \wedge$
`(`M`,`Oi`,`$Os`) satList` *stateInterp state* x

`CFGInterpret` is defined using `satList`, whose definition and key properties are below.

[satList_def]
$\vdash \forall M\ Oi\ Os$ *formList*.
`(`M`,`Oi`,`$Os`) satList` *formList* $\iff$
`FOLDR (`$\lambda x\ y.\ x \wedge y$`) T (MAP (`$\lambda f.$ `(`M`,`Oi`,`$Os`) sat` f`)` *formList*`)`

[satList_conj]
$\vdash \forall l_1\ l_2\ M\ Oi\ Os.$
`(`M`,`Oi`,`$Os`) satList` $l_1 \wedge$ `(`M`,`Oi`,`$Os`) satList` $l_2 \iff$
`(`M`,`Oi`,`$Os`) satList (`$l_1$ `++` l_2`)`

[satList_CONS]
$\vdash \forall h\ t\ M\ Oi\ Os.$
`(`M`,`Oi`,`$Os`) satList (`$h$`::`$t$`)` $\iff$
`(`M`,`Oi`,`$Os`) sat` $h \wedge$ `(`M`,`Oi`,`$Os`) satList` t

[satList_nil]
$\vdash$ `(`M`,`Oi`,`$Os`) satList []`

Essentially, `CFGInterpret` is the conjunction of all the access-control logic formulas that are part of (1) the security *context*, (2) the input x, and (3) the state interpretation *stateInterp*.

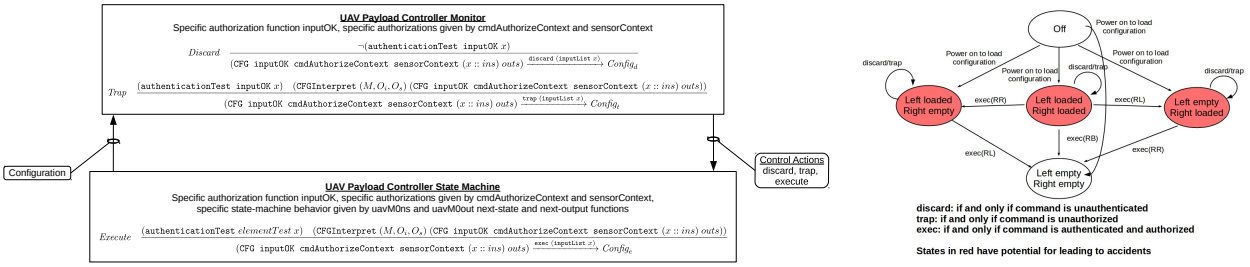


Figure 6.8: UAV Payload Controller CONOPS with Controls

6.3 UAV Payload Controller Definitions and Properties

The CONOPS for the UAV Payload Controller merges two views of the controller, as shown in Figure 6.8. The view developed by STPA-Sec is that of a Payload Controller regulated by a Payload Controller Monitor. All commands and sensor inputs are monitored and (1) discarded if unauthenticated, (2) trapped if authenticated but unauthorized, or (3) executed if authenticated and authorized.

The specific behavior of the Payload Controller is shown by the finite-state machine. It mirrors the physical state of the UAV where each Payload Controller state corresponds to the physical configuration of the UAV in terms of its pylons being loaded or empty. The controller's next-state and next-output functions reflect the physical requirements and consequences of executing the various payload release commands.

The outputs of this phase of the UAV Payload Controller design and verification process are (1) defining the Payload Controller's behavior by specializing SSM parameters for authentication, authorization, next-state, and next-output, and (2) verifying that the Payload Controller satisfies the required safety constraints produced by STPA-Sec, as exemplified by Table 5.5.

We define following functions, which collectively define the controller.

inputOK: the input authentication function

cmdAuthorizationContext: a function that returns authorizations appropriate for the controller's current state and inputs

sensorContext: a function that returns authorizations appropriate for current controller inputs in all states

uavM0ns: the next-state function of the Payload Controller

uavM0out: the next-output function of the Payload Controller

The HOL theory *uavTypes Theory* in Appendix B.4 defines the Payload Controller types.

```
c3input = CMD ctrlAct | MA muniAvail | KBL bool | KBT bool

ctrlAct = RL | RR | RB

muniAvail = N | L | R | B

state = Off | LlRe | LeRl | LlRl | LeRe
```

The Payload Controller inputs of type *c3input* are mediated by the Payload Control Monitor, and both are incorporated into a secure state machine. SSMs are defined by *ssm1 Theory* in Appendix B.1. The datatypes of *ssm1 Theory* are as follows.

```
configuration =
    CFG ((’command option, ’principal, ’d, ’e) Form -> bool)
        (’state ->
         (’command option, ’principal, ’d, ’e) Form list ->
         (’command option, ’principal, ’d, ’e) Form list)
```

```
        (('command option, 'principal, 'd, 'e) Form list ->
         ('command option, 'principal, 'd, 'e) Form list)
        (('command option, 'principal, 'd, 'e) Form list list)
        'state ('output list)

trType = discard 'cmdlist | trap 'cmdlist | exec 'cmdlist
```

trType models the use of control actions *discard, trap, and exec* on lists of *c3input* types. We use lists to model commands and sensor inputs received within a single clock cycle.

The *configuration* type defines SSM configurations. At this abstraction level, all parameters are defined in the access-control logic, where the underlying type variable `'command` is instantiated to be *c3input*. The underlying type of propositions in the access-control logic formulas is `c3input option`. Recall that *option types* add an additional element `NONE`, which we use in cases where nothing is present, returned, or useful.

The principals used by the Payload Controller are defined in *principal Theory* in Appendix B.3.

```
authority = ca num

principal =
    Staff staff
  | Authority authority
  | Role role
  | KeyS (staff pKey)
  | KeyA (authority pKey)
  | C2
  | MunitionAvail
  | GPSKB
  | TimeKB

role = Commander | Operator

staff = Alice | Bob | Carol
```

The first three parameters of SSM *configurations* are the functions `inputOK, cmdAuthorizationContext`, and `sensorContext`. The next three parameters are *state, inputs,* and *outputs*, i.e., the machine state, input stream, and output stream. The input stream to the controller is of type *c3input list list*, i.e., a list of *c3input lists*.

The definition of these three functions must satisfy the safety constraints derived by STPA-Sec as exemplified by Table 5.5, which are summarized as follows.

SC2.82 and SSR1.82: The *discard* command must be provided when Authenticated is False

SC2.83 and SSR1.83: If outside Kill Box then the control action is trap

SC2.124 and SSR1.124: If inside Kill Box, within mission time, Munition Available = L, Authenticated, and Input = RL, then *execute RL*

6.3.1 Authentication

The authentication function `inputOK` implements the requirement that all inputs, i.e., commands and sensor inputs, must come from their expected sources or else they are rejected. Figure 6.9 shows the ML source code that defines `inputOK`. The resulting full definition in HOL is quite lengthy and shown in its entirety as part of *uavSSM0 Theory* in Appendix B.6. As shown by the ML source code in Figure 6.9 and in *uavSSM0 Theory*, the only inputs that are authenticated are (1) commands from C2, (2) munition availability from MunitionAvail, (3) inside the Kill Box or not from GPSKB, and (4) within mission time limits or not from TimeKB. Every other possible input fails `inputOK`.

Figure 6.10 illustrates rejection of the command `prop (SOME (CMD RL))` coming from MunitionAvail.

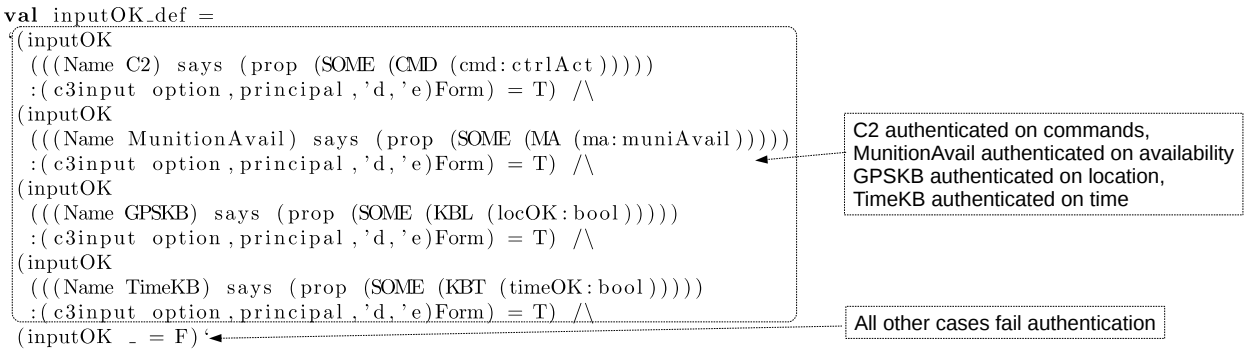


Figure 6.9: ML Source Code for inputOK Definition

6.3.2 Sensors are Trusted

Sensors are believed on the values they are designed to have in all controller states. The ML source code definition for the MunitionAvail sensor context, `maSensorContext` is shown in Figure 6.11. Note that it is believed on `MA load` values only, i.e., if the input is `((Name MunitionAvail) says prop(SOME(MA load)))` then the value returned is the authorization `((Name MunitionAvail) controls prop(SOME(MA load)))`. Every other case is ignored or returns the trivial assumption `TT`.

The definitions for the other two sensors GPSKB and TimeKB are similar. All the definitions are found in *uavSSM0 Theory* in Appendix B.6. The definition of `sensorContext` is the list of authorizations for each of the three sensors.

```
[sensorContext_def]
 ⊢ ∀x.
     sensorContext x =
     [maSensorContext x; gpskbSensorContext x;
      tkbSensorContext x]
```

6.3.3 Command Authorization

Authorizations for payload release commands depend on what command is ordered, the sensor values concurrent with the command, and the state of the Payload Controller. The function `cmdAuthorizationContext` shown in Figure 6.12, defined in *uavSSM0 Theory* in Appendix B.6 returns authorizations based on state and inputs. The definition of `cmdAuthorizationContext` is a series of if-then-else statements.

The logic behind the authorizations is as follows—and is reflected in the annotations in Figure 6.12.

1. If any sensor input is missing—regardless of state—then no action (`NONE`) is authorized.
2. If there is no command requested—regardless of state—then the sensor inputs values are returned as results.
3. If the current state is either `Off` or `LeRe` (both pylons are empty) then no action is authorized.
4. If both pylons are loaded (as confirmed by MunitionAvail) and the UAV is within the Kill Box boundaries in space and time, then `RB, RL`, or `RR` are authorized. Otherwise, no action is authorized.
5. If the left pylon is loaded and the right is empty (as confirmed by MunitionAvail) and the UAV is within the Kill Box boundaries in space and time, then `RL` is authorized. Otherwise, no action is authorized.
6. If the right pylon is loaded and the left is empty (as confirmed by MunitionAvail) and the UAV is within the Kill Box boundaries in space and time, then `RR` is authorized. Otherwise, no action is authorized.

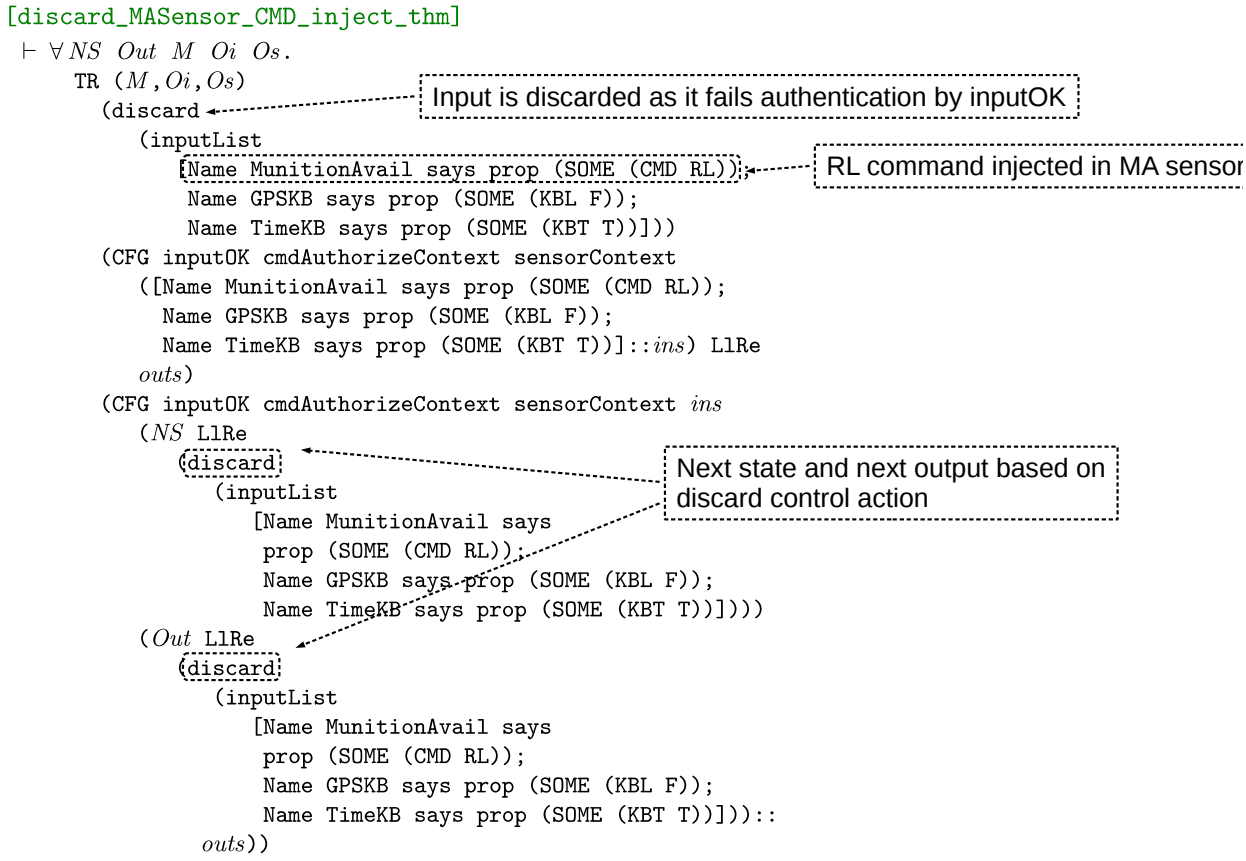


Figure 6.10: Injected Command on MunitionAvail Sensor Discarded

`cmdAuthorizationContext` depends on functions to gather commands and sensor values from the input x made by principals C2, MunitionAvail, GPSKB, and TimeKB. These functions, which work on inputs already authenticated by `inputOK`, start at the head of the list of authenticated statements. They return the first value found made by their associated principal and proceed no further. They return the statement made by their principal that is closest to the head of the input list, which can be thought of as the most recent statement made by their principal within the time period of the input list. If no relevant statement is found, then `NONE` is returned.

The ML source code function defining `getC2Statement` is shown in Figure 6.13 and is defined as part of *uavSSM0 Theory* in Appendix B.6. Its three conjuncts define `getC2Statement` in terms of the input list being empty, having a relevant input statement at the head of the list, or not.

The accessor functions `getMAStatement, getKBLStatement`, and `getKBTStatement` have definitions similar to `getC2Statement`. Their definitions are found in *uavSSM0 Theory* in Appendix B.6.

The authorizations for payload release are the authorization terms for each relevant Payload Controller state where munitions are available.

LlRl: `C2_LlRl_RB_Auth`, `C2_LlRl_RL_Auth`, `C2_LlRl_RR_Auth`

LlRe: `C2_LlRe_RL_Auth`

LeRL: `C2_LeRl_RR_Auth`

As an illustration of how each of the commands is authorized within the context of a particular state, the definition of `C2_LlRe_RL_Auth_def` is shown below. Authorization is granted if the MunitionAvail sensor reports that there is a payload on the left pylon and that the UAV is within the Kill Box in time and space.

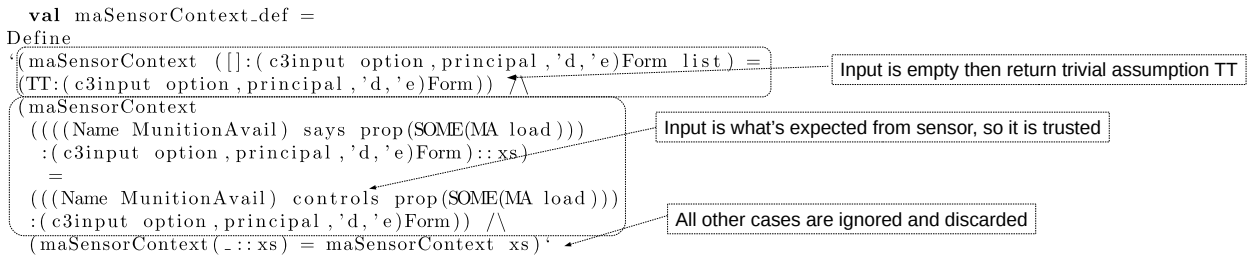


Figure 6.11: ML Source Code for maSensorContext

The other authorizations are defined similarly to `C2_LlRe_RL_Auth`. The authorizations are define as part of *uavSSM0 Theory* in Appendix B.6.

```
[C2_LlRe_RL_Auth_def]
 ⊢ C2_LlRe_RL_Auth =
   prop (SOME (MA L)) impf prop (SOME (KBL T)) impf
   prop (SOME (KBT T)) impf
   Name C2 controls prop (SOME (CMD RL))
```

6.3.4 Security Properties of Control Actions Separate from Next-State and Next-Output Behavior

Assuring Safety and Security Constraint for exec RL is Satisfied

Based on the definitions of `inputOK, cmdAuthorizeContext`, and `sensorContext`, we can prove theorems stating the actions corresponding to payload release , e.g., *exec RL*, occur if and only if the input is authenticated and the action is authorized. The theorem we show here as an example, addresses safety constraint SC2.124 and its associated LTL formula SSR1.124 in Table 5.5:

SC2.124: If inside Kill Box, within mission time, Munition Available = L, Authenticated, and Input = RL, then execute RL

SSR1.124: $\Box(((TimeKB = True) \wedge (GPSKB = True) \wedge (MunitionAvail = L) \wedge (Authenticated = True) \wedge (Input = RL)) \supset (controlAction = exec(RL)))$

Theorem `C2_LlRe_exec_RL_thm` shown in Figure 6.14, is proved as part of *uavSSM0 Theory* in Appendix B.6. Figure 6.14 annotates the theorem. The theorem states that the control action `exec [SOME (CMD RL); SOME (MA L); SOME (KBL T); SOME (KBT T)]` is taken if and only if the corresponding inputs are authenticated and authorized. The sensor values indicate that the UAV munitions state is consistent with that of the Payload Controller, and that the UAV is within the Kill Box in space and time.

Looking closely at `C2_LlRe_exec_RL_thm` shown in Figure 6.14 reveals that the theorem is true *for all* next-state and next-output functions. What the **theorem proves is that the Payload Control Monitor portion of the Payload Controller SSM satisfies safety and security properties in terms of generating control actions**. Generating control actions is determined by configuration authentication and authorization function specified by `inputOK, cmdAuthorizeContext`, and `sensorContext`.

The proof of the theorem is straightforward. Recall the `TR_exec_cmd_rule` in *ssm1 Theory* in Appendix B.1 and annotated in Figure 6.3. If `TR_exec_cmd_rule` is specialized to use `inputOK`, `cmdAuthorizeContext`, and `sensorContext` for authentication and authorization, then `C2_LlRe_exec_RL_thm` is the term that is the conclusion of an implication, where the hypothesis of the implication amounts to showing that executing the input values is justified under the access-control logic interpretation of the Payload Controller's configuration as given by `CFGInterpret`.

Theorem `C2_LlRe_exec_RL_lemma` states that if the configuration is in state `LlRe`, the input is `[Name C2 says prop (SOME (CMD RL)); Name MunitionAvail says prop (SOME (MA L)); Name GPSKB says prop`

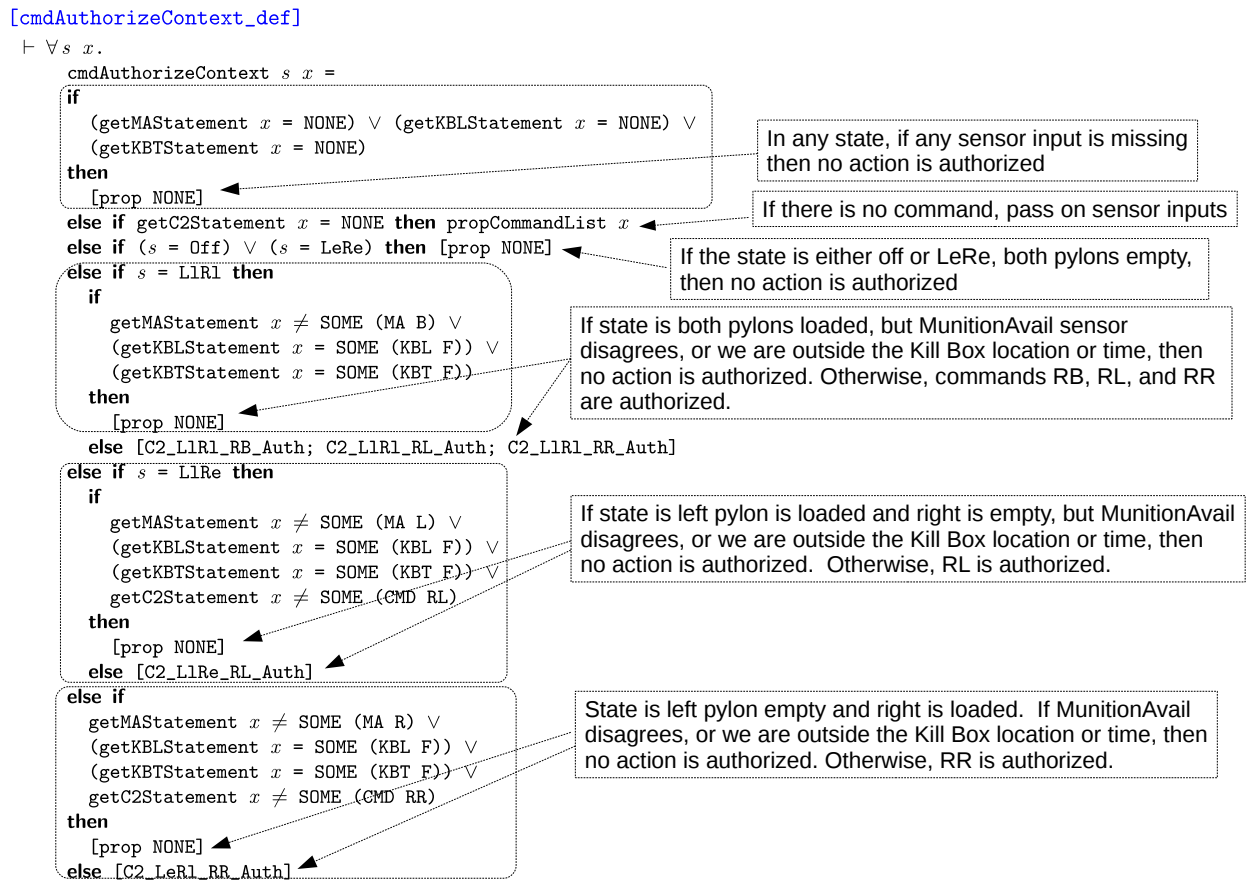


Figure 6.12: Command Authorization Based on State and Input

`(SOME (KBL T)); Name TimeKB says prop (SOME KBT T))]`, then executing `[prop (SOME (CMD RL));prop (SOME (MA L));prop (SOME (KBL T));prop (SOME KBT T))]` is justified, where the list of propositions is the result of applying `propCommandList` to the input.

```
[C2_LlRe_exec_RL_lemma]

⊢ ∀M Oi Os.
    CFGInterpret (M,Oi,Os)
      (CFG inputOK cmdAuthorizeContext sensorContext
         ([Name C2 says prop (SOME (CMD RL));
           Name MunitionAvail says prop (SOME (MA L));
           Name GPSKB says prop (SOME (KBL T));
           Name TimeKB says prop (SOME (KBT T))]::ins) LlRe
         outs) ⇒
    (M,Oi,Os) satList
    propCommandList
      [Name C2 says prop (SOME (CMD RL));
       Name MunitionAvail says prop (SOME (MA L));
       Name GPSKB says prop (SOME (KBL T));
       Name TimeKB says prop (SOME (KBT T))]
```

This theorem, combined with `TR_exec_cmd_rule`, gives rise to the `C2_LlRe_exec_RL_thm` in Figure 6.14.

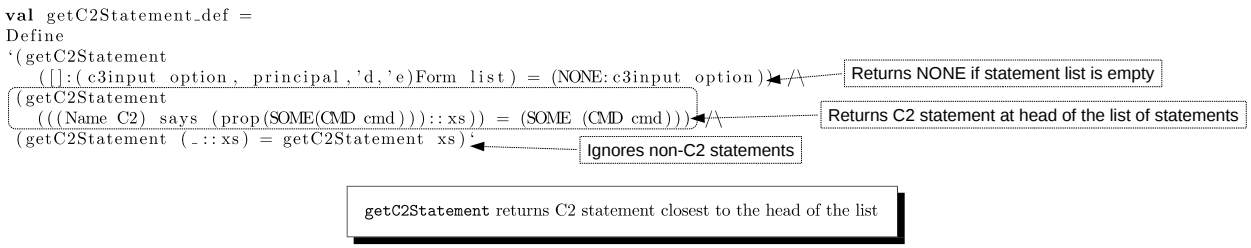


Figure 6.13: getC2Statement Definition in ML

Theorem `C2_LlRe_exec_RL_thm` proves that Safety Constraint SC2.124 and its associated LTL formula SSR1.124 are satisfied by a Secure State Machine equipped with `inputOK`, `cmdAuthorizeContext`, and `sensorContext`.

Assuring Safety and Security Constraint for trap RL is Satisfied

Theorem `C2_LlRe_trap_RL_KBL_F_justified_thm`, annotated in Figure 6.15 and proved in *uavSSM0 Theory* in Appendix B.5, addresses safety constraint SC1.83 and its associated LTL formula SSR1.83 in Table 5.5.

SC2.83: If outside Kill Box then the control action is trap RL

SSR1.83: $\Box((GPSKB = False) \supset (controlAction = trap))$

Looking at theorem `C2_LlRe_trap_RL_KBL_F_justified_thm`, annotated in Figure 6.15, the input `[Name C2 says prop (SOME (CMD RL)); Name MunitionAvail says prop (SOME (MA L)); Name GPSKB says prop (SOME (KBL F)); Name TimeKB says prop (SOME (KBT T))]` shows the GPSKB sensor indicating that the UAV is outside the Kill Box. The theorem also show that the input is trapped if and only if it is authenticated an no action is authorized.

Similar to theorem `C2_LlRe_exec_RL_thm`, `C2_LlRe_trap_RL_KBL_F_justified_thm` is true *for all* next-state and next-output functions. As before, it shows that the Payload Control Monitor portion of the Payload Controller SSM satisfies safety and security properties in terms of generating control actions, based on the authentication and authorization functions `inputOK, cmdAuthorizeContext`, and `sensorContext`.

The proof of theorem `C2_LlRe_trap_RL_KBL_F_justified_thm` uses `TR_trap_cmd_rule`, which is proved in *ssm1 Theory* in Appendix B.1, and shown with annotations in Figure 6.4. If `TR_trap_cmd_rule` is specialized to use `inputOK`, `cmdAuthorizeContext`, and `sensorContext` for authentication and authorization, then `C2_LlRe_trap_RL_KBL_F_justified_thm` is the term that is the conclusion of an implication, where the hypothesis of the implication amounts to showing that trapping the input values is justified under the access-control logic interpretation of the Payload Controller's configuration as given by `CFGInterpret`.

Theorem `C2_LlRe_trap_RL_KBL_F_lemma` states that if the configuration is in state `LlRe` and the input is `[Name C2 says prop (SOME (CMD RL)); Name MunitionAvail says prop (SOME (MA L)); Name GPSKB says prop (SOME (KBL F)); Name TimeKB says prop (SOME KBT T))]`, i.e., the UAV is outside the Kill Box, then no action is authorized.

```
[C2_LlRe_trap_RL_KBL_F_lemma]
 ⊢ ∀M Oi Os.
     CFGInterpret (M,Oi,Os)
       (CFG inputOK cmdAuthorizeContext sensorContext
          ([Name C2 says prop (SOME (CMD RL));
            Name MunitionAvail says prop (SOME (MA L));
            Name GPSKB says prop (SOME (KBL F));
            Name TimeKB says prop (SOME (KBT T))]::ins) LlRe
          outs) ⇒
     (M,Oi,Os) sat prop NONE
```

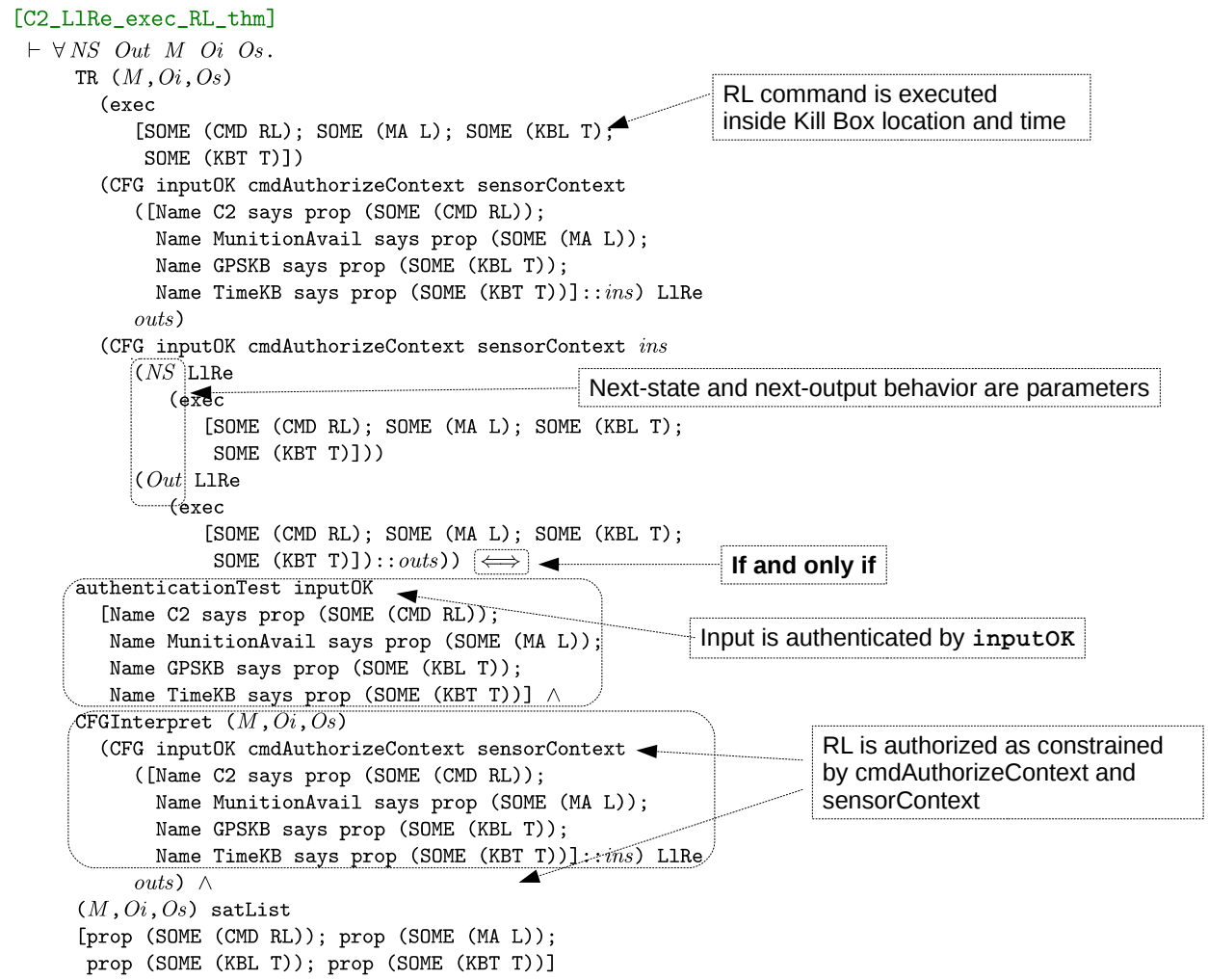


Figure 6.14: Execution of RL Command is Completely Mediated

This theorem, combined with `TR_trap_cmd_rule`, gives rise to the `C2_LlRe_trap_RL_KBL_F_justified_thm` in Figure 6.15.

Theorem `C2_LlRe_trap_RL_KBL_F_justified_thm` proves that Safety Constraint SC2.83 and its associated LTL formula SSR1.83 are satisfied by a Secure State Machine equipped with `inputOK`, `cmdAuthorizeContext`, and `sensorContext`.

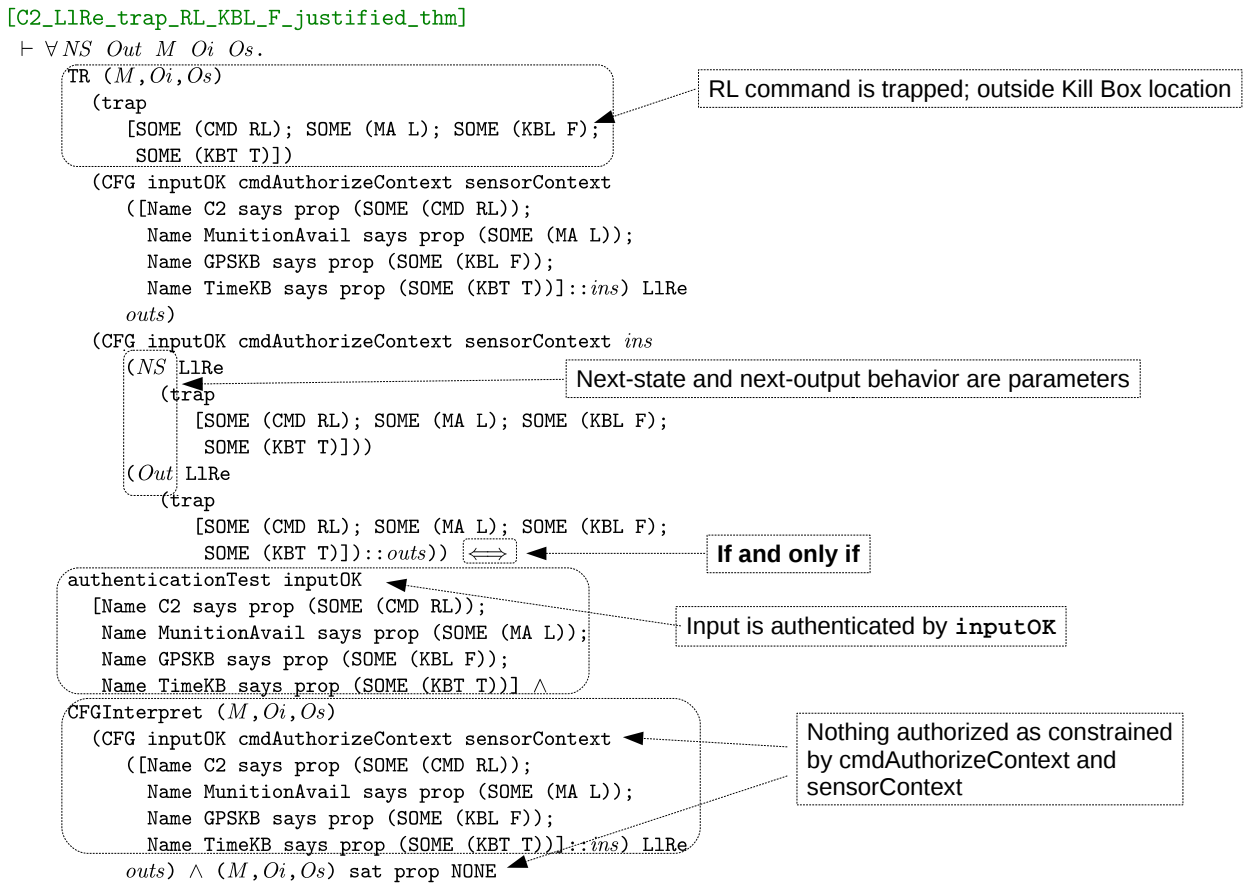


Figure 6.15: Trapping of RL Command Outside Kill Box is Completely Mediated

6.3.5 Definitions and Properties of UAV Next-State and Next-Output Functions

We now turn to the formal definitions of the Payload Controller's next-state and next-output functions, `uavMOns` and `uavMOout`. Both of these definitions appear in Figure 6.16 and are defined in *uavDef Theory* in Appendix B.5. Both functions depend on current state and control actions. Both are implemented as a series of if-then-else statements. Both functions follow the state machine shown in Figures 5.5 and 6.8.

The next-state function `uavMOns` is summarized as follows.

exec control actions: the states model the physical configuration of the UAV. After releasing a munition, the next state corresponds to the physical state of the UAV, e.g., if an RL command is executed in the LlRl state where both pylons have munitions, then the next state reflects emptying the left pylon, which is the state LeRl. No state change or action occurs unless the command is given within the sensor context indicating that the proper munition is available and the UAV is within the Kill Box time and location.

discard and trap control actions: no state change occurs in these cases.

The next-output function `uavMOout` is summarized as follows.

exec control actions: except for the `off` state, the next output is `exec [SOME (CMD action)]`. For the `off` state, the next output is `exec []`. In exactly the same manner as `uavMOns`, no release command is executed unless the command is given within the sensor context indicating that the proper munition is available and the UAV is within the Kill Box time and location.

```
[uavMOns_def]
 ⊢ (uavMOns Off (exec x) =
    if getMA x = SOME (MA N) then LeRe
    else if getMA x = SOME (MA L) then LlRe
    else if getMA x = SOME (MA R) then LeRl
    else if getMA x = SOME (MA B) then LlRl
    else Off) ∧
   (uavMOns LlRe (exec x) =
    if
      (getCMD x = SOME (CMD RL)) ∧ (getMA x = SOME (MA L)) ∧
      (getKBL x = SOME (KBL T)) ∧ (getKBT x = SOME (KBT T))
    then
      LeRe
    else LlRe) ∧
   (uavMOns LeRl (exec x) =
    if
      (getCMD x = SOME (CMD RR)) ∧ (getMA x = SOME (MA R)) ∧
      (getKBL x = SOME (KBL T)) ∧ (getKBT x = SOME (KBT T))
    then
      LeRe
    else LeRl) ∧
   (uavMOns LlRl (exec x) =
    if
      (getCMD x = SOME (CMD RL)) ∧ (getMA x = SOME (MA B)) ∧
      (getKBL x = SOME (KBL T)) ∧ (getKBT x = SOME (KBT T))
    then
      LeRl
    else if
      (getCMD x = SOME (CMD RR)) ∧ (getMA x = SOME (MA B)) ∧
      (getKBL x = SOME (KBL T)) ∧ (getKBT x = SOME (KBT T))
    then
      LlRe
    else if
      (getCMD x = SOME (CMD RB)) ∧ (getMA x = SOME (MA B)) ∧
      (getKBL x = SOME (KBL T)) ∧ (getKBT x = SOME (KBT T))
    then
      LeRe
    else LlRl) ∧ (uavMOns LeRe (exec v0) = LeRe) ∧
   (uavMOns s (trap v1) = s) ∧ (uavMOns s (discard v2) = s)
```

```
[uavMOout_def]
 ⊢ (uavMOout Off (exec x) = exec []) ∧
   (uavMOout LlRe (exec x) =
    if
      (getCMD x = SOME (CMD RL)) ∧ (getMA x = SOME (MA L)) ∧
      (getKBL x = SOME (KBL T)) ∧ (getKBT x = SOME (KBT T))
    then
      exec [SOME (CMD RL)]
    else exec [NONE]) ∧
   (uavMOout LeRl (exec x) =
    if
      (getCMD x = SOME (CMD RR)) ∧ (getMA x = SOME (MA R)) ∧
      (getKBL x = SOME (KBL T)) ∧ (getKBT x = SOME (KBT T))
    then
      exec [SOME (CMD RR)]
    else exec [NONE]) ∧
   (uavMOout LlRl (exec x) =
    if
      (getCMD x = SOME (CMD RL)) ∧ (getMA x = SOME (MA B)) ∧
      (getKBL x = SOME (KBL T)) ∧ (getKBT x = SOME (KBT T))
    then
      exec [SOME (CMD RL)]
    else if
      (getCMD x = SOME (CMD RR)) ∧ (getMA x = SOME (MA B)) ∧
      (getKBL x = SOME (KBL T)) ∧ (getKBT x = SOME (KBT T))
    then
      exec [SOME (CMD RR)]
    else if
      (getCMD x = SOME (CMD RB)) ∧ (getMA x = SOME (MA B)) ∧
      (getKBL x = SOME (KBL T)) ∧ (getKBT x = SOME (KBT T))
    then
      exec [SOME (CMD RB)]
    else exec [NONE]) ∧
   (uavMOout LeRe (exec v0) = exec [NONE]) ∧
   (uavMOout s (trap v1) = exec [NONE]) ∧
   (uavMOout s (discard v2) = exec [NONE])
```

Figure 6.16: UAV Payload Controller Next-State and Next-Output Functions

discard and trap control actions: the output is `exec [NONE]`.

The remaining sections give specific safety theorems for each control action, i.e., *discard, trap,* and *exec.*

discard Safety Theorems

The theorems `discard_out_safe_thm` and `discard_safe_thm` show that for all states and inputs no action is taken and no state change occurs.

```
[discard_out_safe_thm]
 ⊢ uavMOout s (discard x) = exec [NONE]
```

```
[discard_safe_thm]
 ⊢ ∀state x. uavMOns state (discard x) = state
```

trap Safety Theorems

The theorems `trap_out_safe_thm` and `trap_safe_thm` show that for all states and inputs no action is taken and no state change occurs.

```
[trap_out_safe_thm]
 ⊢ uavMOout s (trap x) = exec [NONE]
```

```
[trap_safe_thm]
 ⊢ ∀state x. uavMOns state (trap x) = state
```

exec Safety Theorems

The [exec_hca_out_thm] theorem is a case analysis for necessary conditions for execution of RL, RR, of RB payload release commands. In all cases, the state of the Payload Controller must match the MunitionAvail sensor input, and the time and location sensors must indicate the UAV is inside the Kill Box in both time and space.

[exec_hca_out_thm]

```
⊢ ∀hca s x.
    (uavMOout s (exec x) = exec [SOME (CMD hca)]) ⟺
    (hca = RL) ∧
    ((s = LlRe) ∧ (getCMD x = SOME (CMD RL)) ∧
     (getMA x = SOME (MA L)) ∧ (getKBL x = SOME (KBL T)) ∧
     (getKBT x = SOME (KBT T)) ∨
     (s = LlRl) ∧ (getCMD x = SOME (CMD RL)) ∧
     (getMA x = SOME (MA B)) ∧ (getKBL x = SOME (KBL T)) ∧
     (getKBT x = SOME (KBT T))) ∨
    (hca = RR) ∧
    ((s = LeRl) ∧ (getCMD x = SOME (CMD RR)) ∧
     (getMA x = SOME (MA R)) ∧ (getKBL x = SOME (KBL T)) ∧
     (getKBT x = SOME (KBT T)) ∨
     (s = LlRl) ∧ (getCMD x = SOME (CMD RR)) ∧
     (getMA x = SOME (MA B)) ∧ (getKBL x = SOME (KBL T)) ∧
     (getKBT x = SOME (KBT T))) ∨
    (hca = RB) ∧ (s = LlRl) ∧ (getCMD x = SOME (CMD RB)) ∧
    (getMA x = SOME (MA B)) ∧ (getKBL x = SOME (KBL T)) ∧
    (getKBT x = SOME (KBT T))
```

The [exec_state_change_thm] theorem is a case analysis for the necessary conditions for a state change due to executing a payload release command. The case analysis is similar to that of exec_hca_out_thm with the addition of the cases when the state is off and the value of the MunitionAvail sensor is not NONE.

[exec_state_change_thm]

```
⊢ ∀s x.
    s ≠ uavMOns s (exec x) ⟺
    ((s = Off) ∧ (getMA x = SOME (MA N)) ∨
     (s = Off) ∧ (getMA x = SOME (MA L)) ∨
     (s = Off) ∧ (getMA x = SOME (MA R)) ∨
     (s = Off) ∧ (getMA x = SOME (MA B))) ∨
    (s = LlRe) ∧ (getCMD x = SOME (CMD RL)) ∧
    (getMA x = SOME (MA L)) ∧ (getKBL x = SOME (KBL T)) ∧
    (getKBT x = SOME (KBT T)) ∨
    (s = LeRl) ∧ (getCMD x = SOME (CMD RR)) ∧
    (getMA x = SOME (MA R)) ∧ (getKBL x = SOME (KBL T)) ∧
    (getKBT x = SOME (KBT T)) ∨
    (s = LlRl) ∧ (getCMD x = SOME (CMD RL)) ∧
    (getMA x = SOME (MA B)) ∧ (getKBL x = SOME (KBL T)) ∧
    (getKBT x = SOME (KBT T)) ∨
    (s = LlRl) ∧ (getCMD x = SOME (CMD RR)) ∧
    (getMA x = SOME (MA B)) ∧ (getKBL x = SOME (KBL T)) ∧
    (getKBT x = SOME (KBT T)) ∨
    (s = LlRl) ∧ (getCMD x = SOME (CMD RB)) ∧
    (getMA x = SOME (MA B)) ∧ (getKBL x = SOME (KBL T)) ∧
    (getKBT x = SOME (KBT T))
```

Formally Verified CONOPS in the form of secure state machines implementing control structures, control actions, and constraints are described and formally verified in CSBD using Higher Order Logic. The verified CONOPS developed using CSBD is directly linked to the validated CONOPS developed by STPA-Sec. **The Systems Security Engineering Trustworthiness requirement is met by developing and demonstrating satisfaction of acceptable security**.

Chapter 7

Systems Security Engineering Education Using STORM

"There's no substitute for knowing what you're doing."
– A systems engineering aphorism

"[T]he single biggest impact I can make to the mission and my unit is to take care of my Airmen. This does not mean doing the job for them, but giving them what they need to get the job done."
– Senior Master Sgt. Claus Peris, 660th Aircraft Maintenance Squadron

Recall in Chapter 3 that Systems Security Engineering (SSE) is:

A discipline to achieve stakeholder objectives for the protection of assets, by means of
applying systems and security principles, analysis, and tools, in order to
produce outcomes that

1. prevent and control asset loss and associated consequences, and
2. substantiate security and trustworthiness claims using evidence-based reasoning.

If we rewrite the description of SSE in terms of an educational outcome, we get:

When given stakeholder objectives for the protection of assets,
students are able to apply systems and security principles, analyses, and tools,
to produce outcomes that

(1) prevent and control asset loss and associated consequences, and
(2) substantiate security and trustworthiness claims using evidence-based reasoning.

STORM, as a rigorous and formal methodology based on safety and security principles backed up with temporal, modal, and higher-order logic, satisfies the requirements of Systems Security Engineering. A reasonable question is, with a reasonable amount of effort and time, can students learn STORM and apply it successfully? The answer is, "yes!"

All the elements of STORM discussed here, have been taught within the Air Force, used by undergraduate-level research interns at the Air Force Research Laboratory's ACE Cybersecurity Boot Camp, and taught routinely as part of Syracuse University's undergraduate and graduate programs in computer science, computer engineering, and cybersecurity.

There is a substantial body of educational resources in support of STORM. There is ample evidence that with these resources people other than the originators of STORM are effective instructors whose classes achieve the same educational outcomes as ours. In the following sections, we outline some of the courses, resources, tool support, and relevant experiences to date.

7.1 STPA-Sec Education

STPA-Sec in a modified form called Functional Mission Analysis for Cyber (FMA-C), is being applied across the Defense Department to improve mission assurance against cyber disruptions. The Air Force has directed

Course Introduction Topics	Number of nano-modules	Number of slides	Number of questions	Total Time (hh:mm:ss)
Welcome and Course Operations	2	15	10	00:30.00
Systems Security Engineering (NIST 800-160) Topics	**Number of nano-modules**	**Number of slides**	**Number of questions**	**Total Time (hh:mm:ss)**
Chapter 2: Fundamentals—Systems Security Engineering, System and System Elements, System Security Perspective, Systems Security Engineering Framework	4	30	16	01:00.00
Chapter 3: The Processes—Technical Processes: Business or Mission Analysis, Stakeholder & Requirements Definition, Architecture Definition, System Analysis, Implementation, Integration, and Verification	9	36	18	01:30.00
Course Totals	13	66	34	02:30.00
Systems Theoretic Process Analysis for Security Topics	**Number of nano-modules**	**Number of slides**	**Number of questions**	**Total Time (hh:mm:ss)**
Introduction to STPA-Sec, systems analysis, problem framing, systems thinking, challenges associated with applying traditional risk to cyberspace operations and technology	4	30	16	1:00.00
Problem Framework: Mission/Goal/Purpose, Unacceptable Losses, Examples	16	120	48	04:00.00
Functional Framework: Hazards, Control Structure, Hazardous Control Actions, Constraints/Control Requirements, Examples	16	120	48	04:00.00
Enterprise Architecture: Components, Connections Flows, Disruption Scenarios (Adversary, Accident, Nature), Initial Control Set, Wargame, Refine, Transitioning output to CSBD	16	120	48	04:00.00
Course Totals	52	390	160	13:00.00
Systems Theoretic Process Analysis for Security Course Totals	65	456	194	15:30.00

Table 7.1: STPA-Sec Online Asynchronous Learning Modules

that FMA be used as the baseline mission analysis methodology for all its new Mission Defense Teams. *The Air Force CIO now requires that all Service cyber professionals learn FMA as part of their continuing professional education.*

STPA-Sec has been taught at the University of Florida Research and Education Facility (REEF). The course description is as follows.

AQ-EN 599, System-Theoretic Process Analysis for Security (Graduate/undergrad)

Course Description: A wide range of potential adversaries now seek to disrupt or otherwise exploit the Information and Communications Technology Infrastructure. The disruptions target data processing functions, data storage functions, control functions for critical systems, or even the network's connectivity functions themselves. A new level of rigor is required to augment existing security engineering practice. The new rigor will enhance security engineering by bringing new perspectives to bear. A key to this rigor will be the development of new security analysis techniques and methodologies that allow engineers to develop strategies that complement existing protection-focused tactics with new and more effective mission assurance strategies.

Dr. Young will present a new Security Analysis technique based on Systems Theory. The new technique, System-Theoretic Process Analysis for Security (STPA-Sec), applies a system-theoretic framework to allow engineers to better balance the requirements for desired functionality with the requirement to control undesired functionality (system misbehavior). Systems-Theoretic Process Analysis for Security (STPA-Sec) augments traditional security

approaches by introducing a top-down analysis process designed to help a multidisciplinary team consisting of security, operations, and domain experts identify and constrain the system from entering unsecure states that lead to losses. This new framework shifts the focus of the security analysis away from threats as the proximate cause of losses and focuses instead on the broader system structure that allowed the system to enter an exploitable state.

Based on existing educational materials, we are planning to create online asynchronous learning modules in the form of SCORMS (Sharable Content Object Reference Models). Modules would include narrated slides, video screen captures of problem solving, video screen captures of tools, relevant videos of stories and interviews, and automatically graded assessments. Delivering the modules as SCORMS enables easy deployment by faculty using popular learning management systems (LMS) such as Blackboard and Moodle.

Table 7.1 is a detailed summary of the content, phasing, and times for an STPA-Sec course with specific connections to Systems Security Engineering (NIST 800-160) and Certified Security by Design (CSBD). Based on the experience to date teaching FMA-C within the Air Force, we anticipate an online STPA-Sec course would consist of 65 modules, covering 456 slides, 194 automatically graded questions, running 15 hours and 30 minutes.

7.2 CSBD Education

CSBD in its present form—focused on delivering easily and quickly reproducible formal verifications of system security properties—has been routinely taught at Syracuse University since 2014. These courses were developed as part of the Cyber Engineering Semester created and taught with Air Force Research Lab staff [26]. CSBD has two components: (1) an access-control logic component focused on reasoning about authentication and authorization, and (2) a computer-assisted reasoning component using theorem proving tools combined with the LaTeX document processing system to formally verify and document systems have the property of *complete mediation*, i.e., instructions are executed if and only if they are authenticated and authorized. At the undergraduate level, these components are taught as two elective courses to juniors and seniors. At the graduate level, CSBD is taught in a single course and is a core course in Syracuse University's MS Cybersecurity program.

The course descriptions for CSBD are as follows.

CIS 487: Access Control, Security, and Trust: A Logical Approach (undergraduate level)

Course Description: This course teaches you the principles of security by design that apply to a variety of systems including network protocols, computer hardware, virtual machines, and financial networks. We learn the details of engineering security and integrity into all levels of a system. This approach applies from the hardware level up to and including the information security policies that govern organizations. Topics include: (a) access-control concepts and reasoning about access control using an access-control logic; (b) authentication and authorization; (c) process isolation and sharing, virtual machines and memory protection, access control using descriptors and capabilities; and (d) security and integrity policies. Grades are based on a combination of homework and exams.

CIS 400: Certified Security by Design (undergraduate level)

Course Description: This is a hands-on laboratory applying the principles of access control, security, and trust, in combination with computer-assisted reasoning tools (the Higher Order Logic theorem prover and functional programming), to specify, design, and verify secure systems. Security in this context means every executed command is both authenticated and authorized. The emphasis of the course is on doing. You will learn to: (a) design, analyze, and verify secure computer systems using cryptographic components and security policies, (b) create models, animate them, and verify their properties, and (c) use methods and tools that provide compelling evidence that your designs are assured and certified as secure in ways

that are readily checked by third parties. We will develop a command-and-control example from top to bottom starting with a high-level concept of operations, through communication protocols and secure messages, down to a high-level state machine implementation. Grades are based on a combination of laboratory exercises and homework.

CIS 634: Assurance Foundations (core graduate course)

Course Description: The engineering of assured systems requires the capability to rigorously specify and verify system behavior. When constructing physical systems, one can use physical principles as the basis of calculations that demonstrate one's claims are correct. This course will introduce you to the theory, practice, and tools for building highly assured systems in a context that supports independent verification. The emphasis throughout the course is on doing: you will gain direct experience applying mathematics—in various ways to understand, explore and animate them, and then verify your ideas. By the end of the semester, you will be versed in executing the virtuous cycle of specifying, designing, implementing, and formally verifying systems.

Online asynchronous learning modules exist for all three CSBD courses. CIS 634, which is part of Syracuse University's MS Cybersecurity program, is part of both the online and on-campus MS programs of study. Table 7.2 is a detailed summary of the content, phasing, and times for CIS 634: Assurance Foundations. There are a total of 90 modules consisting of 713 narrated slides and videos with 211 automatically graded questions. Total module time is 21 hours and 58 minutes.

Students in CIS 400 and CIS 634 do 13 weekly projects. These projects require full engineering reports typeset using LaTeX. The HOL theorem prover generates LaTeX macros that typeset all formulas, datatypes, definitions, theorems, and theories in HOL. This capability adds both convenience and assurance of correctness to reports. There are few, if any, times students manually typeset formulas. Documentation is automatically updated when HOL theories change.

Each week, students turn in their source code files in HOL and LaTeX so instructors can quickly reproduce and verify results. Only 33% of a student's project grade depends on the proof code, the remaining 67% depends on the report and successful reproduction of all HOL theories, verifications in HOL, and LaTeX documentation.

Besides the online asynchronous learning modules, CSBD support includes two textbooks and an open-source virtual machine with all the tools and HOL theories necessary for the courses. The textbooks are *Access Control, Security, and Trust: A Logical Approach* [13], and *Certified Security by Design Using Higher Order Logic* [14]. Ubuntu Linux virtual machines running on VirtualBox have all the necessary HOL and LaTeX tools and theories for the courses. The machines are available at `http://ecs.syr.edu/faculty/chin/cis_assurance_foundations/`

7.3 Application of STORM in the AFRL ACE Internship

Elements of CSBD have been taught in the AFRL Advanced Course in Engineering (ACE) Cybersecurity Boot Camp since its inception in 2003 [23] [8] [10] [11] [12]. The two CSBD textbooks were written in large part due to the ACE. Each successive summer, interns were given increasingly more involved and more difficult problems for which they had to provide formal assurances of secure behavior. Over 500 ACE interns have learned elements of CSBD. These interns have come from over 50 different universities in the US and UK.

Of note is that one team of ACE interns in 2017 developed the UAV Payload Controller CONOPS using STPA-Sec as described in Tables 5.1 5.5 5.3 5.4 5.5. A second ACE intern team in 2017 developed the authentication and authorization justifications for the UAV Payload Controller CONOPS and made substantial progress in expressing and verifying the CONOPS using secure state machines.

The educational outcome for Systems Security Engineering is this: when given stakeholder objectives for protection of assets, students will apply systems and security principles, analyses, and tools to produce outcomes that (1) prevent and control asset loss and associated consequences, and (2) substantiate security and trustworthiness claims using evidence-based reasoning. The courseware developed for STORM, with the results to date in courses and internships, are evidence that the educational outcomes for Systems Security Engineering are feasible, practical, and reproducible with STORM.

Course Introduction Topics	Number of nano-modules	Number of slides	Number of questions	Total Time (hh:mm:ss)
Welcome and Course Operations; Welcome and Course Operations	2	22	13	00:32:57
Access-Control Logic Topics (Chapters refer to textbook Access Control, Security and Trust: A Logical Approach)	**Number of nano-modules**	**Number of slides**	**Number of questions**	**Total Time (hh:mm:ss)**
Chapter 1: Access Control, Security, Trust, and Logic; Chapter 2: A Language for Access Control-syntax and semantics	6	53	21	01:16:17
Chapter 3: Reasoning About Access Control-Logical rules and soundness	2	16	4	00:29:22
Chapter 4: Basic Concepts-tickets, access control lists, reference monitors	4	26	6	00:27:28
Chapter 5: Security Policies-Military and Commercial	3	22	3	00:24:58
Chapter 6: Digital Authentication-crypto operations	3	31	10	00:39:44
Chapter 7: Delegation, Concepts of Operations, and Roles	2	20	4	00:19:32
Chapter 9: A Primer on Computer Hardware	6	33	15	01:02:52
Chapter 10: Virtual Machines and Memory Protection	6	63	22	00:56:17
Course Totals	32	264	85	05:36:30
Higher Order Logic (HOL) Topics (Chapters refer to textbook Certified Security by Design Using Higher Order Logic)	**Number of nano-modules**	**Number of slides**	**Number of questions**	**Total Time (hh:mm:ss)**
Chapters 2, 3, & 4: Introduction to Linux, ML, and Emacs	5	21	11	01:11:21
Chapters 6, 7, 8, 9, 10, & 11: Introduction to HOL-forward and goal oriented proofs, algebraic types, structural induction	14	89	32	03:48:19
Chapters 13 & 14: An Access-Control Logic in HOL & Concepts of Operation	4	41	11	01:46:36
Chapters 15: Cryptographic Operations in HOL	5	28	7	00:24:20
Chapters 16, 17, & 18: Transition Systems – High Level State Machines, Secure State Machines, and Secure State Machine Refinements in HOL	20	188	28	05:11:14
Course Totals	48	367	89	12:21:50
LaTeX Combined with HOL Topics	**Number of nano-modules**	**Number of slides**	**Number of questions**	**Total Time (hh:mm:ss)**
Basic LaTeX, Project Reports, Using the HOL EmitTeX Library, debugging	8	60	24	03:26:59
Certified Security by Design Course Totals	90	713	211	21:58:16

Table 7.2: CSBD Online Asynchronous Learning Modules

Chapter 8

Conclusions

"Genius is one percent inspiration, ninety nine percent perspiration."
Thomas Edison

STORM (Systems-Theoretic and Technical Operational Risk Management) is a process for Systems Security Engineering (SSE) that integrates STPA-Sec (Systems Theoretic Process Analysis for Security) and CSBD (Certified Security by Design) to assure missions and manage risk. STORM focuses on CONOPS (Concept of Operation). STPA-Sec is used to *validate* a CONOPS derived from a mission statement, enumeration of unacceptable losses, analyzing scenarios that produce unacceptable losses, and devising suitable constraints and controls to prevent losses.

CSBD is used to *verify* a CONOPS satisfies requirements and constraints. In particular, CSBD focuses on *complete mediation*, i.e., all actions are taken if and only if the action is authenticated and authorized.

The promise of SSE and STORM is through disciplined application of systems and security principles, coupled with formal logic and hard work, we can repeatably develop secure and trustworthy systems that are assured to meet mission requirements.

STORM assures missions and manages risk by

1. Using STPA-Sec to devise and validate the right mission CONOPS, and
2. Using CSBD to formally verify the implementation and properties of the mission CONOPS, in order to provide assurances of conceptual unity, consistency, and completeness as the basis for trustworthiness.

STORM provides conceptual unity by a process that makes explicit

1. A system's mission, i.e., a system's task, purpose, and actions to be taken
2. Business/Mission analysis that produces the problem domain, stakeholders, conditions and constraints bounding the solution domain, requirements, and validation criteria
3. Hazards, i.e., conditions with the potential to cause injury, illness, or death; damage to or loss of equipment or property; or mission degradation
4. Security Control, i.e., safeguards or countermeasures designed to protect a system's confidentiality, integrity, and availability to satisfy security requirements in order to develop an assurance case for acceptable security

STORM provides formal verification of consistency and completeness by formal definitions and formal proofs of security properties in order to demonstrate the assurance case is satisfied. Formal proofs are the basis for demonstrating the case for assurance is satisfied.

In this report, we illustrated the application of STORM to a UAV Payload Controller. We have applied STORM to the development and verification of the following examples.

- A secure memory loader verifier implemented in hardware to load mission software into F-16 Vipers.
- A networked thermostat [9] similar to a NEST thermostat.
- We are developing a STORM description of US Army Patrol Base Operations, as described in the US Army Ranger Handbook [20].

Level	Characteristics
Level 1: Initial	Processes are undocumented, changing, and ad hoc
Level 2: Repeatable	Some processes are repeatable, possibly with consistent results, unlikely to be rigorous
Level 3: Defined	Some defined and documented standard processes exist and these processes are validated in a variety of scenarios
Level 4: Managed (Capable)	Process metrics exist, achievement of process objectives see over a range of operational conditions
Level 5: Optimizing	Focus is on continuous process improvement and addressing statistical common causes of process variation

Table 8.1: Capability Maturity Model Levels and Characteristics

Elements of STORM are taught in a variety of venues, including the US Air Force, University of Florida, and Syracuse University. STORM is taught using a variety of mediums, including traditional on-campus classes, within on-line educational programs, and summer research internships.

STORM as an engineering process is still developing. Borrowing the language of the Capability Maturity Model (CMM) [21], STORM is at Level 2 (Repeatable) and moving into Level 3 (Defined). Table 8.1 summarizes the CMM levels and their defining characteristics.

Our immediate objective is to fully define STORM in terms of defined and documented standard processes. This will put STORM into CMM Level 3. We are already testing STORM out on a variety of scenarios, which is key for STORM reaching CMM Levels 4 (Capable) and 5 (Optimizing).

Appendix A

The Access-Control Logic in HOL

A.1 aclfoundation Theory

Built: 04 March 2017
Parent Theories: indexedLists, patternMatches

A.1.1 Datatypes

```
Form =
    TT
  | FF
  | prop 'aavar
  | notf (('aavar, 'apn, 'il, 'sl) Form)
  | (andf) (('aavar, 'apn, 'il, 'sl) Form)
           (('aavar, 'apn, 'il, 'sl) Form)
  | (orf) (('aavar, 'apn, 'il, 'sl) Form)
          (('aavar, 'apn, 'il, 'sl) Form)
  | (impf) (('aavar, 'apn, 'il, 'sl) Form)
           (('aavar, 'apn, 'il, 'sl) Form)
  | (eqf) (('aavar, 'apn, 'il, 'sl) Form)
          (('aavar, 'apn, 'il, 'sl) Form)
  | (says) ('apn Princ) (('aavar, 'apn, 'il, 'sl) Form)
  | (speaks_for) ('apn Princ) ('apn Princ)
  | (controls) ('apn Princ) (('aavar, 'apn, 'il, 'sl) Form)
  | reps ('apn Princ) ('apn Princ)
         (('aavar, 'apn, 'il, 'sl) Form)
  | (domi) (('apn, 'il) IntLevel) (('apn, 'il) IntLevel)
  | (eqi) (('apn, 'il) IntLevel) (('apn, 'il) IntLevel)
  | (doms) (('apn, 'sl) SecLevel) (('apn, 'sl) SecLevel)
  | (eqs) (('apn, 'sl) SecLevel) (('apn, 'sl) SecLevel)
  | (eqn) num num
  | (lte) num num
  | (lt) num num

Kripke =
    KS ('aavar -> 'aaworld -> bool)
       ('apn -> 'aaworld -> 'aaworld -> bool) ('apn -> 'il)
       ('apn -> 'sl)

Princ =
    Name 'apn
  | (meet) ('apn Princ) ('apn Princ)
  | (quoting) ('apn Princ) ('apn Princ) ;

IntLevel = iLab 'il | il 'apn ;

SecLevel = sLab 'sl | sl 'apn
```

A.1.2 Definitions

[imapKS_def]

$\vdash \forall\, Intp\ Jfn\ ilmap\ slmap.$
`imapKS (KS` $Intp\ Jfn\ ilmap\ slmap$`) =` $ilmap$

[intpKS_def]

$\vdash \forall\, Intp\ Jfn\ ilmap\ slmap.$
`intpKS (KS` $Intp\ Jfn\ ilmap\ slmap$`) =` $Intp$

[jKS_def]

$\vdash \forall\, Intp\ Jfn\ ilmap\ slmap.$ `jKS (KS` $Intp\ Jfn\ ilmap\ slmap$`) =` Jfn

[O1_def]

$\vdash$ `O1 = PO one_weakorder`

[one_weakorder_def]

$\vdash \forall\, x\ y.$ `one_weakorder` $x\ y \iff$ `T`

[po_TY_DEF]

$\vdash \exists\, rep.$ `TYPE_DEFINITION WeakOrder` rep

[po_tybij]

$\vdash$ `(`$\forall\, a.$ `PO (repPO` a`) =` a`) ∧`
$\forall\, r.$ `WeakOrder` $r \iff$ `(repPO (PO` r`) =` r`)`

[prod_PO_def]

$\vdash \forall\, PO_1\ PO_2.$
`prod_PO` $PO_1\ PO_2$ `= PO (RPROD (repPO` PO_1`) (repPO` PO_2`))`

[smapKS_def]

$\vdash \forall\, Intp\ Jfn\ ilmap\ slmap.$
`smapKS (KS` $Intp\ Jfn\ ilmap\ slmap$`) =` $slmap$

[Subset_PO_def]

$\vdash$ `Subset_PO = PO (⊆)`

A.1.3 Theorems

[abs_po11]

$\vdash \forall\, r\ r'.$
`WeakOrder` $r \Rightarrow$ `WeakOrder` $r' \Rightarrow$ `((PO` r `= PO` r'`)` $\iff$ `(`r `=` r'`))`

[absPO_fn_onto]

$\vdash \forall\, a.\ \exists\, r.$ `(`a `= PO` r`) ∧ WeakOrder` r

[antisym_prod_antisym]

$\vdash \forall\, r\ s.$
`antisymmetric` r `∧ antisymmetric` $s \Rightarrow$
`antisymmetric (RPROD` $r\ s$`)`

[EQ_WeakOrder]

$\vdash$ `WeakOrder (=)`

[KS_bij]

$\vdash \forall M.\ M = \mathtt{KS}\ (\mathtt{intpKS}\ M)\ (\mathtt{jKS}\ M)\ (\mathtt{imapKS}\ M)\ (\mathtt{smapKS}\ M)$

[one_weakorder_WO]

$\vdash \mathtt{WeakOrder}\ \mathtt{one\_weakorder}$

[onto_po]

$\vdash \forall r.\ \mathtt{WeakOrder}\ r \iff \exists a.\ r = \mathtt{repPO}\ a$

[po_bij]

$\vdash (\forall a.\ \mathtt{PO}\ (\mathtt{repPO}\ a) = a) \wedge$
$\quad \forall r.\ \mathtt{WeakOrder}\ r \iff (\mathtt{repPO}\ (\mathtt{PO}\ r) = r)$

[PO_repPO]

$\vdash \forall a.\ \mathtt{PO}\ (\mathtt{repPO}\ a) = a$

[refl_prod_refl]

$\vdash \forall r\ s.\ \mathtt{reflexive}\ r \wedge \mathtt{reflexive}\ s \Rightarrow \mathtt{reflexive}\ (\mathtt{RPROD}\ r\ s)$

[repPO_iPO_partial_order]

$\vdash (\forall x.\ \mathtt{repPO}\ iPO\ x\ x) \wedge$
$\quad (\forall x\ y.\ \mathtt{repPO}\ iPO\ x\ y \wedge \mathtt{repPO}\ iPO\ y\ x \Rightarrow (x = y)) \wedge$
$\quad \forall x\ y\ z.\ \mathtt{repPO}\ iPO\ x\ y \wedge \mathtt{repPO}\ iPO\ y\ z \Rightarrow \mathtt{repPO}\ iPO\ x\ z$

[repPO_O1]

$\vdash \mathtt{repPO}\ \mathtt{O1} = \mathtt{one\_weakorder}$

[repPO_prod_PO]

$\vdash \forall po_1\ po_2.$
$\quad \mathtt{repPO}\ (\mathtt{prod\_PO}\ po_1\ po_2) = \mathtt{RPROD}\ (\mathtt{repPO}\ po_1)\ (\mathtt{repPO}\ po_2)$

[repPO_Subset_PO]

$\vdash \mathtt{repPO}\ \mathtt{Subset\_PO} = (\subseteq)$

[RPROD_THM]

$\vdash \forall r\ s\ a\ b.$
$\quad \mathtt{RPROD}\ r\ s\ a\ b \iff r\ (\mathtt{FST}\ a)\ (\mathtt{FST}\ b) \wedge s\ (\mathtt{SND}\ a)\ (\mathtt{SND}\ b)$

[SUBSET_WO]

$\vdash \mathtt{WeakOrder}\ (\subseteq)$

[trans_prod_trans]

$\vdash \forall r\ s.\ \mathtt{transitive}\ r \wedge \mathtt{transitive}\ s \Rightarrow \mathtt{transitive}\ (\mathtt{RPROD}\ r\ s)$

[WeakOrder_Exists]

$\vdash \exists R.\ \mathtt{WeakOrder}\ R$

[WO_prod_WO]

$\vdash \forall r\ s.\ \mathtt{WeakOrder}\ r \wedge \mathtt{WeakOrder}\ s \Rightarrow \mathtt{WeakOrder}\ (\mathtt{RPROD}\ r\ s)$

[WO_repPO]

$\vdash \forall r.\ \mathtt{WeakOrder}\ r \iff (\mathtt{repPO}\ (\mathtt{PO}\ r) = r)$

A.2 aclsemantics Theory

Built: 07 March 2017
Parent Theories: aclfoundation

A.2.1 Definitions

[Efn_def]

$$
\begin{array}{l}
\vdash\ (\forall\, Oi\ Os\ M.\ \texttt{Efn}\ Oi\ Os\ M\ \texttt{TT} = \mathcal{U}(\texttt{:'v})) \wedge \\
\quad (\forall\, Oi\ Os\ M.\ \texttt{Efn}\ Oi\ Os\ M\ \texttt{FF} = \{\,\}) \wedge \\
\quad (\forall\, Oi\ Os\ M\ p.\ \texttt{Efn}\ Oi\ Os\ M\ (\texttt{prop}\ p) = \texttt{intpKS}\ M\ p) \wedge \\
\quad (\forall\, Oi\ Os\ M\ f. \\
\qquad \texttt{Efn}\ Oi\ Os\ M\ (\texttt{notf}\ f) = \mathcal{U}(\texttt{:'v})\ \texttt{DIFF Efn}\ Oi\ Os\ M\ f) \wedge \\
\quad (\forall\, Oi\ Os\ M\ f_1\ f_2. \\
\qquad \texttt{Efn}\ Oi\ Os\ M\ (f_1\ \texttt{andf}\ f_2) = \\
\qquad \texttt{Efn}\ Oi\ Os\ M\ f_1 \cap \texttt{Efn}\ Oi\ Os\ M\ f_2) \wedge \\
\quad (\forall\, Oi\ Os\ M\ f_1\ f_2. \\
\qquad \texttt{Efn}\ Oi\ Os\ M\ (f_1\ \texttt{orf}\ f_2) = \\
\qquad \texttt{Efn}\ Oi\ Os\ M\ f_1 \cup \texttt{Efn}\ Oi\ Os\ M\ f_2) \wedge \\
\quad (\forall\, Oi\ Os\ M\ f_1\ f_2. \\
\qquad \texttt{Efn}\ Oi\ Os\ M\ (f_1\ \texttt{impf}\ f_2) = \\
\qquad \mathcal{U}(\texttt{:'v})\ \texttt{DIFF Efn}\ Oi\ Os\ M\ f_1 \cup \texttt{Efn}\ Oi\ Os\ M\ f_2) \wedge \\
\quad (\forall\, Oi\ Os\ M\ f_1\ f_2. \\
\qquad \texttt{Efn}\ Oi\ Os\ M\ (f_1\ \texttt{eqf}\ f_2) = \\
\qquad (\mathcal{U}(\texttt{:'v})\ \texttt{DIFF Efn}\ Oi\ Os\ M\ f_1 \cup \texttt{Efn}\ Oi\ Os\ M\ f_2) \cap \\
\qquad (\mathcal{U}(\texttt{:'v})\ \texttt{DIFF Efn}\ Oi\ Os\ M\ f_2 \cup \texttt{Efn}\ Oi\ Os\ M\ f_1)) \wedge \\
\quad (\forall\, Oi\ Os\ M\ P\ f. \\
\qquad \texttt{Efn}\ Oi\ Os\ M\ (P\ \texttt{says}\ f) = \\
\qquad \{\, w \mid \texttt{Jext (jKS}\ M)\ P\ w \subseteq \texttt{Efn}\ Oi\ Os\ M\ f\,\}) \wedge \\
\quad (\forall\, Oi\ Os\ M\ P\ Q. \\
\qquad \texttt{Efn}\ Oi\ Os\ M\ (P\ \texttt{speaks\_for}\ Q) = \\
\qquad \textbf{if}\ \texttt{Jext (jKS}\ M)\ Q\ \texttt{RSUBSET Jext (jKS}\ M)\ P\ \textbf{then}\ \mathcal{U}(\texttt{:'v}) \\
\qquad \textbf{else}\ \{\,\}) \wedge \\
\quad (\forall\, Oi\ Os\ M\ P\ f. \\
\qquad \texttt{Efn}\ Oi\ Os\ M\ (P\ \texttt{controls}\ f) = \\
\qquad \mathcal{U}(\texttt{:'v})\ \texttt{DIFF}\ \{\, w \mid \texttt{Jext (jKS}\ M)\ P\ w \subseteq \texttt{Efn}\ Oi\ Os\ M\ f\,\} \cup \\
\qquad \texttt{Efn}\ Oi\ Os\ M\ f) \wedge \\
\quad (\forall\, Oi\ Os\ M\ P\ Q\ f. \\
\qquad \texttt{Efn}\ Oi\ Os\ M\ (\texttt{reps}\ P\ Q\ f) = \\
\qquad \mathcal{U}(\texttt{:'v})\ \texttt{DIFF} \\
\qquad \{\, w \mid \texttt{Jext (jKS}\ M)\ (P\ \texttt{quoting}\ Q)\ w \subseteq \texttt{Efn}\ Oi\ Os\ M\ f\,\} \cup \\
\qquad \{\, w \mid \texttt{Jext (jKS}\ M)\ Q\ w \subseteq \texttt{Efn}\ Oi\ Os\ M\ f\,\}) \wedge \\
\quad (\forall\, Oi\ Os\ M\ intl_1\ intl_2. \\
\qquad \texttt{Efn}\ Oi\ Os\ M\ (intl_1\ \texttt{domi}\ intl_2) = \\
\qquad \textbf{if}\ \texttt{repPO}\ Oi\ \texttt{(Lifn}\ M\ intl_2\texttt{)\ (Lifn}\ M\ intl_1\texttt{)}\ \textbf{then}\ \mathcal{U}(\texttt{:'v}) \\
\qquad \textbf{else}\ \{\,\}) \wedge \\
\quad (\forall\, Oi\ Os\ M\ intl_2\ intl_1. \\
\qquad \texttt{Efn}\ Oi\ Os\ M\ (intl_2\ \texttt{eqi}\ intl_1) = \\
\qquad (\textbf{if}\ \texttt{repPO}\ Oi\ \texttt{(Lifn}\ M\ intl_2\texttt{)\ (Lifn}\ M\ intl_1\texttt{)}\ \textbf{then}\ \mathcal{U}(\texttt{:'v}) \\
\qquad\ \textbf{else}\ \{\,\}) \cap \\
\qquad \textbf{if}\ \texttt{repPO}\ Oi\ \texttt{(Lifn}\ M\ intl_1\texttt{)\ (Lifn}\ M\ intl_2\texttt{)}\ \textbf{then}\ \mathcal{U}(\texttt{:'v}) \\
\qquad \textbf{else}\ \{\,\}) \wedge \\
\quad (\forall\, Oi\ Os\ M\ secl_1\ secl_2. \\
\qquad \texttt{Efn}\ Oi\ Os\ M\ (secl_1\ \texttt{doms}\ secl_2) = \\
\qquad \textbf{if}\ \texttt{repPO}\ Os\ \texttt{(Lsfn}\ M\ secl_2\texttt{)\ (Lsfn}\ M\ secl_1\texttt{)}\ \textbf{then}\ \mathcal{U}(\texttt{:'v}) \\
\qquad \textbf{else}\ \{\,\}) \wedge \\
\quad (\forall\, Oi\ Os\ M\ secl_2\ secl_1.
\end{array}
$$

```
      Efn Oi Os M (secl2 eqs secl1) =
      (if repPO Os (Lsfn M secl2) (Lsfn M secl1) then U(:'v)
       else { }) ∩
      if repPO Os (Lsfn M secl1) (Lsfn M secl2) then U(:'v)
      else { }) ∧
   (∀ Oi Os M numExp1 numExp2.
      Efn Oi Os M (numExp1 eqn numExp2) =
      if numExp1 = numExp2 then U(:'v) else { }) ∧
   (∀ Oi Os M numExp1 numExp2.
      Efn Oi Os M (numExp1 lte numExp2) =
      if numExp1 ≤ numExp2 then U(:'v) else { }) ∧
   ∀ Oi Os M numExp1 numExp2.
     Efn Oi Os M (numExp1 lt numExp2) =
     if numExp1 < numExp2 then U(:'v) else { }
```

[Jext_def]

```
⊢ (∀ J s. Jext J (Name s) = J s) ∧
  (∀ J P1 P2.
     Jext J (P1 meet P2) = Jext J P1 RUNION Jext J P2) ∧
  ∀ J P1 P2. Jext J (P1 quoting P2) = Jext J P2 O Jext J P1
```

[Lifn_def]

```
⊢ (∀ M l. Lifn M (iLab l) = l) ∧
  ∀ M name. Lifn M (il name) = imapKS M name
```

[Lsfn_def]

```
⊢ (∀ M l. Lsfn M (sLab l) = l) ∧
  ∀ M name. Lsfn M (sl name) = smapKS M name
```

A.2.2 Theorems

[andf_def]

```
⊢ ∀ Oi Os M f1 f2.
    Efn Oi Os M (f1 andf f2) = Efn Oi Os M f1 ∩ Efn Oi Os M f2
```

[controls_def]

```
⊢ ∀ Oi Os M P f.
    Efn Oi Os M (P controls f) =
    U(:'v) DIFF {w | Jext (jKS M) P w ⊆ Efn Oi Os M f} ∪
    Efn Oi Os M f
```

[controls_says]

```
⊢ ∀ M P f.
    Efn Oi Os M (P controls f) = Efn Oi Os M (P says f impf f)
```

[domi_def]

```
⊢ ∀ Oi Os M intl1 intl2.
    Efn Oi Os M (intl1 domi intl2) =
    if repPO Oi (Lifn M intl2) (Lifn M intl1) then U(:'v)
    else { }
```

[doms_def]

```
⊢ ∀ Oi Os M secl1 secl2.
    Efn Oi Os M (secl1 doms secl2) =
    if repPO Os (Lsfn M secl2) (Lsfn M secl1) then U(:'v)
    else { }
```

[eqf_def]

$\vdash \forall\, Oi\ Os\ M\ f_1\ f_2.$
`Efn` $Oi\ Os\ M$ `(`f_1 `eqf` f_2`) =`
`(`$\mathcal{U}$`(:'v) DIFF Efn` $Oi\ Os\ M\ f_1$ `∪ Efn` $Oi\ Os\ M\ f_2$`) ∩`
`(`$\mathcal{U}$`(:'v) DIFF Efn` $Oi\ Os\ M\ f_2$ `∪ Efn` $Oi\ Os\ M\ f_1$`)`

[eqf_impf]

$\vdash \forall\, M\ f_1\ f_2.$
`Efn` $Oi\ Os\ M$ `(`f_1 `eqf` f_2`) =`
`Efn` $Oi\ Os\ M$ `((`f_1 `impf` f_2`) andf (`f_2 `impf` f_1`))`

[eqi_def]

$\vdash \forall\, Oi\ Os\ M\ intl_2\ intl_1.$
`Efn` $Oi\ Os\ M$ `(`$intl_2$ `eqi` $intl_1$`) =`
`(`**if** `repPO` Oi `(Lifn` $M\ intl_2$`) (Lifn` $M\ intl_1$`)` **then** $\mathcal{U}$`(:'v)`
else { }`) ∩`
if `repPO` Oi `(Lifn` $M\ intl_1$`) (Lifn` $M\ intl_2$`)` **then** $\mathcal{U}$`(:'v)`
else { }

[eqi_domi]

$\vdash \forall\, M\ intL_1\ intL_2.$
`Efn` $Oi\ Os\ M$ `(`$intL_1$ `eqi` $intL_2$`) =`
`Efn` $Oi\ Os\ M$ `(`$intL_2$ `domi` $intL_1$ `andf` $intL_1$ `domi` $intL_2$`)`

[eqn_def]

$\vdash \forall\, Oi\ Os\ M\ numExp_1\ numExp_2.$
`Efn` $Oi\ Os\ M$ `(`$numExp_1$ `eqn` $numExp_2$`) =`
if $numExp_1$ `=` $numExp_2$ **then** $\mathcal{U}$`(:'v)` **else** { }

[eqs_def]

$\vdash \forall\, Oi\ Os\ M\ secl_2\ secl_1.$
`Efn` $Oi\ Os\ M$ `(`$secl_2$ `eqs` $secl_1$`) =`
`(`**if** `repPO` Os `(Lsfn` $M\ secl_2$`) (Lsfn` $M\ secl_1$`)` **then** $\mathcal{U}$`(:'v)`
else { }`) ∩`
if `repPO` Os `(Lsfn` $M\ secl_1$`) (Lsfn` $M\ secl_2$`)` **then** $\mathcal{U}$`(:'v)`
else { }

[eqs_doms]

$\vdash \forall\, M\ secL_1\ secL_2.$
`Efn` $Oi\ Os\ M$ `(`$secL_1$ `eqs` $secL_2$`) =`
`Efn` $Oi\ Os\ M$ `(`$secL_2$ `doms` $secL_1$ `andf` $secL_1$ `doms` $secL_2$`)`

[FF_def]

$\vdash \forall\, Oi\ Os\ M.$ `Efn` $Oi\ Os\ M$ `FF = { }`

[impf_def]

$\vdash \forall\, Oi\ Os\ M\ f_1\ f_2.$
`Efn` $Oi\ Os\ M$ `(`f_1 `impf` f_2`) =`
$\mathcal{U}$`(:'v) DIFF Efn` $Oi\ Os\ M\ f_1$ `∪ Efn` $Oi\ Os\ M\ f_2$

[lt_def]

$\vdash \forall\, Oi\ Os\ M\ numExp_1\ numExp_2.$
`Efn` $Oi\ Os\ M$ `(`$numExp_1$ `lt` $numExp_2$`) =`
if $numExp_1 < numExp_2$ **then** $\mathcal{U}$`(:'v)` **else** { }

[lte_def]

$\vdash \forall Oi\ Os\ M\ numExp_1\ numExp_2.$
`Efn` $Oi\ Os\ M$ `(`$numExp_1$ `lte` $numExp_2$`) =`
if $numExp_1 \leq numExp_2$ **then** $\mathcal{U}$`(:'v)` **else** $\{\}$

[meet_def]

$\vdash \forall J\ P_1\ P_2.$ `Jext` J `(`P_1 `meet` P_2`) = Jext` $J\ P_1$ `RUNION Jext` $J\ P_2$

[name_def]

$\vdash \forall J\ s.$ `Jext` J `(Name` s`) =` $J\ s$

[notf_def]

$\vdash \forall Oi\ Os\ M\ f.$ `Efn` $Oi\ Os\ M$ `(notf` f`) =` $\mathcal{U}$`(:'v) DIFF Efn` $Oi\ Os\ M\ f$

[orf_def]

$\vdash \forall Oi\ Os\ M\ f_1\ f_2.$
`Efn` $Oi\ Os\ M$ `(`f_1 `orf` f_2`) = Efn` $Oi\ Os\ M\ f_1 \cup$ `Efn` $Oi\ Os\ M\ f_2$

[prop_def]

$\vdash \forall Oi\ Os\ M\ p.$ `Efn` $Oi\ Os\ M$ `(prop` p`) = intpKS` $M\ p$

[quoting_def]

$\vdash \forall J\ P_1\ P_2.$ `Jext` J `(`P_1 `quoting` P_2`) = Jext` $J\ P_2$ `O Jext` $J\ P_1$

[reps_def]

$\vdash \forall Oi\ Os\ M\ P\ Q\ f.$
`Efn` $Oi\ Os\ M$ `(reps` $P\ Q\ f$`) =`
$\mathcal{U}$`(:'v) DIFF`
$\{w$ `|` `Jext (jKS` M`)` `(`P `quoting` Q`)` $w \subseteq$ `Efn` $Oi\ Os\ M\ f\} \cup$
$\{w$ `|` `Jext (jKS` M`)` $Q\ w \subseteq$ `Efn` $Oi\ Os\ M\ f\}$

[says_def]

$\vdash \forall Oi\ Os\ M\ P\ f.$
`Efn` $Oi\ Os\ M$ `(`P `says` f`) =`
$\{w$ `|` `Jext (jKS` M`)` $P\ w \subseteq$ `Efn` $Oi\ Os\ M\ f\}$

[speaks_for_def]

$\vdash \forall Oi\ Os\ M\ P\ Q.$
`Efn` $Oi\ Os\ M$ `(`P `speaks_for` Q`) =`
if `Jext (jKS` M`)` Q `RSUBSET Jext (jKS` M`)` P **then** $\mathcal{U}$`(:'v)`
else $\{\}$

[TT_def]

$\vdash \forall Oi\ Os\ M.$ `Efn` $Oi\ Os\ M$ `TT =` $\mathcal{U}$`(:'v)`

A.3 aclrules Theory

Built: 04 March 2017
Parent Theories: aclsemantics

A.3.1 Definitions

[sat_def]

$\vdash \forall M\ Oi\ Os\ f.$ `(`M`,`Oi`,`Os`) sat` $f \iff$ `(Efn` $Oi\ Os\ M\ f$ `=` $\mathcal{U}$`(:'world))`

A.3.2 Theorems

[And_Says]

$\vdash \forall M\ Oi\ Os\ P\ Q\ f.$
(M,Oi,Os) sat P meet Q says f eqf P says f andf Q says f

[And_Says_Eq]

$\vdash (M,Oi,Os)$ sat P meet Q says $f \iff$
(M,Oi,Os) sat P says f andf Q says f

[and_says_lemma]

$\vdash \forall M\ Oi\ Os\ P\ Q\ f.$
(M,Oi,Os) sat P meet Q says f impf P says f andf Q says f

[Controls_Eq]

$\vdash \forall M\ Oi\ Os\ P\ f.$
(M,Oi,Os) sat P controls $f \iff (M,Oi,Os)$ sat P says f impf f

[DIFF_UNIV_SUBSET]

$\vdash (\mathcal{U}(:'a)$ DIFF $s \cup t = \mathcal{U}(:'a)) \iff s \subseteq t$

[domi_antisymmetric]

$\vdash \forall M\ Oi\ Os\ l_1\ l_2.$
(M,Oi,Os) sat l_1 domi $l_2 \Rightarrow$
(M,Oi,Os) sat l_2 domi $l_1 \Rightarrow$
(M,Oi,Os) sat l_1 eqi l_2

[domi_reflexive]

$\vdash \forall M\ Oi\ Os\ l.\ (M,Oi,Os)$ sat l domi l

[domi_transitive]

$\vdash \forall M\ Oi\ Os\ l_1\ l_2\ l_3.$
(M,Oi,Os) sat l_1 domi $l_2 \Rightarrow$
(M,Oi,Os) sat l_2 domi $l_3 \Rightarrow$
(M,Oi,Os) sat l_1 domi l_3

[doms_antisymmetric]

$\vdash \forall M\ Oi\ Os\ l_1\ l_2.$
(M,Oi,Os) sat l_1 doms $l_2 \Rightarrow$
(M,Oi,Os) sat l_2 doms $l_1 \Rightarrow$
(M,Oi,Os) sat l_1 eqs l_2

[doms_reflexive]

$\vdash \forall M\ Oi\ Os\ l.\ (M,Oi,Os)$ sat l doms l

[doms_transitive]

$\vdash \forall M\ Oi\ Os\ l_1\ l_2\ l_3.$
(M,Oi,Os) sat l_1 doms $l_2 \Rightarrow$
(M,Oi,Os) sat l_2 doms $l_3 \Rightarrow$
(M,Oi,Os) sat l_1 doms l_3

[eqf_and_impf]

$\vdash \forall M\ Oi\ Os\ f_1\ f_2.$
(M,Oi,Os) sat f_1 eqf $f_2 \iff$
(M,Oi,Os) sat $(f_1$ impf $f_2)$ andf $(f_2$ impf $f_1)$

[eqf_andf1]

$\vdash \forall M\ Oi\ Os\ f\ f'\ g.$
(M, Oi, Os) sat f eqf $f' \Rightarrow$
(M, Oi, Os) sat f andf $g \Rightarrow$
(M, Oi, Os) sat f' andf g

[eqf_andf2]

$\vdash \forall M\ Oi\ Os\ f\ f'\ g.$
(M, Oi, Os) sat f eqf $f' \Rightarrow$
(M, Oi, Os) sat g andf $f \Rightarrow$
(M, Oi, Os) sat g andf f'

[eqf_controls]

$\vdash \forall M\ Oi\ Os\ P\ f\ f'.$
(M, Oi, Os) sat f eqf $f' \Rightarrow$
(M, Oi, Os) sat P controls $f \Rightarrow$
(M, Oi, Os) sat P controls f'

[eqf_eq]

$\vdash$ (Efn $Oi\ Os\ M$ (f_1 eqf f_2) = $\mathcal{U}$(:'b)) $\iff$
(Efn $Oi\ Os\ M\ f_1$ = Efn $Oi\ Os\ M\ f_2$)

[eqf_eqf1]

$\vdash \forall M\ Oi\ Os\ f\ f'\ g.$
(M, Oi, Os) sat f eqf $f' \Rightarrow$
(M, Oi, Os) sat f eqf $g \Rightarrow$
(M, Oi, Os) sat f' eqf g

[eqf_eqf2]

$\vdash \forall M\ Oi\ Os\ f\ f'\ g.$
(M, Oi, Os) sat f eqf $f' \Rightarrow$
(M, Oi, Os) sat g eqf $f \Rightarrow$
(M, Oi, Os) sat g eqf f'

[eqf_impf1]

$\vdash \forall M\ Oi\ Os\ f\ f'\ g.$
(M, Oi, Os) sat f eqf $f' \Rightarrow$
(M, Oi, Os) sat f impf $g \Rightarrow$
(M, Oi, Os) sat f' impf g

[eqf_impf2]

$\vdash \forall M\ Oi\ Os\ f\ f'\ g.$
(M, Oi, Os) sat f eqf $f' \Rightarrow$
(M, Oi, Os) sat g impf $f \Rightarrow$
(M, Oi, Os) sat g impf f'

[eqf_notf]

$\vdash \forall M\ Oi\ Os\ f\ f'.$
(M, Oi, Os) sat f eqf $f' \Rightarrow$
(M, Oi, Os) sat notf $f \Rightarrow$
(M, Oi, Os) sat notf f'

[eqf_orf1]

$\vdash \forall M\ Oi\ Os\ f\ f'\ g.$
(M, Oi, Os) sat f eqf $f' \Rightarrow$
(M, Oi, Os) sat f orf $g \Rightarrow$
(M, Oi, Os) sat f' orf g

[eqf_orf2]

$\vdash \forall M\ Oi\ Os\ f\ f'\ g.$
$(M,Oi,Os)\ \texttt{sat}\ f\ \texttt{eqf}\ f' \Rightarrow$
$(M,Oi,Os)\ \texttt{sat}\ g\ \texttt{orf}\ f \Rightarrow$
$(M,Oi,Os)\ \texttt{sat}\ g\ \texttt{orf}\ f'$

[eqf_reps]

$\vdash \forall M\ Oi\ Os\ P\ Q\ f\ f'.$
$(M,Oi,Os)\ \texttt{sat}\ f\ \texttt{eqf}\ f' \Rightarrow$
$(M,Oi,Os)\ \texttt{sat reps}\ P\ Q\ f \Rightarrow$
$(M,Oi,Os)\ \texttt{sat reps}\ P\ Q\ f'$

[eqf_sat]

$\vdash \forall M\ Oi\ Os\ f_1\ f_2.$
$(M,Oi,Os)\ \texttt{sat}\ f_1\ \texttt{eqf}\ f_2 \Rightarrow$
$((M,Oi,Os)\ \texttt{sat}\ f_1 \iff (M,Oi,Os)\ \texttt{sat}\ f_2)$

[eqf_says]

$\vdash \forall M\ Oi\ Os\ P\ f\ f'.$
$(M,Oi,Os)\ \texttt{sat}\ f\ \texttt{eqf}\ f' \Rightarrow$
$(M,Oi,Os)\ \texttt{sat}\ P\ \texttt{says}\ f \Rightarrow$
$(M,Oi,Os)\ \texttt{sat}\ P\ \texttt{says}\ f'$

[eqi_Eq]

$\vdash \forall M\ Oi\ Os\ l_1\ l_2.$
$(M,Oi,Os)\ \texttt{sat}\ l_1\ \texttt{eqi}\ l_2 \iff$
$(M,Oi,Os)\ \texttt{sat}\ l_2\ \texttt{domi}\ l_1\ \texttt{andf}\ l_1\ \texttt{domi}\ l_2$

[eqs_Eq]

$\vdash \forall M\ Oi\ Os\ l_1\ l_2.$
$(M,Oi,Os)\ \texttt{sat}\ l_1\ \texttt{eqs}\ l_2 \iff$
$(M,Oi,Os)\ \texttt{sat}\ l_2\ \texttt{doms}\ l_1\ \texttt{andf}\ l_1\ \texttt{doms}\ l_2$

[Idemp_Speaks_For]

$\vdash \forall M\ Oi\ Os\ P.\ (M,Oi,Os)\ \texttt{sat}\ P\ \texttt{speaks\_for}\ P$

[Image_cmp]

$\vdash \forall R_1\ R_2\ R_3\ u.\ (R_1\ \texttt{O}\ R_2)\ u \subseteq R_3 \iff R_2\ u \subseteq \{y\ |\ R_1\ y \subseteq R_3\}$

[Image_SUBSET]

$\vdash \forall R_1\ R_2.\ R_2\ \texttt{RSUBSET}\ R_1 \Rightarrow \forall w.\ R_2\ w \subseteq R_1\ w$

[Image_UNION]

$\vdash \forall R_1\ R_2\ w.\ (R_1\ \texttt{RUNION}\ R_2)\ w = R_1\ w \cup R_2\ w$

[INTER_EQ_UNIV]

$\vdash (s \cap t = \mathcal{U}(:\texttt{'a})) \iff (s = \mathcal{U}(:\texttt{'a})) \wedge (t = \mathcal{U}(:\texttt{'a}))$

[Modus_Ponens]

$\vdash \forall M\ Oi\ Os\ f_1\ f_2.$
$(M,Oi,Os)\ \texttt{sat}\ f_1 \Rightarrow$
$(M,Oi,Os)\ \texttt{sat}\ f_1\ \texttt{impf}\ f_2 \Rightarrow$
$(M,Oi,Os)\ \texttt{sat}\ f_2$

[Mono_speaks_for]

$\vdash \forall M\ Oi\ Os\ P\ P'\ Q\ Q'.$
(M, Oi, Os) sat P speaks_for $P' \Rightarrow$
(M, Oi, Os) sat Q speaks_for $Q' \Rightarrow$
(M, Oi, Os) sat P quoting Q speaks_for P' quoting Q'

[MP_Says]

$\vdash \forall M\ Oi\ Os\ P\ f_1\ f_2.$
(M, Oi, Os) sat
P says $(f_1$ impf $f_2)$ impf P says f_1 impf P says f_2

[Quoting]

$\vdash \forall M\ Oi\ Os\ P\ Q\ f.$
(M, Oi, Os) sat P quoting Q says f eqf P says Q says f

[Quoting_Eq]

$\vdash \forall M\ Oi\ Os\ P\ Q\ f.$
(M, Oi, Os) sat P quoting Q says $f \iff$
(M, Oi, Os) sat P says Q says f

[reps_def_lemma]

$\vdash \forall M\ Oi\ Os\ P\ Q\ f.$
Efn $Oi\ Os\ M$ (reps $P\ Q\ f$) =
Efn $Oi\ Os\ M$ (P quoting Q says f impf Q says f)

[Reps_Eq]

$\vdash \forall M\ Oi\ Os\ P\ Q\ f.$
(M, Oi, Os) sat reps $P\ Q\ f \iff$
(M, Oi, Os) sat P quoting Q says f impf Q says f

[sat_allworld]

$\vdash \forall M\ f.\ (M, Oi, Os)$ sat $f \iff \forall w.\ w \in$ Efn $Oi\ Os\ M\ f$

[sat_andf_eq_and_sat]

$\vdash (M, Oi, Os)$ sat f_1 andf $f_2 \iff$
(M, Oi, Os) sat $f_1 \wedge (M, Oi, Os)$ sat f_2

[sat_TT]

$\vdash (M, Oi, Os)$ sat TT

[Says]

$\vdash \forall M\ Oi\ Os\ P\ f.\ (M, Oi, Os)$ sat $f \Rightarrow (M, Oi, Os)$ sat P says f

[says_and_lemma]

$\vdash \forall M\ Oi\ Os\ P\ Q\ f.$
(M, Oi, Os) sat P says f andf Q says f impf P meet Q says f

[Speaks_For]

$\vdash \forall M\ Oi\ Os\ P\ Q\ f.$
(M, Oi, Os) sat P speaks_for Q impf P says f impf Q says f

[speaks_for_SUBSET]

$\vdash \forall R_3\ R_2\ R_1.$
R_2 RSUBSET $R_1 \Rightarrow \forall w.\ \{w \mid R_1\ w \subseteq R_3\} \subseteq \{w \mid R_2\ w \subseteq R_3\}$

[SUBSET_Image_SUBSET]

$\vdash \forall R_1\ R_2\ R_3.$
$(\forall w_1.\ R_2\ w_1 \subseteq R_1\ w_1) \Rightarrow$
$\forall w.\ \{w \mid R_1\ w \subseteq R_3\} \subseteq \{w \mid R_2\ w \subseteq R_3\}$

[Trans_Speaks_For]

$\vdash \forall M\ Oi\ Os\ P\ Q\ R.$
$(M, Oi, Os)\ \texttt{sat}\ P\ \texttt{speaks\_for}\ Q \Rightarrow$
$(M, Oi, Os)\ \texttt{sat}\ Q\ \texttt{speaks\_for}\ R \Rightarrow$
$(M, Oi, Os)\ \texttt{sat}\ P\ \texttt{speaks\_for}\ R$

[UNIV_DIFF_SUBSET]

$\vdash \forall R_1\ R_2.\ R_1 \subseteq R_2 \Rightarrow (\mathcal{U}(:'a)\ \texttt{DIFF}\ R_1 \cup R_2 = \mathcal{U}(:'a))$

[world_and]

$\vdash \forall M\ Oi\ Os\ f_1\ f_2\ w.$
$w \in \texttt{Efn}\ Oi\ Os\ M\ (f_1\ \texttt{andf}\ f_2) \iff$
$w \in \texttt{Efn}\ Oi\ Os\ M\ f_1 \wedge w \in \texttt{Efn}\ Oi\ Os\ M\ f_2$

[world_eq]

$\vdash \forall M\ Oi\ Os\ f_1\ f_2\ w.$
$w \in \texttt{Efn}\ Oi\ Os\ M\ (f_1\ \texttt{eqf}\ f_2) \iff$
$(w \in \texttt{Efn}\ Oi\ Os\ M\ f_1 \iff w \in \texttt{Efn}\ Oi\ Os\ M\ f_2)$

[world_eqn]

$\vdash \forall M\ Oi\ Os\ n_1\ n_2\ w.\ w \in \texttt{Efn}\ Oi\ Os\ m\ (n_1\ \texttt{eqn}\ n_2) \iff (n_1 = n_2)$

[world_F]

$\vdash \forall M\ Oi\ Os\ w.\ w \notin \texttt{Efn}\ Oi\ Os\ M\ \texttt{FF}$

[world_imp]

$\vdash \forall M\ Oi\ Os\ f_1\ f_2\ w.$
$w \in \texttt{Efn}\ Oi\ Os\ M\ (f_1\ \texttt{impf}\ f_2) \iff$
$w \in \texttt{Efn}\ Oi\ Os\ M\ f_1 \Rightarrow w \in \texttt{Efn}\ Oi\ Os\ M\ f_2$

[world_lt]

$\vdash \forall M\ Oi\ Os\ n_1\ n_2\ w.\ w \in \texttt{Efn}\ Oi\ Os\ m\ (n_1\ \texttt{lt}\ n_2) \iff n_1 < n_2$

[world_lte]

$\vdash \forall M\ Oi\ Os\ n_1\ n_2\ w.\ w \in \texttt{Efn}\ Oi\ Os\ m\ (n_1\ \texttt{lte}\ n_2) \iff n_1 \leq n_2$

[world_not]

$\vdash \forall M\ Oi\ Os\ f\ w.\ w \in \texttt{Efn}\ Oi\ Os\ M\ (\texttt{notf}\ f) \iff w \notin \texttt{Efn}\ Oi\ Os\ M\ f$

[world_or]

$\vdash \forall M\ f_1\ f_2\ w.$
$w \in \texttt{Efn}\ Oi\ Os\ M\ (f_1\ \texttt{orf}\ f_2) \iff$
$w \in \texttt{Efn}\ Oi\ Os\ M\ f_1 \vee w \in \texttt{Efn}\ Oi\ Os\ M\ f_2$

[world_says]

$\vdash \forall M\ Oi\ Os\ P\ f\ w.$
$w \in \texttt{Efn}\ Oi\ Os\ M\ (P\ \texttt{says}\ f) \iff$
$\forall v.\ v \in \texttt{Jext (jKS}\ M)\ P\ w \Rightarrow v \in \texttt{Efn}\ Oi\ Os\ M\ f$

[world_T]

$\vdash \forall M\ Oi\ Os\ w.\ w \in \texttt{Efn}\ Oi\ Os\ M\ \texttt{TT}$

A.4 aclDrules Theory

Built: 04 March 2017
Parent Theories: aclrules

A.4.1 Theorems

[Conjunction]
$\vdash \forall M\ Oi\ Os\ f_1\ f_2.$
(M, Oi, Os) sat $f_1 \Rightarrow$
(M, Oi, Os) sat $f_2 \Rightarrow$
(M, Oi, Os) sat f_1 andf f_2

[Controls]
$\vdash \forall M\ Oi\ Os\ P\ f.$
(M, Oi, Os) sat P says $f \Rightarrow$
(M, Oi, Os) sat P controls $f \Rightarrow$
(M, Oi, Os) sat f

[Derived_Controls]
$\vdash \forall M\ Oi\ Os\ P\ Q\ f.$
(M, Oi, Os) sat P speaks_for $Q \Rightarrow$
(M, Oi, Os) sat Q controls $f \Rightarrow$
(M, Oi, Os) sat P controls f

[Derived_Speaks_For]
$\vdash \forall M\ Oi\ Os\ P\ Q\ f.$
(M, Oi, Os) sat P speaks_for $Q \Rightarrow$
(M, Oi, Os) sat P says $f \Rightarrow$
(M, Oi, Os) sat Q says f

[Disjunction1]
$\vdash \forall M\ Oi\ Os\ f_1\ f_2.$ (M, Oi, Os) sat $f_1 \Rightarrow (M, Oi, Os)$ sat f_1 orf f_2

[Disjunction2]
$\vdash \forall M\ Oi\ Os\ f_1\ f_2.$ (M, Oi, Os) sat $f_2 \Rightarrow (M, Oi, Os)$ sat f_1 orf f_2

[Disjunctive_Syllogism]
$\vdash \forall M\ Oi\ Os\ f_1\ f_2.$
(M, Oi, Os) sat f_1 orf $f_2 \Rightarrow$
(M, Oi, Os) sat notf $f_1 \Rightarrow$
(M, Oi, Os) sat f_2

[Double_Negation]
$\vdash \forall M\ Oi\ Os\ f.$ (M, Oi, Os) sat notf (notf f) $\Rightarrow (M, Oi, Os)$ sat f

[eqn_eqn]
$\vdash (M, Oi, Os)$ sat c_1 eqn $n_1 \Rightarrow$
(M, Oi, Os) sat c_2 eqn $n_2 \Rightarrow$
(M, Oi, Os) sat n_1 eqn $n_2 \Rightarrow$
(M, Oi, Os) sat c_1 eqn c_2

[eqn_lt]
$\vdash (M, Oi, Os)$ sat c_1 eqn $n_1 \Rightarrow$
(M, Oi, Os) sat c_2 eqn $n_2 \Rightarrow$
(M, Oi, Os) sat n_1 lt $n_2 \Rightarrow$
(M, Oi, Os) sat c_1 lt c_2

[eqn_lte]

$\vdash$ (M, Oi, Os) sat c_1 eqn n_1 $\Rightarrow$
(M, Oi, Os) sat c_2 eqn n_2 $\Rightarrow$
(M, Oi, Os) sat n_1 lte n_2 $\Rightarrow$
(M, Oi, Os) sat c_1 lte c_2

[Hypothetical_Syllogism]

$\vdash \forall M\ Oi\ Os\ f_1\ f_2\ f_3.$
(M, Oi, Os) sat f_1 impf f_2 $\Rightarrow$
(M, Oi, Os) sat f_2 impf f_3 $\Rightarrow$
(M, Oi, Os) sat f_1 impf f_3

[il_domi]

$\vdash \forall M\ Oi\ Os\ P\ Q\ l_1\ l_2.$
(M, Oi, Os) sat il P eqi l_1 $\Rightarrow$
(M, Oi, Os) sat il Q eqi l_2 $\Rightarrow$
(M, Oi, Os) sat l_2 domi l_1 $\Rightarrow$
(M, Oi, Os) sat il Q domi il P

[INTER_EQ_UNIV]

$\vdash \forall s_1\ s_2.\ (s_1 \cap s_2 = \mathcal{U}(:'a)) \iff (s_1 = \mathcal{U}(:'a)) \wedge (s_2 = \mathcal{U}(:'a))$

[Modus_Tollens]

$\vdash \forall M\ Oi\ Os\ f_1\ f_2.$
(M, Oi, Os) sat f_1 impf f_2 $\Rightarrow$
(M, Oi, Os) sat notf f_2 $\Rightarrow$
(M, Oi, Os) sat notf f_1

[Rep_Controls_Eq]

$\vdash \forall M\ Oi\ Os\ A\ B\ f.$
(M, Oi, Os) sat reps A B f $\iff$
(M, Oi, Os) sat A controls B says f

[Rep_Says]

$\vdash \forall M\ Oi\ Os\ P\ Q\ f.$
(M, Oi, Os) sat reps P Q f $\Rightarrow$
(M, Oi, Os) sat P quoting Q says f $\Rightarrow$
(M, Oi, Os) sat Q says f

[Reps]

$\vdash \forall M\ Oi\ Os\ P\ Q\ f.$
(M, Oi, Os) sat reps P Q f $\Rightarrow$
(M, Oi, Os) sat P quoting Q says f $\Rightarrow$
(M, Oi, Os) sat Q controls f $\Rightarrow$
(M, Oi, Os) sat f

[Says_Simplification1]

$\vdash \forall M\ Oi\ Os\ P\ f_1\ f_2.$
(M, Oi, Os) sat P says $(f_1$ andf $f_2) \Rightarrow (M, Oi, Os)$ sat P says f_1

[Says_Simplification2]

$\vdash \forall M\ Oi\ Os\ P\ f_1\ f_2.$
(M, Oi, Os) sat P says $(f_1$ andf $f_2) \Rightarrow (M, Oi, Os)$ sat P says f_2

[Simplification1]

$\vdash \forall M\ Oi\ Os\ f_1\ f_2.\ (M, Oi, Os)$ sat f_1 andf $f_2 \Rightarrow (M, Oi, Os)$ sat f_1

[Simplification2]

$\vdash \forall M\ Oi\ Os\ f_1\ f_2.\ (M, Oi, Os)\ \texttt{sat}\ f_1\ \texttt{andf}\ f_2 \Rightarrow (M, Oi, Os)\ \texttt{sat}\ f_2$

[sl_doms]

$\vdash \forall M\ Oi\ Os\ P\ Q\ l_1\ l_2.$
$(M, Oi, Os)\ \texttt{sat sl}\ P\ \texttt{eqs}\ l_1 \Rightarrow$
$(M, Oi, Os)\ \texttt{sat sl}\ Q\ \texttt{eqs}\ l_2 \Rightarrow$
$(M, Oi, Os)\ \texttt{sat}\ l_2\ \texttt{doms}\ l_1 \Rightarrow$
$(M, Oi, Os)\ \texttt{sat sl}\ Q\ \texttt{doms sl}\ P$

Appendix B

Secure State Machine Theory and Payload Controller Theories

B.1 ssm1 Theory

Built: 16 March 2018
Parent Theories: satList

B.1.1 Datatypes

```
configuration =
    CFG (('command option, 'principal, 'd, 'e) Form -> bool)
        ('state ->
         ('command option, 'principal, 'd, 'e) Form list ->
         ('command option, 'principal, 'd, 'e) Form list)
        (('command option, 'principal, 'd, 'e) Form list ->
         ('command option, 'principal, 'd, 'e) Form list)
        (('command option, 'principal, 'd, 'e) Form list list)
        'state ('output list)

trType = discard 'cmdlist | trap 'cmdlist | exec 'cmdlist
```

B.1.2 Definitions

[authenticationTest_def]

$\vdash \forall\, elementTest\ x.$
`authenticationTest` $elementTest\ x \iff$
`FOLDR` $(\lambda\, p\ q.\ p \wedge q)$ `T (MAP` $elementTest\ x$`)`

[commandList_def]

$\vdash \forall\, x.$ `commandList` x `= MAP extractCommand` x

[inputList_def]

$\vdash \forall\, xs.$ `inputList` xs `= MAP extractInput` xs

[propCommandList_def]

$\vdash \forall\, x.$ `propCommandList` x `= MAP extractPropCommand` x

[TR_def]

$\vdash$ `TR =`
$(\lambda\, a_0\ a_1\ a_2\ a_3.$
$\forall\, TR'.$
$(\forall\, a_0\ a_1\ a_2\ a_3.$
$(\exists\, elementTest\ NS\ M\ Oi\ Os\ Out\ s\ context\ stateInterp\ x$
$ins\ outs.$
$(a_0$ `=` $(M, Oi, Os)) \wedge (a_1$ `= exec (inputList` x`)) \wedge$
$(a_2$ `=`

```
      CFG elementTest stateInterp context (x::ins) s
        outs) ∧
      (a3 =
       CFG elementTest stateInterp context ins
         (NS s (exec (inputList x)))
         (Out s (exec (inputList x))::outs)) ∧
      authenticationTest elementTest x ∧
      CFGInterpret (M,Oi,Os)
        (CFG elementTest stateInterp context (x::ins) s
           outs)) ∨
   (∃ elementTest NS M Oi Os Out s context stateInterp x
      ins outs.
      (a0 = (M,Oi,Os)) ∧ (a1 = trap (inputList x)) ∧
      (a2 =
       CFG elementTest stateInterp context (x::ins) s
         outs) ∧
      (a3 =
       CFG elementTest stateInterp context ins
         (NS s (trap (inputList x)))
         (Out s (trap (inputList x))::outs)) ∧
      authenticationTest elementTest x ∧
      CFGInterpret (M,Oi,Os)
        (CFG elementTest stateInterp context (x::ins) s
           outs)) ∨
   (∃ elementTest NS M Oi Os Out s context stateInterp x
      ins outs.
      (a0 = (M,Oi,Os)) ∧ (a1 = discard (inputList x)) ∧
      (a2 =
       CFG elementTest stateInterp context (x::ins) s
         outs) ∧
      (a3 =
       CFG elementTest stateInterp context ins
         (NS s (discard (inputList x)))
         (Out s (discard (inputList x))::outs)) ∧
      ¬authenticationTest elementTest x) ⇒
   TR' a0 a1 a2 a3) ⇒
TR' a0 a1 a2 a3)
```

B.1.3 Theorems

[CFGInterpret_def]

```
⊢ CFGInterpret (M,Oi,Os)
    (CFG elementTest stateInterp context (x::ins) state
       outStream) ⟺
  (M,Oi,Os) satList context x ∧ (M,Oi,Os) satList x ∧
  (M,Oi,Os) satList stateInterp state x
```

[CFGInterpret_ind]

```
⊢ ∀P.
    (∀M Oi Os elementTest stateInterp context x ins state
       outStream.
       P (M,Oi,Os)
         (CFG elementTest stateInterp context (x::ins) state
            outStream)) ∧
    (∀v15 v10 v11 v12 v13 v14.
       P v15 (CFG v10 v11 v12 [] v13 v14)) ⇒
    ∀v v1 v2 v3. P (v,v1,v2) v3
```

[configuration_one_one]

$\vdash \forall\, a_0\ a_1\ a_2\ a_3\ a_4\ a_5\ a_0'\ a_1'\ a_2'\ a_3'\ a_4'\ a_5'.$
$\texttt{(CFG}\ a_0\ a_1\ a_2\ a_3\ a_4\ a_5\ \texttt{= CFG}\ a_0'\ a_1'\ a_2'\ a_3'\ a_4'\ a_5'\texttt{)} \iff$
$\texttt{(}a_0\ \texttt{=}\ a_0'\texttt{)} \wedge \texttt{(}a_1\ \texttt{=}\ a_1'\texttt{)} \wedge \texttt{(}a_2\ \texttt{=}\ a_2'\texttt{)} \wedge \texttt{(}a_3\ \texttt{=}\ a_3'\texttt{)} \wedge$
$\texttt{(}a_4\ \texttt{=}\ a_4'\texttt{)} \wedge \texttt{(}a_5\ \texttt{=}\ a_5'\texttt{)}$

[extractCommand_def]

$\vdash \texttt{extractCommand (}P\ \texttt{says prop (SOME}\ cmd\texttt{))} = cmd$

[extractCommand_ind]

$\vdash \forall\, P'.$
$(\forall\, P\ cmd.\ P'\ \texttt{(}P\ \texttt{says prop (SOME}\ cmd\texttt{)))} \wedge P'\ \texttt{TT} \wedge P'\ \texttt{FF} \wedge$
$(\forall\, v_1.\ P'\ \texttt{(prop}\ v_1\texttt{))} \wedge (\forall\, v_3.\ P'\ \texttt{(notf}\ v_3\texttt{))} \wedge$
$(\forall\, v_6\ v_7.\ P'\ \texttt{(}v_6\ \texttt{andf}\ v_7\texttt{))} \wedge (\forall\, v_{10}\ v_{11}.\ P'\ \texttt{(}v_{10}\ \texttt{orf}\ v_{11}\texttt{))} \wedge$
$(\forall\, v_{14}\ v_{15}.\ P'\ \texttt{(}v_{14}\ \texttt{impf}\ v_{15}\texttt{))} \wedge$
$(\forall\, v_{18}\ v_{19}.\ P'\ \texttt{(}v_{18}\ \texttt{eqf}\ v_{19}\texttt{))} \wedge (\forall\, \mathit{v129}.\ P'\ \texttt{(}\mathit{v129}\ \texttt{says TT))} \wedge$
$(\forall\, \mathit{v130}.\ P'\ \texttt{(}\mathit{v130}\ \texttt{says FF))} \wedge$
$(\forall\, \mathit{v132}.\ P'\ \texttt{(}\mathit{v132}\ \texttt{says prop NONE))} \wedge$
$(\forall\, \mathit{v133}\ v_{66}.\ P'\ \texttt{(}\mathit{v133}\ \texttt{says notf}\ v_{66}\texttt{))} \wedge$
$(\forall\, \mathit{v134}\ v_{69}\ v_{70}.\ P'\ \texttt{(}\mathit{v134}\ \texttt{says (}v_{69}\ \texttt{andf}\ v_{70}\texttt{)))} \wedge$
$(\forall\, \mathit{v135}\ v_{73}\ v_{74}.\ P'\ \texttt{(}\mathit{v135}\ \texttt{says (}v_{73}\ \texttt{orf}\ v_{74}\texttt{)))} \wedge$
$(\forall\, \mathit{v136}\ v_{77}\ v_{78}.\ P'\ \texttt{(}\mathit{v136}\ \texttt{says (}v_{77}\ \texttt{impf}\ v_{78}\texttt{)))} \wedge$
$(\forall\, \mathit{v137}\ v_{81}\ v_{82}.\ P'\ \texttt{(}\mathit{v137}\ \texttt{says (}v_{81}\ \texttt{eqf}\ v_{82}\texttt{)))} \wedge$
$(\forall\, \mathit{v138}\ v_{85}\ v_{86}.\ P'\ \texttt{(}\mathit{v138}\ \texttt{says}\ v_{85}\ \texttt{says}\ v_{86}\texttt{))} \wedge$
$(\forall\, \mathit{v139}\ v_{89}\ v_{90}.\ P'\ \texttt{(}\mathit{v139}\ \texttt{says}\ v_{89}\ \texttt{speaks\_for}\ v_{90}\texttt{))} \wedge$
$(\forall\, \mathit{v140}\ v_{93}\ v_{94}.\ P'\ \texttt{(}\mathit{v140}\ \texttt{says}\ v_{93}\ \texttt{controls}\ v_{94}\texttt{))} \wedge$
$(\forall\, \mathit{v141}\ v_{98}\ v_{99}\ \mathit{v100}.\ P'\ \texttt{(}\mathit{v141}\ \texttt{says reps}\ v_{98}\ v_{99}\ \mathit{v100}\texttt{))} \wedge$
$(\forall\, \mathit{v142}\ \mathit{v103}\ \mathit{v104}.\ P'\ \texttt{(}\mathit{v142}\ \texttt{says}\ \mathit{v103}\ \texttt{domi}\ \mathit{v104}\texttt{))} \wedge$
$(\forall\, \mathit{v143}\ \mathit{v107}\ \mathit{v108}.\ P'\ \texttt{(}\mathit{v143}\ \texttt{says}\ \mathit{v107}\ \texttt{eqi}\ \mathit{v108}\texttt{))} \wedge$
$(\forall\, \mathit{v144}\ \mathit{v111}\ \mathit{v112}.\ P'\ \texttt{(}\mathit{v144}\ \texttt{says}\ \mathit{v111}\ \texttt{doms}\ \mathit{v112}\texttt{))} \wedge$
$(\forall\, \mathit{v145}\ \mathit{v115}\ \mathit{v116}.\ P'\ \texttt{(}\mathit{v145}\ \texttt{says}\ \mathit{v115}\ \texttt{eqs}\ \mathit{v116}\texttt{))} \wedge$
$(\forall\, \mathit{v146}\ \mathit{v119}\ \mathit{v120}.\ P'\ \texttt{(}\mathit{v146}\ \texttt{says}\ \mathit{v119}\ \texttt{eqn}\ \mathit{v120}\texttt{))} \wedge$
$(\forall\, \mathit{v147}\ \mathit{v123}\ \mathit{v124}.\ P'\ \texttt{(}\mathit{v147}\ \texttt{says}\ \mathit{v123}\ \texttt{lte}\ \mathit{v124}\texttt{))} \wedge$
$(\forall\, \mathit{v148}\ \mathit{v127}\ \mathit{v128}.\ P'\ \texttt{(}\mathit{v148}\ \texttt{says}\ \mathit{v127}\ \texttt{lt}\ \mathit{v128}\texttt{))} \wedge$
$(\forall\, v_{24}\ v_{25}.\ P'\ \texttt{(}v_{24}\ \texttt{speaks\_for}\ v_{25}\texttt{))} \wedge$
$(\forall\, v_{28}\ v_{29}.\ P'\ \texttt{(}v_{28}\ \texttt{controls}\ v_{29}\texttt{))} \wedge$
$(\forall\, v_{33}\ v_{34}\ v_{35}.\ P'\ \texttt{(reps}\ v_{33}\ v_{34}\ v_{35}\texttt{))} \wedge$
$(\forall\, v_{38}\ v_{39}.\ P'\ \texttt{(}v_{38}\ \texttt{domi}\ v_{39}\texttt{))} \wedge$
$(\forall\, v_{42}\ v_{43}.\ P'\ \texttt{(}v_{42}\ \texttt{eqi}\ v_{43}\texttt{))} \wedge$
$(\forall\, v_{46}\ v_{47}.\ P'\ \texttt{(}v_{46}\ \texttt{doms}\ v_{47}\texttt{))} \wedge$
$(\forall\, v_{50}\ v_{51}.\ P'\ \texttt{(}v_{50}\ \texttt{eqs}\ v_{51}\texttt{))} \wedge$
$(\forall\, v_{54}\ v_{55}.\ P'\ \texttt{(}v_{54}\ \texttt{eqn}\ v_{55}\texttt{))} \wedge$
$(\forall\, v_{58}\ v_{59}.\ P'\ \texttt{(}v_{58}\ \texttt{lte}\ v_{59}\texttt{))} \wedge$
$(\forall\, v_{62}\ v_{63}.\ P'\ \texttt{(}v_{62}\ \texttt{lt}\ v_{63}\texttt{))} \Rightarrow$
$\forall\, v.\ P'\ v$

[extractInput_def]

$\vdash \texttt{extractInput (}P\ \texttt{says prop}\ x\texttt{)} = x$

[extractInput_ind]

$\vdash \forall\, P'.$
$(\forall\, P\ x.\ P'\ \texttt{(}P\ \texttt{says prop}\ x\texttt{))} \wedge P'\ \texttt{TT} \wedge P'\ \texttt{FF} \wedge$
$(\forall\, v_1.\ P'\ \texttt{(prop}\ v_1\texttt{))} \wedge (\forall\, v_3.\ P'\ \texttt{(notf}\ v_3\texttt{))} \wedge$
$(\forall\, v_6\ v_7.\ P'\ \texttt{(}v_6\ \texttt{andf}\ v_7\texttt{))} \wedge (\forall\, v_{10}\ v_{11}.\ P'\ \texttt{(}v_{10}\ \texttt{orf}\ v_{11}\texttt{))} \wedge$
$(\forall\, v_{14}\ v_{15}.\ P'\ \texttt{(}v_{14}\ \texttt{impf}\ v_{15}\texttt{))} \wedge$
$(\forall\, v_{18}\ v_{19}.\ P'\ \texttt{(}v_{18}\ \texttt{eqf}\ v_{19}\texttt{))} \wedge (\forall\, \mathit{v129}.\ P'\ \texttt{(}\mathit{v129}\ \texttt{says TT))} \wedge$
$(\forall\, \mathit{v130}.\ P'\ \texttt{(}\mathit{v130}\ \texttt{says FF))} \wedge$

```
    (∀ v131 v66. P' (v131 says notf v66)) ∧
    (∀ v132 v69 v70. P' (v132 says (v69 andf v70))) ∧
    (∀ v133 v73 v74. P' (v133 says (v73 orf v74))) ∧
    (∀ v134 v77 v78. P' (v134 says (v77 impf v78))) ∧
    (∀ v135 v81 v82. P' (v135 says (v81 eqf v82))) ∧
    (∀ v136 v85 v86. P' (v136 says v85 says v86)) ∧
    (∀ v137 v89 v90. P' (v137 says v89 speaks_for v90)) ∧
    (∀ v138 v93 v94. P' (v138 says v93 controls v94)) ∧
    (∀ v139 v98 v99 v100. P' (v139 says reps v98 v99 v100)) ∧
    (∀ v140 v103 v104. P' (v140 says v103 domi v104)) ∧
    (∀ v141 v107 v108. P' (v141 says v107 eqi v108)) ∧
    (∀ v142 v111 v112. P' (v142 says v111 doms v112)) ∧
    (∀ v143 v115 v116. P' (v143 says v115 eqs v116)) ∧
    (∀ v144 v119 v120. P' (v144 says v119 eqn v120)) ∧
    (∀ v145 v123 v124. P' (v145 says v123 lte v124)) ∧
    (∀ v146 v127 v128. P' (v146 says v127 lt v128)) ∧
    (∀ v24 v25. P' (v24 speaks_for v25)) ∧
    (∀ v28 v29. P' (v28 controls v29)) ∧
    (∀ v33 v34 v35. P' (reps v33 v34 v35)) ∧
    (∀ v38 v39. P' (v38 domi v39)) ∧
    (∀ v42 v43. P' (v42 eqi v43)) ∧
    (∀ v46 v47. P' (v46 doms v47)) ∧
    (∀ v50 v51. P' (v50 eqs v51)) ∧
    (∀ v54 v55. P' (v54 eqn v55)) ∧
    (∀ v58 v59. P' (v58 lte v59)) ∧
    (∀ v62 v63. P' (v62 lt v63)) ⇒
    ∀ v. P' v
```

[extractPropCommand_def]

```
⊢ extractPropCommand (P says prop (SOME cmd)) = prop (SOME cmd)
```

[extractPropCommand_ind]

```
⊢ ∀ P'.
    (∀ P cmd. P' (P says prop (SOME cmd))) ∧ P' TT ∧ P' FF ∧
    (∀ v1. P' (prop v1)) ∧ (∀ v3. P' (notf v3)) ∧
    (∀ v6 v7. P' (v6 andf v7)) ∧ (∀ v10 v11. P' (v10 orf v11)) ∧
    (∀ v14 v15. P' (v14 impf v15)) ∧
    (∀ v18 v19. P' (v18 eqf v19)) ∧ (∀ v129. P' (v129 says TT)) ∧
    (∀ v130. P' (v130 says FF)) ∧
    (∀ v132. P' (v132 says prop NONE)) ∧
    (∀ v133 v66. P' (v133 says notf v66)) ∧
    (∀ v134 v69 v70. P' (v134 says (v69 andf v70))) ∧
    (∀ v135 v73 v74. P' (v135 says (v73 orf v74))) ∧
    (∀ v136 v77 v78. P' (v136 says (v77 impf v78))) ∧
    (∀ v137 v81 v82. P' (v137 says (v81 eqf v82))) ∧
    (∀ v138 v85 v86. P' (v138 says v85 says v86)) ∧
    (∀ v139 v89 v90. P' (v139 says v89 speaks_for v90)) ∧
    (∀ v140 v93 v94. P' (v140 says v93 controls v94)) ∧
    (∀ v141 v98 v99 v100. P' (v141 says reps v98 v99 v100)) ∧
    (∀ v142 v103 v104. P' (v142 says v103 domi v104)) ∧
    (∀ v143 v107 v108. P' (v143 says v107 eqi v108)) ∧
    (∀ v144 v111 v112. P' (v144 says v111 doms v112)) ∧
    (∀ v145 v115 v116. P' (v145 says v115 eqs v116)) ∧
    (∀ v146 v119 v120. P' (v146 says v119 eqn v120)) ∧
    (∀ v147 v123 v124. P' (v147 says v123 lte v124)) ∧
    (∀ v148 v127 v128. P' (v148 says v127 lt v128)) ∧
    (∀ v24 v25. P' (v24 speaks_for v25)) ∧
    (∀ v28 v29. P' (v28 controls v29)) ∧
```

```
(∀ v33 v34 v35. P' (reps v33 v34 v35)) ∧
(∀ v38 v39. P' (v38 domi v39)) ∧
(∀ v42 v43. P' (v42 eqi v43)) ∧
(∀ v46 v47. P' (v46 doms v47)) ∧
(∀ v50 v51. P' (v50 eqs v51)) ∧
(∀ v54 v55. P' (v54 eqn v55)) ∧
(∀ v58 v59. P' (v58 lte v59)) ∧
(∀ v62 v63. P' (v62 lt v63)) ⇒
∀ v. P' v
```

[rule0]

```
⊢ ∀ elementTest NS M Oi Os Out s context stateInterp x ins outs.
    authenticationTest elementTest x ∧
    CFGInterpret (M,Oi,Os)
      (CFG elementTest stateInterp context (x::ins) s outs) ⇒
    TR (M,Oi,Os) (exec (inputList x))
      (CFG elementTest stateInterp context (x::ins) s outs)
      (CFG elementTest stateInterp context ins
         (NS s (exec (inputList x)))
         (Out s (exec (inputList x))::outs))
```

[rule1]

```
⊢ ∀ elementTest NS M Oi Os Out s context stateInterp x ins outs.
    authenticationTest elementTest x ∧
    CFGInterpret (M,Oi,Os)
      (CFG elementTest stateInterp context (x::ins) s outs) ⇒
    TR (M,Oi,Os) (trap (inputList x))
      (CFG elementTest stateInterp context (x::ins) s outs)
      (CFG elementTest stateInterp context ins
         (NS s (trap (inputList x)))
         (Out s (trap (inputList x))::outs))
```

[rule2]

```
⊢ ∀ elementTest NS M Oi Os Out s context stateInterp x ins outs.
    ¬authenticationTest elementTest x ⇒
    TR (M,Oi,Os) (discard (inputList x))
      (CFG elementTest stateInterp context (x::ins) s outs)
      (CFG elementTest stateInterp context ins
         (NS s (discard (inputList x)))
         (Out s (discard (inputList x))::outs))
```

[TR_cases]

```
⊢ ∀ a0 a1 a2 a3.
    TR a0 a1 a2 a3 ⟺
    (∃ elementTest NS M Oi Os Out s context stateInterp x ins
       outs.
      (a0 = (M,Oi,Os)) ∧ (a1 = exec (inputList x)) ∧
      (a2 =
       CFG elementTest stateInterp context (x::ins) s outs) ∧
      (a3 =
       CFG elementTest stateInterp context ins
         (NS s (exec (inputList x)))
         (Out s (exec (inputList x))::outs)) ∧
      authenticationTest elementTest x ∧
      CFGInterpret (M,Oi,Os)
        (CFG elementTest stateInterp context (x::ins) s
           outs)) ∨
```

```
      (∃ elementTest NS M Oi Os Out s context stateInterp x ins
          outs.
         (a0 = (M,Oi,Os)) ∧ (a1 = trap (inputList x)) ∧
         (a2 =
          CFG elementTest stateInterp context (x::ins) s outs) ∧
         (a3 =
          CFG elementTest stateInterp context ins
            (NS s (trap (inputList x)))
            (Out s (trap (inputList x))::outs)) ∧
         authenticationTest elementTest x ∧
         CFGInterpret (M,Oi,Os)
           (CFG elementTest stateInterp context (x::ins) s
              outs)) ∨
      ∃ elementTest NS M Oi Os Out s context stateInterp x ins
          outs.
        (a0 = (M,Oi,Os)) ∧ (a1 = discard (inputList x)) ∧
        (a2 =
         CFG elementTest stateInterp context (x::ins) s outs) ∧
        (a3 =
         CFG elementTest stateInterp context ins
           (NS s (discard (inputList x)))
           (Out s (discard (inputList x))::outs)) ∧
        ¬authenticationTest elementTest x
```

[TR_discard_cmd_rule]

```
 ⊢ TR (M,Oi,Os) (discard (inputList x))
     (CFG elementTest stateInterp context (x::ins) s outs)
     (CFG elementTest stateInterp context ins
        (NS s (discard (inputList x)))
        (Out s (discard (inputList x))::outs)) ⟺
   ¬authenticationTest elementTest x
```

[TR_EQ_rules_thm]

```
 ⊢ (TR (M,Oi,Os) (exec (inputList x))
      (CFG elementTest stateInterp context (x::ins) s outs)
      (CFG elementTest stateInterp context ins
         (NS s (exec (inputList x)))
         (Out s (exec (inputList x))::outs)) ⟺
    authenticationTest elementTest x ∧
    CFGInterpret (M,Oi,Os)
      (CFG elementTest stateInterp context (x::ins) s outs)) ∧
   (TR (M,Oi,Os) (trap (inputList x))
      (CFG elementTest stateInterp context (x::ins) s outs)
      (CFG elementTest stateInterp context ins
         (NS s (trap (inputList x)))
         (Out s (trap (inputList x))::outs)) ⟺
    authenticationTest elementTest x ∧
    CFGInterpret (M,Oi,Os)
      (CFG elementTest stateInterp context (x::ins) s outs)) ∧
   (TR (M,Oi,Os) (discard (inputList x))
      (CFG elementTest stateInterp context (x::ins) s outs)
      (CFG elementTest stateInterp context ins
         (NS s (discard (inputList x)))
         (Out s (discard (inputList x))::outs)) ⟺
    ¬authenticationTest elementTest x)
```

[TR_exec_cmd_rule]

```
⊢ ∀ elementTest context stateInterp x ins s outs.
    (∀ M Oi Os.
       CFGInterpret (M, Oi, Os)
         (CFG elementTest stateInterp context (x::ins) s
            outs) ⇒
       (M, Oi, Os) satList propCommandList x) ⇒
    ∀ NS Out M Oi Os.
      TR (M, Oi, Os) (exec (inputList x))
        (CFG elementTest stateInterp context (x::ins) s outs)
        (CFG elementTest stateInterp context ins
           (NS s (exec (inputList x)))
           (Out s (exec (inputList x))::outs)) ⟺
      authenticationTest elementTest x ∧
      CFGInterpret (M, Oi, Os)
        (CFG elementTest stateInterp context (x::ins) s outs) ∧
      (M, Oi, Os) satList propCommandList x
```

[TR_ind]

```
⊢ ∀ TR'.
    (∀ elementTest NS M Oi Os Out s context stateInterp x ins
       outs.
       authenticationTest elementTest x ∧
       CFGInterpret (M, Oi, Os)
         (CFG elementTest stateInterp context (x::ins) s
            outs) ⇒
       TR' (M, Oi, Os) (exec (inputList x))
         (CFG elementTest stateInterp context (x::ins) s outs)
         (CFG elementTest stateInterp context ins
            (NS s (exec (inputList x)))
            (Out s (exec (inputList x))::outs))) ∧
    (∀ elementTest NS M Oi Os Out s context stateInterp x ins
       outs.
       authenticationTest elementTest x ∧
       CFGInterpret (M, Oi, Os)
         (CFG elementTest stateInterp context (x::ins) s
            outs) ⇒
       TR' (M, Oi, Os) (trap (inputList x))
         (CFG elementTest stateInterp context (x::ins) s outs)
         (CFG elementTest stateInterp context ins
            (NS s (trap (inputList x)))
            (Out s (trap (inputList x))::outs))) ∧
    (∀ elementTest NS M Oi Os Out s context stateInterp x ins
       outs.
       ¬authenticationTest elementTest x ⇒
       TR' (M, Oi, Os) (discard (inputList x))
         (CFG elementTest stateInterp context (x::ins) s outs)
         (CFG elementTest stateInterp context ins
            (NS s (discard (inputList x)))
            (Out s (discard (inputList x))::outs))) ⇒
    ∀ a0 a1 a2 a3. TR a0 a1 a2 a3 ⇒ TR' a0 a1 a2 a3
```

[TR_rules]

```
⊢ (∀ elementTest NS M Oi Os Out s context stateInterp x ins
     outs.
     authenticationTest elementTest x ∧
     CFGInterpret (M, Oi, Os)
       (CFG elementTest stateInterp context (x::ins) s outs) ⇒
     TR (M, Oi, Os) (exec (inputList x))
```

```
      (CFG elementTest stateInterp context (x::ins) s outs)
      (CFG elementTest stateInterp context ins
         (NS s (exec (inputList x)))
         (Out s (exec (inputList x))::outs))) ∧
  (∀ elementTest NS M Oi Os Out s context stateInterp x ins
     outs.
    authenticationTest elementTest x ∧
    CFGInterpret (M,Oi,Os)
      (CFG elementTest stateInterp context (x::ins) s outs) ⇒
    TR (M,Oi,Os) (trap (inputList x))
      (CFG elementTest stateInterp context (x::ins) s outs)
      (CFG elementTest stateInterp context ins
         (NS s (trap (inputList x)))
         (Out s (trap (inputList x))::outs))) ∧
  ∀ elementTest NS M Oi Os Out s context stateInterp x ins outs.
    ¬authenticationTest elementTest x ⇒
    TR (M,Oi,Os) (discard (inputList x))
      (CFG elementTest stateInterp context (x::ins) s outs)
      (CFG elementTest stateInterp context ins
         (NS s (discard (inputList x)))
         (Out s (discard (inputList x))::outs))
```

[TR_strongind]

```
⊢ ∀ TR'.
  (∀ elementTest NS M Oi Os Out s context stateInterp x ins
     outs.
    authenticationTest elementTest x ∧
    CFGInterpret (M,Oi,Os)
      (CFG elementTest stateInterp context (x::ins) s
         outs) ⇒
    TR' (M,Oi,Os) (exec (inputList x))
      (CFG elementTest stateInterp context (x::ins) s outs)
      (CFG elementTest stateInterp context ins
         (NS s (exec (inputList x)))
         (Out s (exec (inputList x))::outs))) ∧
  (∀ elementTest NS M Oi Os Out s context stateInterp x ins
     outs.
    authenticationTest elementTest x ∧
    CFGInterpret (M,Oi,Os)
      (CFG elementTest stateInterp context (x::ins) s
         outs) ⇒
    TR' (M,Oi,Os) (trap (inputList x))
      (CFG elementTest stateInterp context (x::ins) s outs)
      (CFG elementTest stateInterp context ins
         (NS s (trap (inputList x)))
         (Out s (trap (inputList x))::outs))) ∧
  (∀ elementTest NS M Oi Os Out s context stateInterp x ins
     outs.
    ¬authenticationTest elementTest x ⇒
    TR' (M,Oi,Os) (discard (inputList x))
      (CFG elementTest stateInterp context (x::ins) s outs)
      (CFG elementTest stateInterp context ins
         (NS s (discard (inputList x)))
         (Out s (discard (inputList x))::outs))) ⇒
  ∀ a0 a1 a2 a3. TR a0 a1 a2 a3 ⇒ TR' a0 a1 a2 a3
```

[TR_trap_cmd_rule]

```
⊢ ∀ elementTest context stateInterp x ins s outs.
  (∀ M Oi Os.
```

```
        CFGInterpret (M,Oi,Os)
          (CFG elementTest stateInterp context (x::ins) s
             outs) ⇒
        (M,Oi,Os) sat prop NONE) ⇒
    ∀NS Out M Oi Os.
      TR (M,Oi,Os) (trap (inputList x))
        (CFG elementTest stateInterp context (x::ins) s outs)
        (CFG elementTest stateInterp context ins
           (NS s (trap (inputList x)))
           (Out s (trap (inputList x))::outs)) ⟺
      authenticationTest elementTest x ∧
      CFGInterpret (M,Oi,Os)
        (CFG elementTest stateInterp context (x::ins) s outs) ∧
      (M,Oi,Os) sat prop NONE
```

[TRrule0]

```
⊢ TR (M,Oi,Os) (exec (inputList x))
    (CFG elementTest stateInterp context (x::ins) s outs)
    (CFG elementTest stateInterp context ins
       (NS s (exec (inputList x)))
       (Out s (exec (inputList x))::outs)) ⟺
  authenticationTest elementTest x ∧
  CFGInterpret (M,Oi,Os)
    (CFG elementTest stateInterp context (x::ins) s outs)
```

[TRrule1]

```
⊢ TR (M,Oi,Os) (trap (inputList x))
    (CFG elementTest stateInterp context (x::ins) s outs)
    (CFG elementTest stateInterp context ins
       (NS s (trap (inputList x)))
       (Out s (trap (inputList x))::outs)) ⟺
  authenticationTest elementTest x ∧
  CFGInterpret (M,Oi,Os)
    (CFG elementTest stateInterp context (x::ins) s outs)
```

[trType_distinct_clauses]

```
⊢ (∀a' a. discard a ≠ trap a') ∧ (∀a' a. discard a ≠ exec a') ∧
  ∀a' a. trap a ≠ exec a'
```

[trType_one_one]

```
⊢ (∀a a'. (discard a = discard a') ⟺ (a = a')) ∧
  (∀a a'. (trap a = trap a') ⟺ (a = a')) ∧
  ∀a a'. (exec a = exec a') ⟺ (a = a')
```

B.2 satList Theory

Built: 11 February 2018
Parent Theories: aclDrules

B.2.1 Definitions

[satList_def]

```
⊢ ∀M Oi Os formList.
    (M,Oi,Os) satList formList ⟺
    FOLDR (λx y. x ∧ y) T (MAP (λf. (M,Oi,Os) sat f) formList)
```

B.2.2 Theorems

[satList_conj]

$\vdash \forall l_1\ l_2\ M\ Oi\ Os.$
(M, Oi, Os) satList $l_1 \wedge (M, Oi, Os)$ satList $l_2 \iff$
(M, Oi, Os) satList $(l_1$ ++ $l_2)$

[satList_CONS]

$\vdash \forall h\ t\ M\ Oi\ Os.$
(M, Oi, Os) satList $(h::t) \iff$
(M, Oi, Os) sat $h \wedge (M, Oi, Os)$ satList t

[satList_nil]

$\vdash (M, Oi, Os)$ satList []

B.3 principal Theory

Built: 16 March 2018
Parent Theories: cipher

B.3.1 Datatypes

authority = ca num

principal =
    Staff staff
  | Authority authority
  | Role role
  | KeyS (staff pKey)
  | KeyA (authority pKey)
  | C2
  | MunitionAvail
  | GPSKB
  | TimeKB

role = Commander | Operator

staff = Alice | Bob | Carol

B.3.2 Theorems

[authority_one_one]

$\vdash \forall a\ a'.$ (ca a = ca a') $\iff$ (a = a')

[principal_distinct_clauses]

$\vdash (\forall a'\ a.$ Staff $a \neq$ Authority $a') \wedge$
$(\forall a'\ a.$ Staff $a \neq$ Role $a') \wedge (\forall a'\ a.$ Staff $a \neq$ KeyS $a') \wedge$
$(\forall a'\ a.$ Staff $a \neq$ KeyA $a') \wedge (\forall a.$ Staff $a \neq$ C2) $\wedge$
$(\forall a.$ Staff $a \neq$ MunitionAvail) $\wedge (\forall a.$ Staff $a \neq$ GPSKB) $\wedge$
$(\forall a.$ Staff $a \neq$ TimeKB) $\wedge (\forall a'\ a.$ Authority $a \neq$ Role $a') \wedge$
$(\forall a'\ a.$ Authority $a \neq$ KeyS $a') \wedge$
$(\forall a'\ a.$ Authority $a \neq$ KeyA $a') \wedge (\forall a.$ Authority $a \neq$ C2) $\wedge$
$(\forall a.$ Authority $a \neq$ MunitionAvail) $\wedge$
$(\forall a.$ Authority $a \neq$ GPSKB) $\wedge (\forall a.$ Authority $a \neq$ TimeKB) $\wedge$
$(\forall a'\ a.$ Role $a \neq$ KeyS $a') \wedge (\forall a'\ a.$ Role $a \neq$ KeyA $a') \wedge$
$(\forall a.$ Role $a \neq$ C2) $\wedge (\forall a.$ Role $a \neq$ MunitionAvail) $\wedge$

$(\forall a.\ \texttt{Role}\ a \neq \texttt{GPSKB}) \wedge (\forall a.\ \texttt{Role}\ a \neq \texttt{TimeKB}) \wedge$
$(\forall a'\ a.\ \texttt{KeyS}\ a \neq \texttt{KeyA}\ a') \wedge (\forall a.\ \texttt{KeyS}\ a \neq \texttt{C2}) \wedge$
$(\forall a.\ \texttt{KeyS}\ a \neq \texttt{MunitionAvail}) \wedge (\forall a.\ \texttt{KeyS}\ a \neq \texttt{GPSKB}) \wedge$
$(\forall a.\ \texttt{KeyS}\ a \neq \texttt{TimeKB}) \wedge (\forall a.\ \texttt{KeyA}\ a \neq \texttt{C2}) \wedge$
$(\forall a.\ \texttt{KeyA}\ a \neq \texttt{MunitionAvail}) \wedge (\forall a.\ \texttt{KeyA}\ a \neq \texttt{GPSKB}) \wedge$
$(\forall a.\ \texttt{KeyA}\ a \neq \texttt{TimeKB}) \wedge \texttt{C2} \neq \texttt{MunitionAvail} \wedge \texttt{C2} \neq \texttt{GPSKB} \wedge$
$\texttt{C2} \neq \texttt{TimeKB} \wedge \texttt{MunitionAvail} \neq \texttt{GPSKB} \wedge$
$\texttt{MunitionAvail} \neq \texttt{TimeKB} \wedge \texttt{GPSKB} \neq \texttt{TimeKB}$

[principal_one_one]

$\vdash (\forall a\ a'.\ (\texttt{Staff}\ a = \texttt{Staff}\ a') \iff (a = a')) \wedge$
$(\forall a\ a'.\ (\texttt{Authority}\ a = \texttt{Authority}\ a') \iff (a = a')) \wedge$
$(\forall a\ a'.\ (\texttt{Role}\ a = \texttt{Role}\ a') \iff (a = a')) \wedge$
$(\forall a\ a'.\ (\texttt{KeyS}\ a = \texttt{KeyS}\ a') \iff (a = a')) \wedge$
$\forall a\ a'.\ (\texttt{KeyA}\ a = \texttt{KeyA}\ a') \iff (a = a')$

[role_distinct_clauses]

$\vdash \texttt{Commander} \neq \texttt{Operator}$

[staff_distinct_clauses]

$\vdash \texttt{Alice} \neq \texttt{Bob} \wedge \texttt{Alice} \neq \texttt{Carol} \wedge \texttt{Bob} \neq \texttt{Carol}$

B.4 uavTypes Theory

Built: 16 March 2018
Parent Theories: indexedLists, patternMatches

B.4.1 Datatypes

c3input = CMD ctrlAct | MA muniAvail | KBL bool | KBT bool

ctrlAct = RL | RR | RB

muniAvail = N | L | R | B

state = Off | LlRe | LeRl | LlRl | LeRe

B.4.2 Theorems

[c3input_distinct_clauses]

$\vdash (\forall a'\ a.\ \texttt{CMD}\ a \neq \texttt{MA}\ a') \wedge (\forall a'\ a.\ \texttt{CMD}\ a \neq \texttt{KBL}\ a') \wedge$
$(\forall a'\ a.\ \texttt{CMD}\ a \neq \texttt{KBT}\ a') \wedge (\forall a'\ a.\ \texttt{MA}\ a \neq \texttt{KBL}\ a') \wedge$
$(\forall a'\ a.\ \texttt{MA}\ a \neq \texttt{KBT}\ a') \wedge \forall a'\ a.\ \texttt{KBL}\ a \neq \texttt{KBT}\ a'$

[c3input_one_one]

$\vdash (\forall a\ a'.\ (\texttt{CMD}\ a = \texttt{CMD}\ a') \iff (a = a')) \wedge$
$(\forall a\ a'.\ (\texttt{MA}\ a = \texttt{MA}\ a') \iff (a = a')) \wedge$
$(\forall a\ a'.\ (\texttt{KBL}\ a = \texttt{KBL}\ a') \iff (a \iff a')) \wedge$
$\forall a\ a'.\ (\texttt{KBT}\ a = \texttt{KBT}\ a') \iff (a \iff a')$

[ctrlAct_distinct_clauses]

$\vdash \texttt{RL} \neq \texttt{RR} \wedge \texttt{RL} \neq \texttt{RB} \wedge \texttt{RR} \neq \texttt{RB}$

[muniAvail_distinct_clauses]

$\vdash \texttt{N} \neq \texttt{L} \wedge \texttt{N} \neq \texttt{R} \wedge \texttt{N} \neq \texttt{B} \wedge \texttt{L} \neq \texttt{R} \wedge \texttt{L} \neq \texttt{B} \wedge \texttt{R} \neq \texttt{B}$

[state_distinct_clauses]

$\vdash \texttt{Off} \neq \texttt{LlRe} \wedge \texttt{Off} \neq \texttt{LeRl} \wedge \texttt{Off} \neq \texttt{LlRl} \wedge \texttt{Off} \neq \texttt{LeRe} \wedge$
$\texttt{LlRe} \neq \texttt{LeRl} \wedge \texttt{LlRe} \neq \texttt{LlRl} \wedge \texttt{LlRe} \neq \texttt{LeRe} \wedge \texttt{LeRl} \neq \texttt{LlRl} \wedge$
$\texttt{LeRl} \neq \texttt{LeRe} \wedge \texttt{LlRl} \neq \texttt{LeRe}$

B.5 uavDef Theory

Built: 11 February 2018
Parent Theories: uavTypes, ssm1

B.5.1 Theorems

[discard_out_safe_thm]

$\vdash$ uavMOout s (discard x) = exec [NONE]

[discard_safe_thm]

$\vdash \forall state\ x.$ uavMOns $state$ (discard x) = $state$

[exec_hca_out_thm]

$\vdash \forall hca\ s\ x.$
(uavMOout s (exec x) = exec [SOME (CMD hca)]) $\iff$
(hca = RL) $\wedge$
((s = LlRe) $\wedge$ (getCMD x = SOME (CMD RL)) $\wedge$
(getMA x = SOME (MA L)) $\wedge$ (getKBL x = SOME (KBL T)) $\wedge$
(getKBT x = SOME (KBT T)) $\vee$
(s = LlRl) $\wedge$ (getCMD x = SOME (CMD RL)) $\wedge$
(getMA x = SOME (MA B)) $\wedge$ (getKBL x = SOME (KBL T)) $\wedge$
(getKBT x = SOME (KBT T))) $\vee$
(hca = RR) $\wedge$
((s = LeRl) $\wedge$ (getCMD x = SOME (CMD RR)) $\wedge$
(getMA x = SOME (MA R)) $\wedge$ (getKBL x = SOME (KBL T)) $\wedge$
(getKBT x = SOME (KBT T)) $\vee$
(s = LlRl) $\wedge$ (getCMD x = SOME (CMD RR)) $\wedge$
(getMA x = SOME (MA B)) $\wedge$ (getKBL x = SOME (KBL T)) $\wedge$
(getKBT x = SOME (KBT T))) $\vee$
(hca = RB) $\wedge$ (s = LlRl) $\wedge$ (getCMD x = SOME (CMD RB)) $\wedge$
(getMA x = SOME (MA B)) $\wedge$ (getKBL x = SOME (KBL T)) $\wedge$
(getKBT x = SOME (KBT T))

[exec_RB_out_thm]

$\vdash \forall s\ x'.$
(uavMOout s (exec x) = exec [SOME (CMD RB)]) $\iff$
(s = LlRl) $\wedge$ (getCMD x = SOME (CMD RB)) $\wedge$
(getMA x = SOME (MA B)) $\wedge$ (getKBL x = SOME (KBL T)) $\wedge$
(getKBT x = SOME (KBT T))

[exec_RL_out_thm]

$\vdash \forall s\ x'.$
(uavMOout s (exec x) = exec [SOME (CMD RL)]) $\iff$
(s = LlRe) $\wedge$ (getCMD x = SOME (CMD RL)) $\wedge$
(getMA x = SOME (MA L)) $\wedge$ (getKBL x = SOME (KBL T)) $\wedge$
(getKBT x = SOME (KBT T)) $\vee$
(s = LlRl) $\wedge$ (getCMD x = SOME (CMD RL)) $\wedge$
(getMA x = SOME (MA B)) $\wedge$ (getKBL x = SOME (KBL T)) $\wedge$
(getKBT x = SOME (KBT T))

[exec_RR_out_thm]

$\vdash \forall s\ x'.$
(uavMOout s (exec x) = exec [SOME (CMD RR)]) $\iff$
(s = LeRl) $\wedge$ (getCMD x = SOME (CMD RR)) $\wedge$
(getMA x = SOME (MA R)) $\wedge$ (getKBL x = SOME (KBL T)) $\wedge$

(getKBT x = SOME (KBT T)) ∨
(s = LlRl) ∧ (getCMD x = SOME (CMD RR)) ∧
(getMA x = SOME (MA B)) ∧ (getKBL x = SOME (KBL T)) ∧
(getKBT x = SOME (KBT T))

[exec_state_change_thm]
⊢ ∀s x.
s ≠ uavMOns s (exec x) ⟺
((s = Off) ∧ (getMA x = SOME (MA N)) ∨
(s = Off) ∧ (getMA x = SOME (MA L)) ∨
(s = Off) ∧ (getMA x = SOME (MA R)) ∨
(s = Off) ∧ (getMA x = SOME (MA B))) ∨
(s = LlRe) ∧ (getCMD x = SOME (CMD RL)) ∧
(getMA x = SOME (MA L)) ∧ (getKBL x = SOME (KBL T)) ∧
(getKBT x = SOME (KBT T)) ∨
(s = LeRl) ∧ (getCMD x = SOME (CMD RR)) ∧
(getMA x = SOME (MA R)) ∧ (getKBL x = SOME (KBL T)) ∧
(getKBT x = SOME (KBT T)) ∨
(s = LlRl) ∧ (getCMD x = SOME (CMD RL)) ∧
(getMA x = SOME (MA B)) ∧ (getKBL x = SOME (KBL T)) ∧
(getKBT x = SOME (KBT T)) ∨
(s = LlRl) ∧ (getCMD x = SOME (CMD RR)) ∧
(getMA x = SOME (MA B)) ∧ (getKBL x = SOME (KBL T)) ∧
(getKBT x = SOME (KBT T)) ∨
(s = LlRl) ∧ (getCMD x = SOME (CMD RB)) ∧
(getMA x = SOME (MA B)) ∧ (getKBL x = SOME (KBL T)) ∧
(getKBT x = SOME (KBT T))

[getCMD_def]
⊢ (getCMD [] = NONE) ∧
(∀xs cmd. getCMD (SOME (CMD cmd)::xs) = SOME (CMD cmd)) ∧
(∀xs ma. getCMD (SOME (MA ma)::xs) = getCMD xs) ∧
(∀xs loc. getCMD (SOME (KBL loc)::xs) = getCMD xs) ∧
∀xs $time$. getCMD (SOME (KBT $time$)::xs) = getCMD xs

[getCMD_ind]
⊢ ∀P.
P [] ∧ (∀cmd xs. P (SOME (CMD cmd)::xs)) ∧
(∀ma xs. P xs ⇒ P (SOME (MA ma)::xs)) ∧
(∀loc xs. P xs ⇒ P (SOME (KBL loc)::xs)) ∧
(∀$time$ xs. P xs ⇒ P (SOME (KBT $time$)::xs)) ∧
(∀v_3. P (NONE::v_3)) ⇒
∀v. P v

[getKBL_def]
⊢ (getKBL [] = NONE) ∧
(∀xs loc. getKBL (SOME (KBL loc)::xs) = SOME (KBL loc)) ∧
(∀xs cmd. getKBL (SOME (CMD cmd)::xs) = getKBL xs) ∧
(∀xs ma. getKBL (SOME (MA ma)::xs) = getKBL xs) ∧
∀xs $time$. getKBL (SOME (KBT $time$)::xs) = getKBL xs

[getKBL_ind]
⊢ ∀P.
P [] ∧ (∀loc xs. P (SOME (KBL loc)::xs)) ∧
(∀cmd xs. P xs ⇒ P (SOME (CMD cmd)::xs)) ∧
(∀ma xs. P xs ⇒ P (SOME (MA ma)::xs)) ∧
(∀$time$ xs. P xs ⇒ P (SOME (KBT $time$)::xs)) ∧
(∀v_3. P (NONE::v_3)) ⇒
∀v. P v

[getKBT_def]

```
⊢ (getKBT [] = NONE) ∧
  (∀xs time. getKBT (SOME (KBT time)::xs) = SOME (KBT time)) ∧
  (∀xs cmd. getKBT (SOME (CMD cmd)::xs) = getKBT xs) ∧
  (∀xs ma. getKBT (SOME (MA ma)::xs) = getKBT xs) ∧
  ∀xs loc. getKBT (SOME (KBL loc)::xs) = getKBT xs
```

[getKBT_ind]

```
⊢ ∀P.
    P [] ∧ (∀time xs. P (SOME (KBT time)::xs)) ∧
    (∀cmd xs. P xs ⇒ P (SOME (CMD cmd)::xs)) ∧
    (∀ma xs. P xs ⇒ P (SOME (MA ma)::xs)) ∧
    (∀loc xs. P xs ⇒ P (SOME (KBL loc)::xs)) ∧
    (∀v3. P (NONE::v3)) ⇒
    ∀v. P v
```

[getMA_def]

```
⊢ (getMA [] = NONE) ∧
  (∀xs ma. getMA (SOME (MA ma)::xs) = SOME (MA ma)) ∧
  (∀xs cmd. getMA (SOME (CMD cmd)::xs) = getMA xs) ∧
  (∀xs loc. getMA (SOME (KBL loc)::xs) = getMA xs) ∧
  ∀xs time. getMA (SOME (KBT time)::xs) = getMA xs
```

[getMA_ind]

```
⊢ ∀P.
    P [] ∧ (∀ma xs. P (SOME (MA ma)::xs)) ∧
    (∀cmd xs. P xs ⇒ P (SOME (CMD cmd)::xs)) ∧
    (∀loc xs. P xs ⇒ P (SOME (KBL loc)::xs)) ∧
    (∀time xs. P xs ⇒ P (SOME (KBT time)::xs)) ∧
    (∀v3. P (NONE::v3)) ⇒
    ∀v. P v
```

[trap_out_safe_thm]

```
⊢ uavMOout s (trap x) = exec [NONE]
```

[trap_safe_thm]

```
⊢ ∀state x. uavMOns state (trap x) = state
```

[uavMOns_def]

```
⊢ (uavMOns Off (exec x) =
   if getMA x = SOME (MA N) then LeRe
   else if getMA x = SOME (MA L) then LlRe
   else if getMA x = SOME (MA R) then LeRl
   else if getMA x = SOME (MA B) then LlRl
   else Off) ∧
  (uavMOns LlRe (exec x) =
   if
     (getCMD x = SOME (CMD RL)) ∧ (getMA x = SOME (MA L)) ∧
     (getKBL x = SOME (KBL T)) ∧ (getKBT x = SOME (KBT T))
   then
     LeRe
   else LlRe) ∧
  (uavMOns LeRl (exec x) =
   if
     (getCMD x = SOME (CMD RR)) ∧ (getMA x = SOME (MA R)) ∧
     (getKBL x = SOME (KBL T)) ∧ (getKBT x = SOME (KBT T))
   then
```

```
    LeRe
  else LeRl) ∧
 (uavM0ns LlRl (exec x) =
  if
    (getCMD x = SOME (CMD RL)) ∧ (getMA x = SOME (MA B)) ∧
    (getKBL x = SOME (KBL T)) ∧ (getKBT x = SOME (KBT T))
  then
    LeRl
  else if
    (getCMD x = SOME (CMD RR)) ∧ (getMA x = SOME (MA B)) ∧
    (getKBL x = SOME (KBL T)) ∧ (getKBT x = SOME (KBT T))
  then
    LlRe
  else if
    (getCMD x = SOME (CMD RB)) ∧ (getMA x = SOME (MA B)) ∧
    (getKBL x = SOME (KBL T)) ∧ (getKBT x = SOME (KBT T))
  then
    LeRe
  else LlRl) ∧ (uavM0ns LeRe (exec v0) = LeRe) ∧
 (uavM0ns s (trap v1) = s) ∧ (uavM0ns s (discard v2) = s)
```

[uavM0ns_ind]

```
⊢ ∀P.
    (∀x. P Off (exec x)) ∧ (∀x. P LlRe (exec x)) ∧
    (∀x. P LeRl (exec x)) ∧ (∀x. P LlRl (exec x)) ∧
    (∀v0. P LeRe (exec v0)) ∧ (∀s v1. P s (trap v1)) ∧
    (∀s v2. P s (discard v2)) ⇒
    ∀v v1. P v v1
```

[uavM0out_def]

```
⊢ (uavM0out Off (exec x) = exec []) ∧
  (uavM0out LlRe (exec x) =
   if
     (getCMD x = SOME (CMD RL)) ∧ (getMA x = SOME (MA L)) ∧
     (getKBL x = SOME (KBL T)) ∧ (getKBT x = SOME (KBT T))
   then
     exec [SOME (CMD RL)]
   else exec [NONE]) ∧
  (uavM0out LeRl (exec x) =
   if
     (getCMD x = SOME (CMD RR)) ∧ (getMA x = SOME (MA R)) ∧
     (getKBL x = SOME (KBL T)) ∧ (getKBT x = SOME (KBT T))
   then
     exec [SOME (CMD RR)]
   else exec [NONE]) ∧
  (uavM0out LlRl (exec x) =
   if
     (getCMD x = SOME (CMD RL)) ∧ (getMA x = SOME (MA B)) ∧
     (getKBL x = SOME (KBL T)) ∧ (getKBT x = SOME (KBT T))
   then
     exec [SOME (CMD RL)]
   else if
     (getCMD x = SOME (CMD RR)) ∧ (getMA x = SOME (MA B)) ∧
     (getKBL x = SOME (KBL T)) ∧ (getKBT x = SOME (KBT T))
   then
     exec [SOME (CMD RR)]
   else if
     (getCMD x = SOME (CMD RB)) ∧ (getMA x = SOME (MA B)) ∧
```

```
      (getKBL x = SOME (KBL T)) ∧ (getKBT x = SOME (KBT T))
    then
      exec [SOME (CMD RB)]
    else exec [NONE]) ∧
   (uavM0out LeRe (exec v0) = exec [NONE]) ∧
   (uavM0out s (trap v1) = exec [NONE]) ∧
   (uavM0out s (discard v2) = exec [NONE])
```

[uavM0out_ind]

```
⊢ ∀P.
    (∀x. P Off (exec x)) ∧ (∀x. P LlRe (exec x)) ∧
    (∀x. P LeRl (exec x)) ∧ (∀x. P LlRl (exec x)) ∧
    (∀v0. P LeRe (exec v0)) ∧ (∀s v1. P s (trap v1)) ∧
    (∀s v2. P s (discard v2)) ⇒
    ∀v v1. P v v1
```

B.6 uavSSM0 Theory

Built: 11 February 2018
Parent Theories: uavDef, principal

B.6.1 Definitions

[C2_LeRl_RR_Auth_def]

```
⊢ C2_LeRl_RR_Auth =
  prop (SOME (MA R)) impf prop (SOME (KBL T)) impf
  prop (SOME (KBT T)) impf
  Name C2 controls prop (SOME (CMD RR))
```

[C2_LlRe_RL_Auth_def]

```
⊢ C2_LlRe_RL_Auth =
  prop (SOME (MA L)) impf prop (SOME (KBL T)) impf
  prop (SOME (KBT T)) impf
  Name C2 controls prop (SOME (CMD RL))
```

[C2_LlRl_RB_Auth_def]

```
⊢ C2_LlRl_RB_Auth =
  prop (SOME (MA B)) impf prop (SOME (KBL T)) impf
  prop (SOME (KBT T)) impf
  Name C2 controls prop (SOME (CMD RB))
```

[C2_LlRl_RL_Auth_def]

```
⊢ C2_LlRl_RL_Auth =
  prop (SOME (MA B)) impf prop (SOME (KBL T)) impf
  prop (SOME (KBT T)) impf
  Name C2 controls prop (SOME (CMD RL))
```

[C2_LlRl_RR_Auth_def]

```
⊢ C2_LlRl_RR_Auth =
  prop (SOME (MA B)) impf prop (SOME (KBL T)) impf
  prop (SOME (KBT T)) impf
  Name C2 controls prop (SOME (CMD RR))
```

[cmdAuthorizeContext_def]

```
⊢ ∀s x.
    cmdAuthorizeContext s x =
    if
      (getMAStatement x = NONE) ∨ (getKBLStatement x = NONE) ∨
      (getKBTStatement x = NONE)
    then
      [prop NONE]
    else if getC2Statement x = NONE then propCommandList x
    else if (s = Off) ∨ (s = LeRe) then [prop NONE]
    else if s = LlRl then
      if
        getMAStatement x ≠ SOME (MA B) ∨
        (getKBLStatement x = SOME (KBL F)) ∨
        (getKBTStatement x = SOME (KBT F))
      then
        [prop NONE]
      else [C2_LlRl_RB_Auth; C2_LlRl_RL_Auth; C2_LlRl_RR_Auth]
    else if s = LlRe then
      if
        getMAStatement x ≠ SOME (MA L) ∨
        (getKBLStatement x = SOME (KBL F)) ∨
        (getKBTStatement x = SOME (KBT F)) ∨
        getC2Statement x ≠ SOME (CMD RL)
      then
        [prop NONE]
      else [C2_LlRe_RL_Auth]
    else if
      getMAStatement x ≠ SOME (MA R) ∨
      (getKBLStatement x = SOME (KBL F)) ∨
      (getKBTStatement x = SOME (KBT F)) ∨
      getC2Statement x ≠ SOME (CMD RR)
    then
      [prop NONE]
    else [C2_LeRl_RR_Auth]
```

[sensorContext_def]

```
⊢ ∀x.
    sensorContext x =
    [maSensorContext x; gpskbSensorContext x;
     tkbSensorContext x]
```

B.6.2 Theorems

[C2_LlRe_exec_NOP_justified_lemma]

```
⊢ ∀NS Out M Oi Os.
    TR (M, Oi, Os)
      (exec
         (inputList
            [Name MunitionAvail says prop (SOME (MA L));
             Name GPSKB says prop (SOME (KBL T));
             Name TimeKB says prop (SOME (KBT T))]))
      (CFG inputOK cmdAuthorizeContext sensorContext
         ([Name MunitionAvail says prop (SOME (MA L));
           Name GPSKB says prop (SOME (KBL T));
           Name TimeKB says prop (SOME (KBT T))]::ins) LlRe
         outs)
```

```
    (CFG inputOK cmdAuthorizeContext sensorContext ins
       (NS LlRe
          (exec
             (inputList
                [Name MunitionAvail says prop (SOME (MA L));
                 Name GPSKB says prop (SOME (KBL T));
                 Name TimeKB says prop (SOME (KBT T))])))
       (Out LlRe
          (exec
             (inputList
                [Name MunitionAvail says prop (SOME (MA L));
                 Name GPSKB says prop (SOME (KBL T));
                 Name TimeKB says prop (SOME (KBT T))])):::
         outs)) ⟺
  authenticationTest inputOK
    [Name MunitionAvail says prop (SOME (MA L));
     Name GPSKB says prop (SOME (KBL T));
     Name TimeKB says prop (SOME (KBT T))] ∧
  CFGInterpret (M,Oi,Os)
    (CFG inputOK cmdAuthorizeContext sensorContext
       ([Name MunitionAvail says prop (SOME (MA L));
         Name GPSKB says prop (SOME (KBL T));
         Name TimeKB says prop (SOME (KBT T))]::ins) LlRe
       outs) ∧
  (M,Oi,Os) satList
  propCommandList
    [Name MunitionAvail says prop (SOME (MA L));
     Name GPSKB says prop (SOME (KBL T));
     Name TimeKB says prop (SOME (KBT T))]
```

[C2_LlRe_exec_NOP_lemma]

```
⊢ ∀M Oi Os.
  CFGInterpret (M,Oi,Os)
    (CFG inputOK cmdAuthorizeContext sensorContext
       ([Name MunitionAvail says prop (SOME (MA L));
         Name GPSKB says prop (SOME (KBL T));
         Name TimeKB says prop (SOME (KBT T))]::ins) LlRe
       outs) ⇒
  (M,Oi,Os) satList
  propCommandList
    [Name MunitionAvail says prop (SOME (MA L));
     Name GPSKB says prop (SOME (KBL T));
     Name TimeKB says prop (SOME (KBT T))]
```

[C2_LlRe_exec_NOP_thm]

```
⊢ ∀NS Out M Oi Os.
  TR (M,Oi,Os)
    (exec [SOME (MA L); SOME (KBL T); SOME (KBT T)])
    (CFG inputOK cmdAuthorizeContext sensorContext
       ([Name MunitionAvail says prop (SOME (MA L));
         Name GPSKB says prop (SOME (KBL T));
         Name TimeKB says prop (SOME (KBT T))]::ins) LlRe
       outs)
    (CFG inputOK cmdAuthorizeContext sensorContext ins
       (NS LlRe
          (exec [SOME (MA L); SOME (KBL T); SOME (KBT T)]))
       (Out LlRe
          (exec [SOME (MA L); SOME (KBL T); SOME (KBT T)])::
```

```
           outs)) ⟺
    authenticationTest inputOK
      [Name MunitionAvail says prop (SOME (MA L));
       Name GPSKB says prop (SOME (KBL T));
       Name TimeKB says prop (SOME (KBT T))] ∧
    CFGInterpret (M,Oi,Os)
      (CFG inputOK cmdAuthorizeContext sensorContext
         ([Name MunitionAvail says prop (SOME (MA L));
           Name GPSKB says prop (SOME (KBL T));
           Name TimeKB says prop (SOME (KBT T))]::ins) LlRe
         outs) ∧
    (M,Oi,Os) satList
    [prop (SOME (MA L)); prop (SOME (KBL T));
     prop (SOME (KBT T))]
```

[C2_LlRe_exec_RL_justified_lemma]

```
⊢ ∀NS Out M Oi Os.
    TR (M,Oi,Os)
      (exec
         (inputList
            [Name C2 says prop (SOME (CMD RL));
             Name MunitionAvail says prop (SOME (MA L));
             Name GPSKB says prop (SOME (KBL T));
             Name TimeKB says prop (SOME (KBT T))]))
      (CFG inputOK cmdAuthorizeContext sensorContext
         ([Name C2 says prop (SOME (CMD RL));
           Name MunitionAvail says prop (SOME (MA L));
           Name GPSKB says prop (SOME (KBL T));
           Name TimeKB says prop (SOME (KBT T))]::ins) LlRe
         outs)
      (CFG inputOK cmdAuthorizeContext sensorContext ins
         (NS LlRe
            (exec
               (inputList
                  [Name C2 says prop (SOME (CMD RL));
                   Name MunitionAvail says prop (SOME (MA L));
                   Name GPSKB says prop (SOME (KBL T));
                   Name TimeKB says prop (SOME (KBT T))])))
         (Out LlRe
            (exec
               (inputList
                  [Name C2 says prop (SOME (CMD RL));
                   Name MunitionAvail says prop (SOME (MA L));
                   Name GPSKB says prop (SOME (KBL T));
                   Name TimeKB says prop (SOME (KBT T))]))::
           outs)) ⟺
    authenticationTest inputOK
      [Name C2 says prop (SOME (CMD RL));
       Name MunitionAvail says prop (SOME (MA L));
       Name GPSKB says prop (SOME (KBL T));
       Name TimeKB says prop (SOME (KBT T))] ∧
    CFGInterpret (M,Oi,Os)
      (CFG inputOK cmdAuthorizeContext sensorContext
         ([Name C2 says prop (SOME (CMD RL));
           Name MunitionAvail says prop (SOME (MA L));
           Name GPSKB says prop (SOME (KBL T));
           Name TimeKB says prop (SOME (KBT T))]::ins) LlRe
         outs) ∧
```

```
     (M,Oi,Os) satList
     propCommandList
       [Name C2 says prop (SOME (CMD RL));
        Name MunitionAvail says prop (SOME (MA L));
        Name GPSKB says prop (SOME (KBL T));
        Name TimeKB says prop (SOME (KBT T))]
```

[C2_LlRe_exec_RL_lemma]

```
 ⊢ ∀M Oi Os.
     CFGInterpret (M,Oi,Os)
       (CFG inputOK cmdAuthorizeContext sensorContext
          ([Name C2 says prop (SOME (CMD RL));
            Name MunitionAvail says prop (SOME (MA L));
            Name GPSKB says prop (SOME (KBL T));
            Name TimeKB says prop (SOME (KBT T))]::ins) LlRe
          outs) ⇒
     (M,Oi,Os) satList
     propCommandList
       [Name C2 says prop (SOME (CMD RL));
        Name MunitionAvail says prop (SOME (MA L));
        Name GPSKB says prop (SOME (KBL T));
        Name TimeKB says prop (SOME (KBT T))]
```

[C2_LlRe_exec_RL_thm]

```
 ⊢ ∀NS Out M Oi Os.
     TR (M,Oi,Os)
       (exec
          [SOME (CMD RL); SOME (MA L); SOME (KBL T);
           SOME (KBT T)])
       (CFG inputOK cmdAuthorizeContext sensorContext
          ([Name C2 says prop (SOME (CMD RL));
            Name MunitionAvail says prop (SOME (MA L));
            Name GPSKB says prop (SOME (KBL T));
            Name TimeKB says prop (SOME (KBT T))]::ins) LlRe
          outs)
       (CFG inputOK cmdAuthorizeContext sensorContext ins
          (NS LlRe
             (exec
                [SOME (CMD RL); SOME (MA L); SOME (KBL T);
                 SOME (KBT T)]))
          (Out LlRe
             (exec
                [SOME (CMD RL); SOME (MA L); SOME (KBL T);
                 SOME (KBT T)])::outs)) ⟺
     authenticationTest inputOK
       [Name C2 says prop (SOME (CMD RL));
        Name MunitionAvail says prop (SOME (MA L));
        Name GPSKB says prop (SOME (KBL T));
        Name TimeKB says prop (SOME (KBT T))] ∧
     CFGInterpret (M,Oi,Os)
       (CFG inputOK cmdAuthorizeContext sensorContext
          ([Name C2 says prop (SOME (CMD RL));
            Name MunitionAvail says prop (SOME (MA L));
            Name GPSKB says prop (SOME (KBL T));
            Name TimeKB says prop (SOME (KBT T))]::ins) LlRe
          outs) ∧
     (M,Oi,Os) satList
     [prop (SOME (CMD RL)); prop (SOME (MA L));
      prop (SOME (KBL T)); prop (SOME (KBT T))]
```

[C2_LlRe_trap_RL_KBL_F_justified_lemma]

```
⊢ ∀ NS Out M Oi Os.
    TR (M,Oi,Os)
      (trap
         (inputList
            [Name C2 says prop (SOME (CMD RL));
             Name MunitionAvail says prop (SOME (MA L));
             Name GPSKB says prop (SOME (KBL F));
             Name TimeKB says prop (SOME (KBT T))]))
      (CFG inputOK cmdAuthorizeContext sensorContext
         ([Name C2 says prop (SOME (CMD RL));
           Name MunitionAvail says prop (SOME (MA L));
           Name GPSKB says prop (SOME (KBL F));
           Name TimeKB says prop (SOME (KBT T))]::ins) LlRe
         outs)
      (CFG inputOK cmdAuthorizeContext sensorContext ins
         (NS LlRe
            (trap
               (inputList
                  [Name C2 says prop (SOME (CMD RL));
                   Name MunitionAvail says prop (SOME (MA L));
                   Name GPSKB says prop (SOME (KBL F));
                   Name TimeKB says prop (SOME (KBT T))])))
         (Out LlRe
            (trap
               (inputList
                  [Name C2 says prop (SOME (CMD RL));
                   Name MunitionAvail says prop (SOME (MA L));
                   Name GPSKB says prop (SOME (KBL F));
                   Name TimeKB says prop (SOME (KBT T))])))::
           outs)) ⟺
    authenticationTest inputOK
      [Name C2 says prop (SOME (CMD RL));
       Name MunitionAvail says prop (SOME (MA L));
       Name GPSKB says prop (SOME (KBL F));
       Name TimeKB says prop (SOME (KBT T))] ∧
    CFGInterpret (M,Oi,Os)
      (CFG inputOK cmdAuthorizeContext sensorContext
         ([Name C2 says prop (SOME (CMD RL));
           Name MunitionAvail says prop (SOME (MA L));
           Name GPSKB says prop (SOME (KBL F));
           Name TimeKB says prop (SOME (KBT T))]::ins) LlRe
         outs) ∧ (M,Oi,Os) sat prop NONE
```

[C2_LlRe_trap_RL_KBL_F_justified_thm]

```
⊢ ∀ NS Out M Oi Os.
    TR (M,Oi,Os)
      (trap
         [SOME (CMD RL); SOME (MA L); SOME (KBL F);
          SOME (KBT T)])
      (CFG inputOK cmdAuthorizeContext sensorContext
         ([Name C2 says prop (SOME (CMD RL));
           Name MunitionAvail says prop (SOME (MA L));
           Name GPSKB says prop (SOME (KBL F));
           Name TimeKB says prop (SOME (KBT T))]::ins) LlRe
         outs)
      (CFG inputOK cmdAuthorizeContext sensorContext ins
         (NS LlRe
```

```
            (trap
               [SOME (CMD RL); SOME (MA L); SOME (KBL F);
                SOME (KBT T)]))
         (Out LlRe
            (trap
               [SOME (CMD RL); SOME (MA L); SOME (KBL F);
                SOME (KBT T)])::outs)) ⟺
   authenticationTest inputOK
     [Name C2 says prop (SOME (CMD RL));
      Name MunitionAvail says prop (SOME (MA L));
      Name GPSKB says prop (SOME (KBL F));
      Name TimeKB says prop (SOME (KBT T))] ∧
   CFGInterpret (M,Oi,Os)
     (CFG inputOK cmdAuthorizeContext sensorContext
        ([Name C2 says prop (SOME (CMD RL));
          Name MunitionAvail says prop (SOME (MA L));
          Name GPSKB says prop (SOME (KBL F));
          Name TimeKB says prop (SOME (KBT T))]::ins) LlRe
        outs) ∧ (M,Oi,Os) sat prop NONE
```

[C2_LlRe_trap_RL_KBL_F_lemma]

```
⊢ ∀M Oi Os.
   CFGInterpret (M,Oi,Os)
     (CFG inputOK cmdAuthorizeContext sensorContext
        ([Name C2 says prop (SOME (CMD RL));
          Name MunitionAvail says prop (SOME (MA L));
          Name GPSKB says prop (SOME (KBL F));
          Name TimeKB says prop (SOME (KBT T))]::ins) LlRe
        outs) ⇒
   (M,Oi,Os) sat prop NONE
```

[discard_MASensor_CMD_inject_thm]

```
⊢ ∀NS Out M Oi Os.
   TR (M,Oi,Os)
     (discard
        (inputList
           [Name MunitionAvail says prop (SOME (CMD RL));
            Name GPSKB says prop (SOME (KBL F));
            Name TimeKB says prop (SOME (KBT T))]))
     (CFG inputOK cmdAuthorizeContext sensorContext
        ([Name MunitionAvail says prop (SOME (CMD RL));
          Name GPSKB says prop (SOME (KBL F));
          Name TimeKB says prop (SOME (KBT T))]::ins) LlRe
        outs)
     (CFG inputOK cmdAuthorizeContext sensorContext ins
        (NS LlRe
           (discard
              (inputList
                 [Name MunitionAvail says
                  prop (SOME (CMD RL));
                  Name GPSKB says prop (SOME (KBL F));
                  Name TimeKB says prop (SOME (KBT T))])))
        (Out LlRe
           (discard
              (inputList
                 [Name MunitionAvail says
                  prop (SOME (CMD RL));
                  Name GPSKB says prop (SOME (KBL F));
```

```
                Name TimeKB says prop (SOME (KBT T))])):: outs))
```

[getC2Statement_def]

```
⊢ (getC2Statement [] = NONE) ∧
  (∀xs cmd.
     getC2Statement (Name C2 says prop (SOME (CMD cmd))::xs) =
     SOME (CMD cmd)) ∧
  (∀xs. getC2Statement (TT::xs) = getC2Statement xs) ∧
  (∀xs. getC2Statement (FF::xs) = getC2Statement xs) ∧
  (∀xs v2. getC2Statement (prop v2::xs) = getC2Statement xs) ∧
  (∀xs v3. getC2Statement (notf v3::xs) = getC2Statement xs) ∧
  (∀xs v5 v4.
     getC2Statement (v4 andf v5::xs) = getC2Statement xs) ∧
  (∀xs v7 v6.
     getC2Statement (v6 orf v7::xs) = getC2Statement xs) ∧
  (∀xs v9 v8.
     getC2Statement (v8 impf v9::xs) = getC2Statement xs) ∧
  (∀xs v11 v10.
     getC2Statement (v10 eqf v11::xs) = getC2Statement xs) ∧
  (∀xs v12.
     getC2Statement (v12 says TT::xs) = getC2Statement xs) ∧
  (∀xs v12.
     getC2Statement (v12 says FF::xs) = getC2Statement xs) ∧
  (∀xs v134.
     getC2Statement (Name v134 says prop NONE::xs) =
     getC2Statement xs) ∧
  (∀xs v146 v144.
     getC2Statement
       (Name (Staff v146) says prop (SOME v144)::xs) =
     getC2Statement xs) ∧
  (∀xs v147 v144.
     getC2Statement
       (Name (Authority v147) says prop (SOME v144)::xs) =
     getC2Statement xs) ∧
  (∀xs v148 v144.
     getC2Statement
       (Name (Role v148) says prop (SOME v144)::xs) =
     getC2Statement xs) ∧
  (∀xs v149 v144.
     getC2Statement
       (Name (KeyS v149) says prop (SOME v144)::xs) =
     getC2Statement xs) ∧
  (∀xs v150 v144.
     getC2Statement
       (Name (KeyA v150) says prop (SOME v144)::xs) =
     getC2Statement xs) ∧
  (∀xs v157.
     getC2Statement (Name C2 says prop (SOME (MA v157))::xs) =
     getC2Statement xs) ∧
  (∀xs v158.
     getC2Statement (Name C2 says prop (SOME (KBL v158))::xs) =
     getC2Statement xs) ∧
  (∀xs v159.
     getC2Statement (Name C2 says prop (SOME (KBT v159))::xs) =
     getC2Statement xs) ∧
  (∀xs v144.
     getC2Statement
```

```
    (Name MunitionAvail says prop (SOME v144)::xs) =
   getC2Statement xs) ∧
(∀ xs v144 .
   getC2Statement (Name GPSKB says prop (SOME v144)::xs) =
   getC2Statement xs) ∧
(∀ xs v144 .
   getC2Statement (Name TimeKB says prop (SOME v144)::xs) =
   getC2Statement xs) ∧
(∀ xs v68 v136 v135 .
   getC2Statement (v135 meet v136 says prop v68::xs) =
   getC2Statement xs) ∧
(∀ xs v68 v138 v137 .
   getC2Statement (v137 quoting v138 says prop v68::xs) =
   getC2Statement xs) ∧
(∀ xs v69 v12 .
   getC2Statement (v12 says notf v69::xs) =
   getC2Statement xs) ∧
(∀ xs v71 v70 v12 .
   getC2Statement (v12 says (v70 andf v71)::xs) =
   getC2Statement xs) ∧
(∀ xs v73 v72 v12 .
   getC2Statement (v12 says (v72 orf v73)::xs) =
   getC2Statement xs) ∧
(∀ xs v75 v74 v12 .
   getC2Statement (v12 says (v74 impf v75)::xs) =
   getC2Statement xs) ∧
(∀ xs v77 v76 v12 .
   getC2Statement (v12 says (v76 eqf v77)::xs) =
   getC2Statement xs) ∧
(∀ xs v79 v78 v12 .
   getC2Statement (v12 says v78 says v79::xs) =
   getC2Statement xs) ∧
(∀ xs v81 v80 v12 .
   getC2Statement (v12 says v80 speaks_for v81::xs) =
   getC2Statement xs) ∧
(∀ xs v83 v82 v12 .
   getC2Statement (v12 says v82 controls v83::xs) =
   getC2Statement xs) ∧
(∀ xs v86 v85 v84 v12 .
   getC2Statement (v12 says reps v84 v85 v86::xs) =
   getC2Statement xs) ∧
(∀ xs v88 v87 v12 .
   getC2Statement (v12 says v87 domi v88::xs) =
   getC2Statement xs) ∧
(∀ xs v90 v89 v12 .
   getC2Statement (v12 says v89 eqi v90::xs) =
   getC2Statement xs) ∧
(∀ xs v92 v91 v12 .
   getC2Statement (v12 says v91 doms v92::xs) =
   getC2Statement xs) ∧
(∀ xs v94 v93 v12 .
   getC2Statement (v12 says v93 eqs v94::xs) =
   getC2Statement xs) ∧
(∀ xs v96 v95 v12 .
   getC2Statement (v12 says v95 eqn v96::xs) =
   getC2Statement xs) ∧
(∀ xs v98 v97 v12 .
   getC2Statement (v12 says v97 lte v98::xs) =
```

```
    getC2Statement xs) ∧
  (∀ xs v99 v12 v100.
    getC2Statement (v12 says v99 lt v100::xs) =
    getC2Statement xs) ∧
  (∀ xs v15 v14.
    getC2Statement (v14 speaks_for v15::xs) =
    getC2Statement xs) ∧
  (∀ xs v17 v16.
    getC2Statement (v16 controls v17::xs) =
    getC2Statement xs) ∧
  (∀ xs v20 v19 v18.
    getC2Statement (reps v18 v19 v20::xs) =
    getC2Statement xs) ∧
  (∀ xs v22 v21.
    getC2Statement (v21 domi v22::xs) = getC2Statement xs) ∧
  (∀ xs v24 v23.
    getC2Statement (v23 eqi v24::xs) = getC2Statement xs) ∧
  (∀ xs v26 v25.
    getC2Statement (v25 doms v26::xs) = getC2Statement xs) ∧
  (∀ xs v28 v27.
    getC2Statement (v27 eqs v28::xs) = getC2Statement xs) ∧
  (∀ xs v30 v29.
    getC2Statement (v29 eqn v30::xs) = getC2Statement xs) ∧
  (∀ xs v32 v31.
    getC2Statement (v31 lte v32::xs) = getC2Statement xs) ∧
  ∀ xs v34 v33.
    getC2Statement (v33 lt v34::xs) = getC2Statement xs
```

[getC2Statement_ind]

```
⊢ ∀ P.
    P [] ∧
    (∀ cmd xs. P (Name C2 says prop (SOME (CMD cmd))::xs)) ∧
    (∀ xs. P xs ⇒ P (TT::xs)) ∧ (∀ xs. P xs ⇒ P (FF::xs)) ∧
    (∀ v2 xs. P xs ⇒ P (prop v2::xs)) ∧
    (∀ v3 xs. P xs ⇒ P (notf v3::xs)) ∧
    (∀ v4 v5 xs. P xs ⇒ P (v4 andf v5::xs)) ∧
    (∀ v6 v7 xs. P xs ⇒ P (v6 orf v7::xs)) ∧
    (∀ v8 v9 xs. P xs ⇒ P (v8 impf v9::xs)) ∧
    (∀ v10 v11 xs. P xs ⇒ P (v10 eqf v11::xs)) ∧
    (∀ v12 xs. P xs ⇒ P (v12 says TT::xs)) ∧
    (∀ v12 xs. P xs ⇒ P (v12 says FF::xs)) ∧
    (∀ v134 xs. P xs ⇒ P (Name v134 says prop NONE::xs)) ∧
    (∀ v146 v144 xs.
       P xs ⇒
       P (Name (Staff v146) says prop (SOME v144)::xs)) ∧
    (∀ v147 v144 xs.
       P xs ⇒
       P (Name (Authority v147) says prop (SOME v144)::xs)) ∧
    (∀ v148 v144 xs.
       P xs ⇒ P (Name (Role v148) says prop (SOME v144)::xs)) ∧
    (∀ v149 v144 xs.
       P xs ⇒ P (Name (KeyS v149) says prop (SOME v144)::xs)) ∧
    (∀ v150 v144 xs.
       P xs ⇒ P (Name (KeyA v150) says prop (SOME v144)::xs)) ∧
    (∀ v157 xs.
       P xs ⇒ P (Name C2 says prop (SOME (MA v157))::xs)) ∧
    (∀ v158 xs.
       P xs ⇒ P (Name C2 says prop (SOME (KBL v158))::xs)) ∧
```

```
   (∀ v159 xs.
      P xs ⇒ P (Name C2 says prop (SOME (KBT v159))::xs)) ∧
   (∀ v144 xs.
      P xs ⇒
      P (Name MunitionAvail says prop (SOME v144)::xs)) ∧
   (∀ v144 xs.
      P xs ⇒ P (Name GPSKB says prop (SOME v144)::xs)) ∧
   (∀ v144 xs.
      P xs ⇒ P (Name TimeKB says prop (SOME v144)::xs)) ∧
   (∀ v135 v136 v68 xs.
      P xs ⇒ P (v135 meet v136 says prop v68::xs)) ∧
   (∀ v137 v138 v68 xs.
      P xs ⇒ P (v137 quoting v138 says prop v68::xs)) ∧
   (∀ v12 v69 xs. P xs ⇒ P (v12 says notf v69::xs)) ∧
   (∀ v12 v70 v71 xs. P xs ⇒ P (v12 says (v70 andf v71)::xs)) ∧
   (∀ v12 v72 v73 xs. P xs ⇒ P (v12 says (v72 orf v73)::xs)) ∧
   (∀ v12 v74 v75 xs. P xs ⇒ P (v12 says (v74 impf v75)::xs)) ∧
   (∀ v12 v76 v77 xs. P xs ⇒ P (v12 says (v76 eqf v77)::xs)) ∧
   (∀ v12 v78 v79 xs. P xs ⇒ P (v12 says v78 says v79::xs)) ∧
   (∀ v12 v80 v81 xs.
      P xs ⇒ P (v12 says v80 speaks_for v81::xs)) ∧
   (∀ v12 v82 v83 xs.
      P xs ⇒ P (v12 says v82 controls v83::xs)) ∧
   (∀ v12 v84 v85 v86 xs.
      P xs ⇒ P (v12 says reps v84 v85 v86::xs)) ∧
   (∀ v12 v87 v88 xs. P xs ⇒ P (v12 says v87 domi v88::xs)) ∧
   (∀ v12 v89 v90 xs. P xs ⇒ P (v12 says v89 eqi v90::xs)) ∧
   (∀ v12 v91 v92 xs. P xs ⇒ P (v12 says v91 doms v92::xs)) ∧
   (∀ v12 v93 v94 xs. P xs ⇒ P (v12 says v93 eqs v94::xs)) ∧
   (∀ v12 v95 v96 xs. P xs ⇒ P (v12 says v95 eqn v96::xs)) ∧
   (∀ v12 v97 v98 xs. P xs ⇒ P (v12 says v97 lte v98::xs)) ∧
   (∀ v12 v99 v100 xs. P xs ⇒ P (v12 says v99 lt v100::xs)) ∧
   (∀ v14 v15 xs. P xs ⇒ P (v14 speaks_for v15::xs)) ∧
   (∀ v16 v17 xs. P xs ⇒ P (v16 controls v17::xs)) ∧
   (∀ v18 v19 v20 xs. P xs ⇒ P (reps v18 v19 v20::xs)) ∧
   (∀ v21 v22 xs. P xs ⇒ P (v21 domi v22::xs)) ∧
   (∀ v23 v24 xs. P xs ⇒ P (v23 eqi v24::xs)) ∧
   (∀ v25 v26 xs. P xs ⇒ P (v25 doms v26::xs)) ∧
   (∀ v27 v28 xs. P xs ⇒ P (v27 eqs v28::xs)) ∧
   (∀ v29 v30 xs. P xs ⇒ P (v29 eqn v30::xs)) ∧
   (∀ v31 v32 xs. P xs ⇒ P (v31 lte v32::xs)) ∧
   (∀ v33 v34 xs. P xs ⇒ P (v33 lt v34::xs)) ⇒
   ∀ v. P v
```

[getKBLStatement_def]

```
⊢ (getKBLStatement [] = NONE) ∧
  (∀ xs loc.
     getKBLStatement
       (Name GPSKB says prop (SOME (KBL loc))::xs) =
     SOME (KBL loc)) ∧
  (∀ xs. getKBLStatement (TT::xs) = getKBLStatement xs) ∧
  (∀ xs. getKBLStatement (FF::xs) = getKBLStatement xs) ∧
  (∀ xs v2.
     getKBLStatement (prop v2::xs) = getKBLStatement xs) ∧
  (∀ xs v3.
     getKBLStatement (notf v3::xs) = getKBLStatement xs) ∧
  (∀ xs v5 v4.
     getKBLStatement (v4 andf v5::xs) = getKBLStatement xs) ∧
```

```
(∀ xs v7 v6.
   getKBLStatement (v6 orf v7::xs) = getKBLStatement xs) ∧
(∀ xs v9 v8.
   getKBLStatement (v8 impf v9::xs) = getKBLStatement xs) ∧
(∀ xs v11 v10.
   getKBLStatement (v10 eqf v11::xs) = getKBLStatement xs) ∧
(∀ xs v12.
   getKBLStatement (v12 says TT::xs) = getKBLStatement xs) ∧
(∀ xs v12.
   getKBLStatement (v12 says FF::xs) = getKBLStatement xs) ∧
(∀ xs v134.
   getKBLStatement (Name v134 says prop NONE::xs) =
   getKBLStatement xs) ∧
(∀ xs v146 v144.
   getKBLStatement
     (Name (Staff v146) says prop (SOME v144)::xs) =
   getKBLStatement xs) ∧
(∀ xs v147 v144.
   getKBLStatement
     (Name (Authority v147) says prop (SOME v144)::xs) =
   getKBLStatement xs) ∧
(∀ xs v148 v144.
   getKBLStatement
     (Name (Role v148) says prop (SOME v144)::xs) =
   getKBLStatement xs) ∧
(∀ xs v149 v144.
   getKBLStatement
     (Name (KeyS v149) says prop (SOME v144)::xs) =
   getKBLStatement xs) ∧
(∀ xs v150 v144.
   getKBLStatement
     (Name (KeyA v150) says prop (SOME v144)::xs) =
   getKBLStatement xs) ∧
(∀ xs v144.
   getKBLStatement (Name C2 says prop (SOME v144)::xs) =
   getKBLStatement xs) ∧
(∀ xs v144.
   getKBLStatement
     (Name MunitionAvail says prop (SOME v144)::xs) =
   getKBLStatement xs) ∧
(∀ xs v156.
   getKBLStatement
     (Name GPSKB says prop (SOME (CMD v156))::xs) =
   getKBLStatement xs) ∧
(∀ xs v157.
   getKBLStatement
     (Name GPSKB says prop (SOME (MA v157))::xs) =
   getKBLStatement xs) ∧
(∀ xs v159.
   getKBLStatement
     (Name GPSKB says prop (SOME (KBT v159))::xs) =
   getKBLStatement xs) ∧
(∀ xs v144.
   getKBLStatement (Name TimeKB says prop (SOME v144)::xs) =
   getKBLStatement xs) ∧
(∀ xs v68 v136 v135.
   getKBLStatement (v135 meet v136 says prop v68::xs) =
   getKBLStatement xs) ∧
```

```
(∀ xs v68 v138 v137.
   getKBLStatement (v137 quoting v138 says prop v68::xs) =
   getKBLStatement xs) ∧
(∀ xs v69 v12.
   getKBLStatement (v12 says notf v69::xs) =
   getKBLStatement xs) ∧
(∀ xs v71 v70 v12.
   getKBLStatement (v12 says (v70 andf v71)::xs) =
   getKBLStatement xs) ∧
(∀ xs v73 v72 v12.
   getKBLStatement (v12 says (v72 orf v73)::xs) =
   getKBLStatement xs) ∧
(∀ xs v75 v74 v12.
   getKBLStatement (v12 says (v74 impf v75)::xs) =
   getKBLStatement xs) ∧
(∀ xs v77 v76 v12.
   getKBLStatement (v12 says (v76 eqf v77)::xs) =
   getKBLStatement xs) ∧
(∀ xs v79 v78 v12.
   getKBLStatement (v12 says v78 says v79::xs) =
   getKBLStatement xs) ∧
(∀ xs v81 v80 v12.
   getKBLStatement (v12 says v80 speaks_for v81::xs) =
   getKBLStatement xs) ∧
(∀ xs v83 v82 v12.
   getKBLStatement (v12 says v82 controls v83::xs) =
   getKBLStatement xs) ∧
(∀ xs v86 v85 v84 v12.
   getKBLStatement (v12 says reps v84 v85 v86::xs) =
   getKBLStatement xs) ∧
(∀ xs v88 v87 v12.
   getKBLStatement (v12 says v87 domi v88::xs) =
   getKBLStatement xs) ∧
(∀ xs v90 v89 v12.
   getKBLStatement (v12 says v89 eqi v90::xs) =
   getKBLStatement xs) ∧
(∀ xs v92 v91 v12.
   getKBLStatement (v12 says v91 doms v92::xs) =
   getKBLStatement xs) ∧
(∀ xs v94 v93 v12.
   getKBLStatement (v12 says v93 eqs v94::xs) =
   getKBLStatement xs) ∧
(∀ xs v96 v95 v12.
   getKBLStatement (v12 says v95 eqn v96::xs) =
   getKBLStatement xs) ∧
(∀ xs v98 v97 v12.
   getKBLStatement (v12 says v97 lte v98::xs) =
   getKBLStatement xs) ∧
(∀ xs v99 v12 v100.
   getKBLStatement (v12 says v99 lt v100::xs) =
   getKBLStatement xs) ∧
(∀ xs v15 v14.
   getKBLStatement (v14 speaks_for v15::xs) =
   getKBLStatement xs) ∧
(∀ xs v17 v16.
   getKBLStatement (v16 controls v17::xs) =
   getKBLStatement xs) ∧
(∀ xs v20 v19 v18.
```

```
    getKBLStatement (reps v18 v19 v20::xs) =
    getKBLStatement xs) ∧
  (∀ xs v22 v21.
    getKBLStatement (v21 domi v22::xs) = getKBLStatement xs) ∧
  (∀ xs v24 v23.
    getKBLStatement (v23 eqi v24::xs) = getKBLStatement xs) ∧
  (∀ xs v26 v25.
    getKBLStatement (v25 doms v26::xs) = getKBLStatement xs) ∧
  (∀ xs v28 v27.
    getKBLStatement (v27 eqs v28::xs) = getKBLStatement xs) ∧
  (∀ xs v30 v29.
    getKBLStatement (v29 eqn v30::xs) = getKBLStatement xs) ∧
  (∀ xs v32 v31.
    getKBLStatement (v31 lte v32::xs) = getKBLStatement xs) ∧
  ∀ xs v34 v33.
    getKBLStatement (v33 lt v34::xs) = getKBLStatement xs
```

[getKBLStatement_ind]

```
⊢ ∀ P.
    P [] ∧
    (∀ loc xs. P (Name GPSKB says prop (SOME (KBL loc))::xs)) ∧
    (∀ xs. P xs ⇒ P (TT::xs)) ∧ (∀ xs. P xs ⇒ P (FF::xs)) ∧
    (∀ v2 xs. P xs ⇒ P (prop v2::xs)) ∧
    (∀ v3 xs. P xs ⇒ P (notf v3::xs)) ∧
    (∀ v4 v5 xs. P xs ⇒ P (v4 andf v5::xs)) ∧
    (∀ v6 v7 xs. P xs ⇒ P (v6 orf v7::xs)) ∧
    (∀ v8 v9 xs. P xs ⇒ P (v8 impf v9::xs)) ∧
    (∀ v10 v11 xs. P xs ⇒ P (v10 eqf v11::xs)) ∧
    (∀ v12 xs. P xs ⇒ P (v12 says TT::xs)) ∧
    (∀ v12 xs. P xs ⇒ P (v12 says FF::xs)) ∧
    (∀ v134 xs. P xs ⇒ P (Name v134 says prop NONE::xs)) ∧
    (∀ v146 v144 xs.
       P xs ⇒
       P (Name (Staff v146) says prop (SOME v144)::xs)) ∧
    (∀ v147 v144 xs.
       P xs ⇒
       P (Name (Authority v147) says prop (SOME v144)::xs)) ∧
    (∀ v148 v144 xs.
       P xs ⇒ P (Name (Role v148) says prop (SOME v144)::xs)) ∧
    (∀ v149 v144 xs.
       P xs ⇒ P (Name (KeyS v149) says prop (SOME v144)::xs)) ∧
    (∀ v150 v144 xs.
       P xs ⇒ P (Name (KeyA v150) says prop (SOME v144)::xs)) ∧
    (∀ v144 xs. P xs ⇒ P (Name C2 says prop (SOME v144)::xs)) ∧
    (∀ v144 xs.
       P xs ⇒
       P (Name MunitionAvail says prop (SOME v144)::xs)) ∧
    (∀ v156 xs.
       P xs ⇒ P (Name GPSKB says prop (SOME (CMD v156))::xs)) ∧
    (∀ v157 xs.
       P xs ⇒ P (Name GPSKB says prop (SOME (MA v157))::xs)) ∧
    (∀ v159 xs.
       P xs ⇒ P (Name GPSKB says prop (SOME (KBT v159))::xs)) ∧
    (∀ v144 xs.
       P xs ⇒ P (Name TimeKB says prop (SOME v144)::xs)) ∧
    (∀ v135 v136 v68 xs.
       P xs ⇒ P (v135 meet v136 says prop v68::xs)) ∧
    (∀ v137 v138 v68 xs.
```

```
      P xs ⇒ P (v137 quoting v138 says prop v68::xs)) ∧
    (∀ v12 v69 xs. P xs ⇒ P (v12 says notf v69::xs)) ∧
    (∀ v12 v70 v71 xs. P xs ⇒ P (v12 says (v70 andf v71)::xs)) ∧
    (∀ v12 v72 v73 xs. P xs ⇒ P (v12 says (v72 orf v73)::xs)) ∧
    (∀ v12 v74 v75 xs. P xs ⇒ P (v12 says (v74 impf v75)::xs)) ∧
    (∀ v12 v76 v77 xs. P xs ⇒ P (v12 says (v76 eqf v77)::xs)) ∧
    (∀ v12 v78 v79 xs. P xs ⇒ P (v12 says v78 says v79::xs)) ∧
    (∀ v12 v80 v81 xs.
      P xs ⇒ P (v12 says v80 speaks_for v81::xs)) ∧
    (∀ v12 v82 v83 xs.
      P xs ⇒ P (v12 says v82 controls v83::xs)) ∧
    (∀ v12 v84 v85 v86 xs.
      P xs ⇒ P (v12 says reps v84 v85 v86::xs)) ∧
    (∀ v12 v87 v88 xs. P xs ⇒ P (v12 says v87 domi v88::xs)) ∧
    (∀ v12 v89 v90 xs. P xs ⇒ P (v12 says v89 eqi v90::xs)) ∧
    (∀ v12 v91 v92 xs. P xs ⇒ P (v12 says v91 doms v92::xs)) ∧
    (∀ v12 v93 v94 xs. P xs ⇒ P (v12 says v93 eqs v94::xs)) ∧
    (∀ v12 v95 v96 xs. P xs ⇒ P (v12 says v95 eqn v96::xs)) ∧
    (∀ v12 v97 v98 xs. P xs ⇒ P (v12 says v97 lte v98::xs)) ∧
    (∀ v12 v99 v100 xs. P xs ⇒ P (v12 says v99 lt v100::xs)) ∧
    (∀ v14 v15 xs. P xs ⇒ P (v14 speaks_for v15::xs)) ∧
    (∀ v16 v17 xs. P xs ⇒ P (v16 controls v17::xs)) ∧
    (∀ v18 v19 v20 xs. P xs ⇒ P (reps v18 v19 v20::xs)) ∧
    (∀ v21 v22 xs. P xs ⇒ P (v21 domi v22::xs)) ∧
    (∀ v23 v24 xs. P xs ⇒ P (v23 eqi v24::xs)) ∧
    (∀ v25 v26 xs. P xs ⇒ P (v25 doms v26::xs)) ∧
    (∀ v27 v28 xs. P xs ⇒ P (v27 eqs v28::xs)) ∧
    (∀ v29 v30 xs. P xs ⇒ P (v29 eqn v30::xs)) ∧
    (∀ v31 v32 xs. P xs ⇒ P (v31 lte v32::xs)) ∧
    (∀ v33 v34 xs. P xs ⇒ P (v33 lt v34::xs)) ⇒
    ∀ v. P v
```

[getKBTStatement_def]

```
⊢ (getKBTStatement [] = NONE) ∧
  (∀ xs loc.
     getKBTStatement
       (Name TimeKB says prop (SOME (KBT loc))::xs) =
     SOME (KBT loc)) ∧
  (∀ xs. getKBTStatement (TT::xs) = getKBTStatement xs) ∧
  (∀ xs. getKBTStatement (FF::xs) = getKBTStatement xs) ∧
  (∀ xs v2.
     getKBTStatement (prop v2::xs) = getKBTStatement xs) ∧
  (∀ xs v3.
     getKBTStatement (notf v3::xs) = getKBTStatement xs) ∧
  (∀ xs v5 v4.
     getKBTStatement (v4 andf v5::xs) = getKBTStatement xs) ∧
  (∀ xs v7 v6.
     getKBTStatement (v6 orf v7::xs) = getKBTStatement xs) ∧
  (∀ xs v9 v8.
     getKBTStatement (v8 impf v9::xs) = getKBTStatement xs) ∧
  (∀ xs v11 v10.
     getKBTStatement (v10 eqf v11::xs) = getKBTStatement xs) ∧
  (∀ xs v12.
     getKBTStatement (v12 says TT::xs) = getKBTStatement xs) ∧
  (∀ xs v12.
     getKBTStatement (v12 says FF::xs) = getKBTStatement xs) ∧
  (∀ xs v134.
     getKBTStatement (Name v134 says prop NONE::xs) =
```

```
   getKBTStatement xs) ∧
(∀xs v146 v144.
   getKBTStatement
     (Name (Staff v146) says prop (SOME v144)::xs) =
   getKBTStatement xs) ∧
(∀xs v147 v144.
   getKBTStatement
     (Name (Authority v147) says prop (SOME v144)::xs) =
   getKBTStatement xs) ∧
(∀xs v148 v144.
   getKBTStatement
     (Name (Role v148) says prop (SOME v144)::xs) =
   getKBTStatement xs) ∧
(∀xs v149 v144.
   getKBTStatement
     (Name (KeyS v149) says prop (SOME v144)::xs) =
   getKBTStatement xs) ∧
(∀xs v150 v144.
   getKBTStatement
     (Name (KeyA v150) says prop (SOME v144)::xs) =
   getKBTStatement xs) ∧
(∀xs v144.
   getKBTStatement (Name C2 says prop (SOME v144)::xs) =
   getKBTStatement xs) ∧
(∀xs v144.
   getKBTStatement
     (Name MunitionAvail says prop (SOME v144)::xs) =
   getKBTStatement xs) ∧
(∀xs v144.
   getKBTStatement (Name GPSKB says prop (SOME v144)::xs) =
   getKBTStatement xs) ∧
(∀xs v156.
   getKBTStatement
     (Name TimeKB says prop (SOME (CMD v156))::xs) =
   getKBTStatement xs) ∧
(∀xs v157.
   getKBTStatement
     (Name TimeKB says prop (SOME (MA v157))::xs) =
   getKBTStatement xs) ∧
(∀xs v158.
   getKBTStatement
     (Name TimeKB says prop (SOME (KBL v158))::xs) =
   getKBTStatement xs) ∧
(∀xs v68 v136 v135.
   getKBTStatement (v135 meet v136 says prop v68::xs) =
   getKBTStatement xs) ∧
(∀xs v68 v138 v137.
   getKBTStatement (v137 quoting v138 says prop v68::xs) =
   getKBTStatement xs) ∧
(∀xs v69 v12.
   getKBTStatement (v12 says notf v69::xs) =
   getKBTStatement xs) ∧
(∀xs v71 v70 v12.
   getKBTStatement (v12 says (v70 andf v71)::xs) =
   getKBTStatement xs) ∧
(∀xs v73 v72 v12.
   getKBTStatement (v12 says (v72 orf v73)::xs) =
   getKBTStatement xs) ∧
```

```
(∀ xs v75 v74 v12.
   getKBTStatement (v12 says (v74 impf v75)::xs) =
   getKBTStatement xs) ∧
(∀ xs v77 v76 v12.
   getKBTStatement (v12 says (v76 eqf v77)::xs) =
   getKBTStatement xs) ∧
(∀ xs v79 v78 v12.
   getKBTStatement (v12 says v78 says v79::xs) =
   getKBTStatement xs) ∧
(∀ xs v81 v80 v12.
   getKBTStatement (v12 says v80 speaks_for v81::xs) =
   getKBTStatement xs) ∧
(∀ xs v83 v82 v12.
   getKBTStatement (v12 says v82 controls v83::xs) =
   getKBTStatement xs) ∧
(∀ xs v86 v85 v84 v12.
   getKBTStatement (v12 says reps v84 v85 v86::xs) =
   getKBTStatement xs) ∧
(∀ xs v88 v87 v12.
   getKBTStatement (v12 says v87 domi v88::xs) =
   getKBTStatement xs) ∧
(∀ xs v90 v89 v12.
   getKBTStatement (v12 says v89 eqi v90::xs) =
   getKBTStatement xs) ∧
(∀ xs v92 v91 v12.
   getKBTStatement (v12 says v91 doms v92::xs) =
   getKBTStatement xs) ∧
(∀ xs v94 v93 v12.
   getKBTStatement (v12 says v93 eqs v94::xs) =
   getKBTStatement xs) ∧
(∀ xs v96 v95 v12.
   getKBTStatement (v12 says v95 eqn v96::xs) =
   getKBTStatement xs) ∧
(∀ xs v98 v97 v12.
   getKBTStatement (v12 says v97 lte v98::xs) =
   getKBTStatement xs) ∧
(∀ xs v99 v12 v100.
   getKBTStatement (v12 says v99 lt v100::xs) =
   getKBTStatement xs) ∧
(∀ xs v15 v14.
   getKBTStatement (v14 speaks_for v15::xs) =
   getKBTStatement xs) ∧
(∀ xs v17 v16.
   getKBTStatement (v16 controls v17::xs) =
   getKBTStatement xs) ∧
(∀ xs v20 v19 v18.
   getKBTStatement (reps v18 v19 v20::xs) =
   getKBTStatement xs) ∧
(∀ xs v22 v21.
   getKBTStatement (v21 domi v22::xs) = getKBTStatement xs) ∧
(∀ xs v24 v23.
   getKBTStatement (v23 eqi v24::xs) = getKBTStatement xs) ∧
(∀ xs v26 v25.
   getKBTStatement (v25 doms v26::xs) = getKBTStatement xs) ∧
(∀ xs v28 v27.
   getKBTStatement (v27 eqs v28::xs) = getKBTStatement xs) ∧
(∀ xs v30 v29.
   getKBTStatement (v29 eqn v30::xs) = getKBTStatement xs) ∧
```

(∀ xs v_{32} v_{31}.
  getKBTStatement (v_{31} lte v_{32}::xs) = getKBTStatement xs) ∧
∀ xs v_{34} v_{33}.
  getKBTStatement (v_{33} lt v_{34}::xs) = getKBTStatement xs

[getKBTStatement_ind]

⊢ ∀ P.
  P [] ∧
  (∀ loc xs. P (Name TimeKB says prop (SOME (KBT loc))::xs)) ∧
  (∀ xs. P xs ⇒ P (TT::xs)) ∧ (∀ xs. P xs ⇒ P (FF::xs)) ∧
  (∀ v_2 xs. P xs ⇒ P (prop v_2::xs)) ∧
  (∀ v_3 xs. P xs ⇒ P (notf v_3::xs)) ∧
  (∀ v_4 v_5 xs. P xs ⇒ P (v_4 andf v_5::xs)) ∧
  (∀ v_6 v_7 xs. P xs ⇒ P (v_6 orf v_7::xs)) ∧
  (∀ v_8 v_9 xs. P xs ⇒ P (v_8 impf v_9::xs)) ∧
  (∀ v_{10} v_{11} xs. P xs ⇒ P (v_{10} eqf v_{11}::xs)) ∧
  (∀ v_{12} xs. P xs ⇒ P (v_{12} says TT::xs)) ∧
  (∀ v_{12} xs. P xs ⇒ P (v_{12} says FF::xs)) ∧
  (∀ $v134$ xs. P xs ⇒ P (Name $v134$ says prop NONE::xs)) ∧
  (∀ $v146$ $v144$ xs.
    P xs ⇒
    P (Name (Staff $v146$) says prop (SOME $v144$)::xs)) ∧
  (∀ $v147$ $v144$ xs.
    P xs ⇒
    P (Name (Authority $v147$) says prop (SOME $v144$)::xs)) ∧
  (∀ $v148$ $v144$ xs.
    P xs ⇒ P (Name (Role $v148$) says prop (SOME $v144$)::xs)) ∧
  (∀ $v149$ $v144$ xs.
    P xs ⇒ P (Name (KeyS $v149$) says prop (SOME $v144$)::xs)) ∧
  (∀ $v150$ $v144$ xs.
    P xs ⇒ P (Name (KeyA $v150$) says prop (SOME $v144$)::xs)) ∧
  (∀ $v144$ xs. P xs ⇒ P (Name C2 says prop (SOME $v144$)::xs)) ∧
  (∀ $v144$ xs.
    P xs ⇒
    P (Name MunitionAvail says prop (SOME $v144$)::xs)) ∧
  (∀ $v144$ xs.
    P xs ⇒ P (Name GPSKB says prop (SOME $v144$)::xs)) ∧
  (∀ $v156$ xs.
    P xs ⇒
    P (Name TimeKB says prop (SOME (CMD $v156$))::xs)) ∧
  (∀ $v157$ xs.
    P xs ⇒ P (Name TimeKB says prop (SOME (MA $v157$))::xs)) ∧
  (∀ $v158$ xs.
    P xs ⇒
    P (Name TimeKB says prop (SOME (KBL $v158$))::xs)) ∧
  (∀ $v135$ $v136$ v_{68} xs.
    P xs ⇒ P ($v135$ meet $v136$ says prop v_{68}::xs)) ∧
  (∀ $v137$ $v138$ v_{68} xs.
    P xs ⇒ P ($v137$ quoting $v138$ says prop v_{68}::xs)) ∧
  (∀ v_{12} v_{69} xs. P xs ⇒ P (v_{12} says notf v_{69}::xs)) ∧
  (∀ v_{12} v_{70} v_{71} xs. P xs ⇒ P (v_{12} says (v_{70} andf v_{71})::xs)) ∧
  (∀ v_{12} v_{72} v_{73} xs. P xs ⇒ P (v_{12} says (v_{72} orf v_{73})::xs)) ∧
  (∀ v_{12} v_{74} v_{75} xs. P xs ⇒ P (v_{12} says (v_{74} impf v_{75})::xs)) ∧
  (∀ v_{12} v_{76} v_{77} xs. P xs ⇒ P (v_{12} says (v_{76} eqf v_{77})::xs)) ∧
  (∀ v_{12} v_{78} v_{79} xs. P xs ⇒ P (v_{12} says v_{78} says v_{79}::xs)) ∧
  (∀ v_{12} v_{80} v_{81} xs.
    P xs ⇒ P (v_{12} says v_{80} speaks_for v_{81}::xs)) ∧
  (∀ v_{12} v_{82} v_{83} xs.

```
      P xs ⇒ P (v12 says v82 controls v83::xs)) ∧
    (∀ v12 v84 v85 v86 xs.
      P xs ⇒ P (v12 says reps v84 v85 v86::xs)) ∧
    (∀ v12 v87 v88 xs. P xs ⇒ P (v12 says v87 domi v88::xs)) ∧
    (∀ v12 v89 v90 xs. P xs ⇒ P (v12 says v89 eqi v90::xs)) ∧
    (∀ v12 v91 v92 xs. P xs ⇒ P (v12 says v91 doms v92::xs)) ∧
    (∀ v12 v93 v94 xs. P xs ⇒ P (v12 says v93 eqs v94::xs)) ∧
    (∀ v12 v95 v96 xs. P xs ⇒ P (v12 says v95 eqn v96::xs)) ∧
    (∀ v12 v97 v98 xs. P xs ⇒ P (v12 says v97 lte v98::xs)) ∧
    (∀ v12 v99 v100 xs. P xs ⇒ P (v12 says v99 lt v100::xs)) ∧
    (∀ v14 v15 xs. P xs ⇒ P (v14 speaks_for v15::xs)) ∧
    (∀ v16 v17 xs. P xs ⇒ P (v16 controls v17::xs)) ∧
    (∀ v18 v19 v20 xs. P xs ⇒ P (reps v18 v19 v20::xs)) ∧
    (∀ v21 v22 xs. P xs ⇒ P (v21 domi v22::xs)) ∧
    (∀ v23 v24 xs. P xs ⇒ P (v23 eqi v24::xs)) ∧
    (∀ v25 v26 xs. P xs ⇒ P (v25 doms v26::xs)) ∧
    (∀ v27 v28 xs. P xs ⇒ P (v27 eqs v28::xs)) ∧
    (∀ v29 v30 xs. P xs ⇒ P (v29 eqn v30::xs)) ∧
    (∀ v31 v32 xs. P xs ⇒ P (v31 lte v32::xs)) ∧
    (∀ v33 v34 xs. P xs ⇒ P (v33 lt v34::xs)) ⇒
    ∀ v. P v
```

[getMAStatement_def]

```
⊢ (getMAStatement [] = NONE) ∧
  (∀ xs ma.
     getMAStatement
       (Name MunitionAvail says prop (SOME (MA ma))::xs) =
     SOME (MA ma)) ∧
  (∀ xs. getMAStatement (TT::xs) = getMAStatement xs) ∧
  (∀ xs. getMAStatement (FF::xs) = getMAStatement xs) ∧
  (∀ xs v2. getMAStatement (prop v2::xs) = getMAStatement xs) ∧
  (∀ xs v3. getMAStatement (notf v3::xs) = getMAStatement xs) ∧
  (∀ xs v5 v4.
     getMAStatement (v4 andf v5::xs) = getMAStatement xs) ∧
  (∀ xs v7 v6.
     getMAStatement (v6 orf v7::xs) = getMAStatement xs) ∧
  (∀ xs v9 v8.
     getMAStatement (v8 impf v9::xs) = getMAStatement xs) ∧
  (∀ xs v11 v10.
     getMAStatement (v10 eqf v11::xs) = getMAStatement xs) ∧
  (∀ xs v12.
     getMAStatement (v12 says TT::xs) = getMAStatement xs) ∧
  (∀ xs v12.
     getMAStatement (v12 says FF::xs) = getMAStatement xs) ∧
  (∀ xs v134.
     getMAStatement (Name v134 says prop NONE::xs) =
     getMAStatement xs) ∧
  (∀ xs v146 v144.
     getMAStatement
       (Name (Staff v146) says prop (SOME v144)::xs) =
     getMAStatement xs) ∧
  (∀ xs v147 v144.
     getMAStatement
       (Name (Authority v147) says prop (SOME v144)::xs) =
     getMAStatement xs) ∧
  (∀ xs v148 v144.
     getMAStatement
       (Name (Role v148) says prop (SOME v144)::xs) =
```

```
   getMAStatement xs) ∧
(∀ xs v149 v144 .
   getMAStatement
     (Name (KeyS v149) says prop (SOME v144)::xs) =
   getMAStatement xs) ∧
(∀ xs v150 v144 .
   getMAStatement
     (Name (KeyA v150) says prop (SOME v144)::xs) =
   getMAStatement xs) ∧
(∀ xs v144 .
   getMAStatement (Name C2 says prop (SOME v144)::xs) =
   getMAStatement xs) ∧
(∀ xs v156 .
   getMAStatement
     (Name MunitionAvail says prop (SOME (CMD v156))::xs) =
   getMAStatement xs) ∧
(∀ xs v158 .
   getMAStatement
     (Name MunitionAvail says prop (SOME (KBL v158))::xs) =
   getMAStatement xs) ∧
(∀ xs v159 .
   getMAStatement
     (Name MunitionAvail says prop (SOME (KBT v159))::xs) =
   getMAStatement xs) ∧
(∀ xs v144 .
   getMAStatement (Name GPSKB says prop (SOME v144)::xs) =
   getMAStatement xs) ∧
(∀ xs v144 .
   getMAStatement (Name TimeKB says prop (SOME v144)::xs) =
   getMAStatement xs) ∧
(∀ xs v68 v136 v135 .
   getMAStatement (v135 meet v136 says prop v68::xs) =
   getMAStatement xs) ∧
(∀ xs v68 v138 v137 .
   getMAStatement (v137 quoting v138 says prop v68::xs) =
   getMAStatement xs) ∧
(∀ xs v69 v12 .
   getMAStatement (v12 says notf v69::xs) =
   getMAStatement xs) ∧
(∀ xs v71 v70 v12 .
   getMAStatement (v12 says (v70 andf v71)::xs) =
   getMAStatement xs) ∧
(∀ xs v73 v72 v12 .
   getMAStatement (v12 says (v72 orf v73)::xs) =
   getMAStatement xs) ∧
(∀ xs v75 v74 v12 .
   getMAStatement (v12 says (v74 impf v75)::xs) =
   getMAStatement xs) ∧
(∀ xs v77 v76 v12 .
   getMAStatement (v12 says (v76 eqf v77)::xs) =
   getMAStatement xs) ∧
(∀ xs v79 v78 v12 .
   getMAStatement (v12 says v78 says v79::xs) =
   getMAStatement xs) ∧
(∀ xs v81 v80 v12 .
   getMAStatement (v12 says v80 speaks_for v81::xs) =
   getMAStatement xs) ∧
(∀ xs v83 v82 v12 .
```

```
    getMAStatement (v12 says v82 controls v83::xs) =
    getMAStatement xs) ∧
  (∀ xs v86 v85 v84 v12.
    getMAStatement (v12 says reps v84 v85 v86::xs) =
    getMAStatement xs) ∧
  (∀ xs v88 v87 v12.
    getMAStatement (v12 says v87 domi v88::xs) =
    getMAStatement xs) ∧
  (∀ xs v90 v89 v12.
    getMAStatement (v12 says v89 eqi v90::xs) =
    getMAStatement xs) ∧
  (∀ xs v92 v91 v12.
    getMAStatement (v12 says v91 doms v92::xs) =
    getMAStatement xs) ∧
  (∀ xs v94 v93 v12.
    getMAStatement (v12 says v93 eqs v94::xs) =
    getMAStatement xs) ∧
  (∀ xs v96 v95 v12.
    getMAStatement (v12 says v95 eqn v96::xs) =
    getMAStatement xs) ∧
  (∀ xs v98 v97 v12.
    getMAStatement (v12 says v97 lte v98::xs) =
    getMAStatement xs) ∧
  (∀ xs v99 v12 v100.
    getMAStatement (v12 says v99 lt v100::xs) =
    getMAStatement xs) ∧
  (∀ xs v15 v14.
    getMAStatement (v14 speaks_for v15::xs) =
    getMAStatement xs) ∧
  (∀ xs v17 v16.
    getMAStatement (v16 controls v17::xs) =
    getMAStatement xs) ∧
  (∀ xs v20 v19 v18.
    getMAStatement (reps v18 v19 v20::xs) =
    getMAStatement xs) ∧
  (∀ xs v22 v21.
    getMAStatement (v21 domi v22::xs) = getMAStatement xs) ∧
  (∀ xs v24 v23.
    getMAStatement (v23 eqi v24::xs) = getMAStatement xs) ∧
  (∀ xs v26 v25.
    getMAStatement (v25 doms v26::xs) = getMAStatement xs) ∧
  (∀ xs v28 v27.
    getMAStatement (v27 eqs v28::xs) = getMAStatement xs) ∧
  (∀ xs v30 v29.
    getMAStatement (v29 eqn v30::xs) = getMAStatement xs) ∧
  (∀ xs v32 v31.
    getMAStatement (v31 lte v32::xs) = getMAStatement xs) ∧
  ∀ xs v34 v33.
    getMAStatement (v33 lt v34::xs) = getMAStatement xs
```

[getMAStatement_ind]

```
⊢ ∀ P.
    P [] ∧
    (∀ ma xs.
       P (Name MunitionAvail says prop (SOME (MA ma))::xs)) ∧
    (∀ xs. P xs ⇒ P (TT::xs)) ∧ (∀ xs. P xs ⇒ P (FF::xs)) ∧
    (∀ v2 xs. P xs ⇒ P (prop v2::xs)) ∧
    (∀ v3 xs. P xs ⇒ P (notf v3::xs)) ∧
```

```
(∀ v4 v5 xs. P xs ⇒ P (v4 andf v5::xs)) ∧
(∀ v6 v7 xs. P xs ⇒ P (v6 orf v7::xs)) ∧
(∀ v8 v9 xs. P xs ⇒ P (v8 impf v9::xs)) ∧
(∀ v10 v11 xs. P xs ⇒ P (v10 eqf v11::xs)) ∧
(∀ v12 xs. P xs ⇒ P (v12 says TT::xs)) ∧
(∀ v12 xs. P xs ⇒ P (v12 says FF::xs)) ∧
(∀ v134 xs. P xs ⇒ P (Name v134 says prop NONE::xs)) ∧
(∀ v146 v144 xs.
   P xs ⇒
   P (Name (Staff v146) says prop (SOME v144)::xs)) ∧
(∀ v147 v144 xs.
   P xs ⇒
   P (Name (Authority v147) says prop (SOME v144)::xs)) ∧
(∀ v148 v144 xs.
   P xs ⇒ P (Name (Role v148) says prop (SOME v144)::xs)) ∧
(∀ v149 v144 xs.
   P xs ⇒ P (Name (KeyS v149) says prop (SOME v144)::xs)) ∧
(∀ v150 v144 xs.
   P xs ⇒ P (Name (KeyA v150) says prop (SOME v144)::xs)) ∧
(∀ v144 xs. P xs ⇒ P (Name C2 says prop (SOME v144)::xs)) ∧
(∀ v156 xs.
   P xs ⇒
   P
     (Name MunitionAvail says prop (SOME (CMD v156))::
          xs)) ∧
(∀ v158 xs.
   P xs ⇒
   P
     (Name MunitionAvail says prop (SOME (KBL v158))::
          xs)) ∧
(∀ v159 xs.
   P xs ⇒
   P
     (Name MunitionAvail says prop (SOME (KBT v159))::
          xs)) ∧
(∀ v144 xs.
   P xs ⇒ P (Name GPSKB says prop (SOME v144)::xs)) ∧
(∀ v144 xs.
   P xs ⇒ P (Name TimeKB says prop (SOME v144)::xs)) ∧
(∀ v135 v136 v68 xs.
   P xs ⇒ P (v135 meet v136 says prop v68::xs)) ∧
(∀ v137 v138 v68 xs.
   P xs ⇒ P (v137 quoting v138 says prop v68::xs)) ∧
(∀ v12 v69 xs. P xs ⇒ P (v12 says notf v69::xs)) ∧
(∀ v12 v70 v71 xs. P xs ⇒ P (v12 says (v70 andf v71)::xs)) ∧
(∀ v12 v72 v73 xs. P xs ⇒ P (v12 says (v72 orf v73)::xs)) ∧
(∀ v12 v74 v75 xs. P xs ⇒ P (v12 says (v74 impf v75)::xs)) ∧
(∀ v12 v76 v77 xs. P xs ⇒ P (v12 says (v76 eqf v77)::xs)) ∧
(∀ v12 v78 v79 xs. P xs ⇒ P (v12 says v78 says v79::xs)) ∧
(∀ v12 v80 v81 xs.
   P xs ⇒ P (v12 says v80 speaks_for v81::xs)) ∧
(∀ v12 v82 v83 xs.
   P xs ⇒ P (v12 says v82 controls v83::xs)) ∧
(∀ v12 v84 v85 v86 xs.
   P xs ⇒ P (v12 says reps v84 v85 v86::xs)) ∧
(∀ v12 v87 v88 xs. P xs ⇒ P (v12 says v87 domi v88::xs)) ∧
(∀ v12 v89 v90 xs. P xs ⇒ P (v12 says v89 eqi v90::xs)) ∧
(∀ v12 v91 v92 xs. P xs ⇒ P (v12 says v91 doms v92::xs)) ∧
```

```
   (∀ v12 v93 v94 xs. P xs ⇒ P (v12 says v93 eqs v94::xs)) ∧
   (∀ v12 v95 v96 xs. P xs ⇒ P (v12 says v95 eqn v96::xs)) ∧
   (∀ v12 v97 v98 xs. P xs ⇒ P (v12 says v97 lte v98::xs)) ∧
   (∀ v12 v99 v100 xs. P xs ⇒ P (v12 says v99 lt v100::xs)) ∧
   (∀ v14 v15 xs. P xs ⇒ P (v14 speaks_for v15::xs)) ∧
   (∀ v16 v17 xs. P xs ⇒ P (v16 controls v17::xs)) ∧
   (∀ v18 v19 v20 xs. P xs ⇒ P (reps v18 v19 v20::xs)) ∧
   (∀ v21 v22 xs. P xs ⇒ P (v21 domi v22::xs)) ∧
   (∀ v23 v24 xs. P xs ⇒ P (v23 eqi v24::xs)) ∧
   (∀ v25 v26 xs. P xs ⇒ P (v25 doms v26::xs)) ∧
   (∀ v27 v28 xs. P xs ⇒ P (v27 eqs v28::xs)) ∧
   (∀ v29 v30 xs. P xs ⇒ P (v29 eqn v30::xs)) ∧
   (∀ v31 v32 xs. P xs ⇒ P (v31 lte v32::xs)) ∧
   (∀ v33 v34 xs. P xs ⇒ P (v33 lt v34::xs)) ⇒
   ∀ v. P v
```

[gpskbSensorContext_def]

```
⊢ (gpskbSensorContext [] = TT) ∧
  (∀ xs loc.
     gpskbSensorContext
       (Name GPSKB says prop (SOME (KBL loc))::xs) =
     Name GPSKB controls prop (SOME (KBL loc))) ∧
  (∀ xs. gpskbSensorContext (TT::xs) = gpskbSensorContext xs) ∧
  (∀ xs. gpskbSensorContext (FF::xs) = gpskbSensorContext xs) ∧
  (∀ xs v2.
     gpskbSensorContext (prop v2::xs) =
     gpskbSensorContext xs) ∧
  (∀ xs v3.
     gpskbSensorContext (notf v3::xs) =
     gpskbSensorContext xs) ∧
  (∀ xs v5 v4.
     gpskbSensorContext (v4 andf v5::xs) =
     gpskbSensorContext xs) ∧
  (∀ xs v7 v6.
     gpskbSensorContext (v6 orf v7::xs) =
     gpskbSensorContext xs) ∧
  (∀ xs v9 v8.
     gpskbSensorContext (v8 impf v9::xs) =
     gpskbSensorContext xs) ∧
  (∀ xs v11 v10.
     gpskbSensorContext (v10 eqf v11::xs) =
     gpskbSensorContext xs) ∧
  (∀ xs v12.
     gpskbSensorContext (v12 says TT::xs) =
     gpskbSensorContext xs) ∧
  (∀ xs v12.
     gpskbSensorContext (v12 says FF::xs) =
     gpskbSensorContext xs) ∧
  (∀ xs v134.
     gpskbSensorContext (Name v134 says prop NONE::xs) =
     gpskbSensorContext xs) ∧
  (∀ xs v146 v144.
     gpskbSensorContext
       (Name (Staff v146) says prop (SOME v144)::xs) =
     gpskbSensorContext xs) ∧
  (∀ xs v147 v144.
     gpskbSensorContext
       (Name (Authority v147) says prop (SOME v144)::xs) =
```

```
   gpskbSensorContext xs) ∧
(∀ xs v148 v144 .
   gpskbSensorContext
     (Name (Role v148) says prop (SOME v144)::xs) =
   gpskbSensorContext xs) ∧
(∀ xs v149 v144 .
   gpskbSensorContext
     (Name (KeyS v149) says prop (SOME v144)::xs) =
   gpskbSensorContext xs) ∧
(∀ xs v150 v144 .
   gpskbSensorContext
     (Name (KeyA v150) says prop (SOME v144)::xs) =
   gpskbSensorContext xs) ∧
(∀ xs v144 .
   gpskbSensorContext (Name C2 says prop (SOME v144)::xs) =
   gpskbSensorContext xs) ∧
(∀ xs v144 .
   gpskbSensorContext
     (Name MunitionAvail says prop (SOME v144)::xs) =
   gpskbSensorContext xs) ∧
(∀ xs v156 .
   gpskbSensorContext
     (Name GPSKB says prop (SOME (CMD v156))::xs) =
   gpskbSensorContext xs) ∧
(∀ xs v157 .
   gpskbSensorContext
     (Name GPSKB says prop (SOME (MA v157))::xs) =
   gpskbSensorContext xs) ∧
(∀ xs v159 .
   gpskbSensorContext
     (Name GPSKB says prop (SOME (KBT v159))::xs) =
   gpskbSensorContext xs) ∧
(∀ xs v144 .
   gpskbSensorContext
     (Name TimeKB says prop (SOME v144)::xs) =
   gpskbSensorContext xs) ∧
(∀ xs v68 v136 v135 .
   gpskbSensorContext (v135 meet v136 says prop v68::xs) =
   gpskbSensorContext xs) ∧
(∀ xs v68 v138 v137 .
   gpskbSensorContext (v137 quoting v138 says prop v68::xs) =
   gpskbSensorContext xs) ∧
(∀ xs v69 v12 .
   gpskbSensorContext (v12 says notf v69::xs) =
   gpskbSensorContext xs) ∧
(∀ xs v71 v70 v12 .
   gpskbSensorContext (v12 says (v70 andf v71)::xs) =
   gpskbSensorContext xs) ∧
(∀ xs v73 v72 v12 .
   gpskbSensorContext (v12 says (v72 orf v73)::xs) =
   gpskbSensorContext xs) ∧
(∀ xs v75 v74 v12 .
   gpskbSensorContext (v12 says (v74 impf v75)::xs) =
   gpskbSensorContext xs) ∧
(∀ xs v77 v76 v12 .
   gpskbSensorContext (v12 says (v76 eqf v77)::xs) =
   gpskbSensorContext xs) ∧
(∀ xs v79 v78 v12 .
```

```
   gpskbSensorContext (v12 says v78 says v79::xs) =
   gpskbSensorContext xs) ∧
(∀ xs v81 v80 v12.
   gpskbSensorContext (v12 says v80 speaks_for v81::xs) =
   gpskbSensorContext xs) ∧
(∀ xs v83 v82 v12.
   gpskbSensorContext (v12 says v82 controls v83::xs) =
   gpskbSensorContext xs) ∧
(∀ xs v86 v85 v84 v12.
   gpskbSensorContext (v12 says reps v84 v85 v86::xs) =
   gpskbSensorContext xs) ∧
(∀ xs v88 v87 v12.
   gpskbSensorContext (v12 says v87 domi v88::xs) =
   gpskbSensorContext xs) ∧
(∀ xs v90 v89 v12.
   gpskbSensorContext (v12 says v89 eqi v90::xs) =
   gpskbSensorContext xs) ∧
(∀ xs v92 v91 v12.
   gpskbSensorContext (v12 says v91 doms v92::xs) =
   gpskbSensorContext xs) ∧
(∀ xs v94 v93 v12.
   gpskbSensorContext (v12 says v93 eqs v94::xs) =
   gpskbSensorContext xs) ∧
(∀ xs v96 v95 v12.
   gpskbSensorContext (v12 says v95 eqn v96::xs) =
   gpskbSensorContext xs) ∧
(∀ xs v98 v97 v12.
   gpskbSensorContext (v12 says v97 lte v98::xs) =
   gpskbSensorContext xs) ∧
(∀ xs v99 v12 v100.
   gpskbSensorContext (v12 says v99 lt v100::xs) =
   gpskbSensorContext xs) ∧
(∀ xs v15 v14.
   gpskbSensorContext (v14 speaks_for v15::xs) =
   gpskbSensorContext xs) ∧
(∀ xs v17 v16.
   gpskbSensorContext (v16 controls v17::xs) =
   gpskbSensorContext xs) ∧
(∀ xs v20 v19 v18.
   gpskbSensorContext (reps v18 v19 v20::xs) =
   gpskbSensorContext xs) ∧
(∀ xs v22 v21.
   gpskbSensorContext (v21 domi v22::xs) =
   gpskbSensorContext xs) ∧
(∀ xs v24 v23.
   gpskbSensorContext (v23 eqi v24::xs) =
   gpskbSensorContext xs) ∧
(∀ xs v26 v25.
   gpskbSensorContext (v25 doms v26::xs) =
   gpskbSensorContext xs) ∧
(∀ xs v28 v27.
   gpskbSensorContext (v27 eqs v28::xs) =
   gpskbSensorContext xs) ∧
(∀ xs v30 v29.
   gpskbSensorContext (v29 eqn v30::xs) =
   gpskbSensorContext xs) ∧
(∀ xs v32 v31.
   gpskbSensorContext (v31 lte v32::xs) =
```

```
    gpskbSensorContext xs) ∧
  ∀xs v34 v33.
    gpskbSensorContext (v33 lt v34::xs) = gpskbSensorContext xs
```

[gpskbSensorContext_ind]

```
⊢ ∀P.
    P [] ∧
    (∀loc xs. P (Name GPSKB says prop (SOME (KBL loc))::xs)) ∧
    (∀xs. P xs ⇒ P (TT::xs)) ∧ (∀xs. P xs ⇒ P (FF::xs)) ∧
    (∀v2 xs. P xs ⇒ P (prop v2::xs)) ∧
    (∀v3 xs. P xs ⇒ P (notf v3::xs)) ∧
    (∀v4 v5 xs. P xs ⇒ P (v4 andf v5::xs)) ∧
    (∀v6 v7 xs. P xs ⇒ P (v6 orf v7::xs)) ∧
    (∀v8 v9 xs. P xs ⇒ P (v8 impf v9::xs)) ∧
    (∀v10 v11 xs. P xs ⇒ P (v10 eqf v11::xs)) ∧
    (∀v12 xs. P xs ⇒ P (v12 says TT::xs)) ∧
    (∀v12 xs. P xs ⇒ P (v12 says FF::xs)) ∧
    (∀v134 xs. P xs ⇒ P (Name v134 says prop NONE::xs)) ∧
    (∀v146 v144 xs.
       P xs ⇒
       P (Name (Staff v146) says prop (SOME v144)::xs)) ∧
    (∀v147 v144 xs.
       P xs ⇒
       P (Name (Authority v147) says prop (SOME v144)::xs)) ∧
    (∀v148 v144 xs.
       P xs ⇒ P (Name (Role v148) says prop (SOME v144)::xs)) ∧
    (∀v149 v144 xs.
       P xs ⇒ P (Name (KeyS v149) says prop (SOME v144)::xs)) ∧
    (∀v150 v144 xs.
       P xs ⇒ P (Name (KeyA v150) says prop (SOME v144)::xs)) ∧
    (∀v144 xs. P xs ⇒ P (Name C2 says prop (SOME v144)::xs)) ∧
    (∀v144 xs.
       P xs ⇒
       P (Name MunitionAvail says prop (SOME v144)::xs)) ∧
    (∀v156 xs.
       P xs ⇒ P (Name GPSKB says prop (SOME (CMD v156))::xs)) ∧
    (∀v157 xs.
       P xs ⇒ P (Name GPSKB says prop (SOME (MA v157))::xs)) ∧
    (∀v159 xs.
       P xs ⇒ P (Name GPSKB says prop (SOME (KBT v159))::xs)) ∧
    (∀v144 xs.
       P xs ⇒ P (Name TimeKB says prop (SOME v144)::xs)) ∧
    (∀v135 v136 v68 xs.
       P xs ⇒ P (v135 meet v136 says prop v68::xs)) ∧
    (∀v137 v138 v68 xs.
       P xs ⇒ P (v137 quoting v138 says prop v68::xs)) ∧
    (∀v12 v69 xs. P xs ⇒ P (v12 says notf v69::xs)) ∧
    (∀v12 v70 v71 xs. P xs ⇒ P (v12 says (v70 andf v71)::xs)) ∧
    (∀v12 v72 v73 xs. P xs ⇒ P (v12 says (v72 orf v73)::xs)) ∧
    (∀v12 v74 v75 xs. P xs ⇒ P (v12 says (v74 impf v75)::xs)) ∧
    (∀v12 v76 v77 xs. P xs ⇒ P (v12 says (v76 eqf v77)::xs)) ∧
    (∀v12 v78 v79 xs. P xs ⇒ P (v12 says v78 says v79::xs)) ∧
    (∀v12 v80 v81 xs.
       P xs ⇒ P (v12 says v80 speaks_for v81::xs)) ∧
    (∀v12 v82 v83 xs.
       P xs ⇒ P (v12 says v82 controls v83::xs)) ∧
    (∀v12 v84 v85 v86 xs.
       P xs ⇒ P (v12 says reps v84 v85 v86::xs)) ∧
```

$(\forall\, v_{12}\ v_{87}\ v_{88}\ xs.\ P\ xs \Rightarrow P\ (v_{12}\ \texttt{says}\ v_{87}\ \texttt{domi}\ v_{88}\texttt{::}xs)) \wedge$
$(\forall\, v_{12}\ v_{89}\ v_{90}\ xs.\ P\ xs \Rightarrow P\ (v_{12}\ \texttt{says}\ v_{89}\ \texttt{eqi}\ v_{90}\texttt{::}xs)) \wedge$
$(\forall\, v_{12}\ v_{91}\ v_{92}\ xs.\ P\ xs \Rightarrow P\ (v_{12}\ \texttt{says}\ v_{91}\ \texttt{doms}\ v_{92}\texttt{::}xs)) \wedge$
$(\forall\, v_{12}\ v_{93}\ v_{94}\ xs.\ P\ xs \Rightarrow P\ (v_{12}\ \texttt{says}\ v_{93}\ \texttt{eqs}\ v_{94}\texttt{::}xs)) \wedge$
$(\forall\, v_{12}\ v_{95}\ v_{96}\ xs.\ P\ xs \Rightarrow P\ (v_{12}\ \texttt{says}\ v_{95}\ \texttt{eqn}\ v_{96}\texttt{::}xs)) \wedge$
$(\forall\, v_{12}\ v_{97}\ v_{98}\ xs.\ P\ xs \Rightarrow P\ (v_{12}\ \texttt{says}\ v_{97}\ \texttt{lte}\ v_{98}\texttt{::}xs)) \wedge$
$(\forall\, v_{12}\ v_{99}\ v100\ xs.\ P\ xs \Rightarrow P\ (v_{12}\ \texttt{says}\ v_{99}\ \texttt{lt}\ v100\texttt{::}xs)) \wedge$
$(\forall\, v_{14}\ v_{15}\ xs.\ P\ xs \Rightarrow P\ (v_{14}\ \texttt{speaks\_for}\ v_{15}\texttt{::}xs)) \wedge$
$(\forall\, v_{16}\ v_{17}\ xs.\ P\ xs \Rightarrow P\ (v_{16}\ \texttt{controls}\ v_{17}\texttt{::}xs)) \wedge$
$(\forall\, v_{18}\ v_{19}\ v_{20}\ xs.\ P\ xs \Rightarrow P\ (\texttt{reps}\ v_{18}\ v_{19}\ v_{20}\texttt{::}xs)) \wedge$
$(\forall\, v_{21}\ v_{22}\ xs.\ P\ xs \Rightarrow P\ (v_{21}\ \texttt{domi}\ v_{22}\texttt{::}xs)) \wedge$
$(\forall\, v_{23}\ v_{24}\ xs.\ P\ xs \Rightarrow P\ (v_{23}\ \texttt{eqi}\ v_{24}\texttt{::}xs)) \wedge$
$(\forall\, v_{25}\ v_{26}\ xs.\ P\ xs \Rightarrow P\ (v_{25}\ \texttt{doms}\ v_{26}\texttt{::}xs)) \wedge$
$(\forall\, v_{27}\ v_{28}\ xs.\ P\ xs \Rightarrow P\ (v_{27}\ \texttt{eqs}\ v_{28}\texttt{::}xs)) \wedge$
$(\forall\, v_{29}\ v_{30}\ xs.\ P\ xs \Rightarrow P\ (v_{29}\ \texttt{eqn}\ v_{30}\texttt{::}xs)) \wedge$
$(\forall\, v_{31}\ v_{32}\ xs.\ P\ xs \Rightarrow P\ (v_{31}\ \texttt{lte}\ v_{32}\texttt{::}xs)) \wedge$
$(\forall\, v_{33}\ v_{34}\ xs.\ P\ xs \Rightarrow P\ (v_{33}\ \texttt{lt}\ v_{34}\texttt{::}xs)) \Rightarrow$
$\forall\, v.\ P\ v$

[inputOK_def]

$\vdash$ (inputOK (Name C2 says prop (SOME (CMD cmd))) $\iff$ T) $\wedge$
(inputOK (Name MunitionAvail says prop (SOME (MA ma))) $\iff$
T) $\wedge$
(inputOK (Name GPSKB says prop (SOME (KBL $locOK$))) $\iff$ T) $\wedge$
(inputOK (Name TimeKB says prop (SOME (KBT $timeOK$))) $\iff$ T) $\wedge$
(inputOK TT $\iff$ F) $\wedge$ (inputOK FF $\iff$ F) $\wedge$
(inputOK (prop v) $\iff$ F) $\wedge$ (inputOK (notf v_1) $\iff$ F) $\wedge$
(inputOK (v_2 andf v_3) $\iff$ F) $\wedge$ (inputOK (v_4 orf v_5) $\iff$ F) $\wedge$
(inputOK (v_6 impf v_7) $\iff$ F) $\wedge$ (inputOK (v_8 eqf v_9) $\iff$ F) $\wedge$
(inputOK (v_{10} says TT) $\iff$ F) $\wedge$ (inputOK (v_{10} says FF) $\iff$ F) $\wedge$
(inputOK (Name $v132$ says prop NONE) $\iff$ F) $\wedge$
(inputOK (Name (Staff $v144$) says prop (SOME $v142$)) $\iff$ F) $\wedge$
(inputOK (Name (Authority $v145$) says prop (SOME $v142$)) $\iff$
F) $\wedge$
(inputOK (Name (Role $v146$) says prop (SOME $v142$)) $\iff$ F) $\wedge$
(inputOK (Name (KeyS $v147$) says prop (SOME $v142$)) $\iff$ F) $\wedge$
(inputOK (Name (KeyA $v148$) says prop (SOME $v142$)) $\iff$ F) $\wedge$
(inputOK (Name C2 says prop (SOME (MA $v155$))) $\iff$ F) $\wedge$
(inputOK (Name C2 says prop (SOME (KBL $v156$))) $\iff$ F) $\wedge$
(inputOK (Name C2 says prop (SOME (KBT $v157$))) $\iff$ F) $\wedge$
(inputOK (Name MunitionAvail says prop (SOME (CMD $v162$))) $\iff$
F) $\wedge$
(inputOK (Name MunitionAvail says prop (SOME (KBL $v164$))) $\iff$
F) $\wedge$
(inputOK (Name MunitionAvail says prop (SOME (KBT $v165$))) $\iff$
F) $\wedge$
(inputOK (Name GPSKB says prop (SOME (CMD $v170$))) $\iff$ F) $\wedge$
(inputOK (Name GPSKB says prop (SOME (MA $v171$))) $\iff$ F) $\wedge$
(inputOK (Name GPSKB says prop (SOME (KBT $v173$))) $\iff$ F) $\wedge$
(inputOK (Name TimeKB says prop (SOME (CMD $v178$))) $\iff$ F) $\wedge$
(inputOK (Name TimeKB says prop (SOME (MA $v179$))) $\iff$ F) $\wedge$
(inputOK (Name TimeKB says prop (SOME (KBL $v180$))) $\iff$ F) $\wedge$
(inputOK ($v133$ meet $v134$ says prop v_{66}) $\iff$ F) $\wedge$
(inputOK ($v135$ quoting $v136$ says prop v_{66}) $\iff$ F) $\wedge$
(inputOK (v_{10} says notf v_{67}) $\iff$ F) $\wedge$
(inputOK (v_{10} says (v_{68} andf v_{69})) $\iff$ F) $\wedge$
(inputOK (v_{10} says (v_{70} orf v_{71})) $\iff$ F) $\wedge$
(inputOK (v_{10} says (v_{72} impf v_{73})) $\iff$ F) $\wedge$

(inputOK (v_{10} says (v_{74} eqf v_{75})) ⟺ F) ∧
(inputOK (v_{10} says v_{76} says v_{77}) ⟺ F) ∧
(inputOK (v_{10} says v_{78} speaks_for v_{79}) ⟺ F) ∧
(inputOK (v_{10} says v_{80} controls v_{81}) ⟺ F) ∧
(inputOK (v_{10} says reps v_{82} v_{83} v_{84}) ⟺ F) ∧
(inputOK (v_{10} says v_{85} domi v_{86}) ⟺ F) ∧
(inputOK (v_{10} says v_{87} eqi v_{88}) ⟺ F) ∧
(inputOK (v_{10} says v_{89} doms v_{90}) ⟺ F) ∧
(inputOK (v_{10} says v_{91} eqs v_{92}) ⟺ F) ∧
(inputOK (v_{10} says v_{93} eqn v_{94}) ⟺ F) ∧
(inputOK (v_{10} says v_{95} lte v_{96}) ⟺ F) ∧
(inputOK (v_{10} says v_{97} lt v_{98}) ⟺ F) ∧
(inputOK (v_{12} speaks_for v_{13}) ⟺ F) ∧
(inputOK (v_{14} controls v_{15}) ⟺ F) ∧
(inputOK (reps v_{16} v_{17} v_{18}) ⟺ F) ∧
(inputOK (v_{19} domi v_{20}) ⟺ F) ∧
(inputOK (v_{21} eqi v_{22}) ⟺ F) ∧
(inputOK (v_{23} doms v_{24}) ⟺ F) ∧
(inputOK (v_{25} eqs v_{26}) ⟺ F) ∧ (inputOK (v_{27} eqn v_{28}) ⟺ F) ∧
(inputOK (v_{29} lte v_{30}) ⟺ F) ∧ (inputOK (v_{31} lt v_{32}) ⟺ F)

[inputOK_ind]

⊢ ∀ P.
(∀ *cmd*. P (Name C2 says prop (SOME (CMD *cmd*)))) ∧
(∀ *ma*. P (Name MunitionAvail says prop (SOME (MA *ma*)))) ∧
(∀ *locOK*. P (Name GPSKB says prop (SOME (KBL *locOK*)))) ∧
(∀ *timeOK*. P (Name TimeKB says prop (SOME (KBT *timeOK*)))) ∧
P TT ∧ P FF ∧ (∀ v. P (prop v)) ∧ (∀ v_1. P (notf v_1)) ∧
(∀ v_2 v_3. P (v_2 andf v_3)) ∧ (∀ v_4 v_5. P (v_4 orf v_5)) ∧
(∀ v_6 v_7. P (v_6 impf v_7)) ∧ (∀ v_8 v_9. P (v_8 eqf v_9)) ∧
(∀ v_{10}. P (v_{10} says TT)) ∧ (∀ v_{10}. P (v_{10} says FF)) ∧
(∀ *v132*. P (Name *v132* says prop NONE)) ∧
(∀ *v144* *v142*. P (Name (Staff *v144*) says prop (SOME *v142*))) ∧
(∀ *v145* *v142*.
P (Name (Authority *v145*) says prop (SOME *v142*))) ∧
(∀ *v146* *v142*. P (Name (Role *v146*) says prop (SOME *v142*))) ∧
(∀ *v147* *v142*. P (Name (KeyS *v147*) says prop (SOME *v142*))) ∧
(∀ *v148* *v142*. P (Name (KeyA *v148*) says prop (SOME *v142*))) ∧
(∀ *v155*. P (Name C2 says prop (SOME (MA *v155*)))) ∧
(∀ *v156*. P (Name C2 says prop (SOME (KBL *v156*)))) ∧
(∀ *v157*. P (Name C2 says prop (SOME (KBT *v157*)))) ∧
(∀ *v162*.
P (Name MunitionAvail says prop (SOME (CMD *v162*)))) ∧
(∀ *v164*.
P (Name MunitionAvail says prop (SOME (KBL *v164*)))) ∧
(∀ *v165*.
P (Name MunitionAvail says prop (SOME (KBT *v165*)))) ∧
(∀ *v170*. P (Name GPSKB says prop (SOME (CMD *v170*)))) ∧
(∀ *v171*. P (Name GPSKB says prop (SOME (MA *v171*)))) ∧
(∀ *v173*. P (Name GPSKB says prop (SOME (KBT *v173*)))) ∧
(∀ *v178*. P (Name TimeKB says prop (SOME (CMD *v178*)))) ∧
(∀ *v179*. P (Name TimeKB says prop (SOME (MA *v179*)))) ∧
(∀ *v180*. P (Name TimeKB says prop (SOME (KBL *v180*)))) ∧
(∀ *v133* *v134* v_{66}. P (*v133* meet *v134* says prop v_{66})) ∧
(∀ *v135* *v136* v_{66}. P (*v135* quoting *v136* says prop v_{66})) ∧
(∀ v_{10} v_{67}. P (v_{10} says notf v_{67})) ∧
(∀ v_{10} v_{68} v_{69}. P (v_{10} says (v_{68} andf v_{69}))) ∧
(∀ v_{10} v_{70} v_{71}. P (v_{10} says (v_{70} orf v_{71}))) ∧

```
  (∀ v10 v72 v73. P (v10 says (v72 impf v73))) ∧
  (∀ v10 v74 v75. P (v10 says (v74 eqf v75))) ∧
  (∀ v10 v76 v77. P (v10 says v76 says v77)) ∧
  (∀ v10 v78 v79. P (v10 says v78 speaks_for v79)) ∧
  (∀ v10 v80 v81. P (v10 says v80 controls v81)) ∧
  (∀ v10 v82 v83 v84. P (v10 says reps v82 v83 v84)) ∧
  (∀ v10 v85 v86. P (v10 says v85 domi v86)) ∧
  (∀ v10 v87 v88. P (v10 says v87 eqi v88)) ∧
  (∀ v10 v89 v90. P (v10 says v89 doms v90)) ∧
  (∀ v10 v91 v92. P (v10 says v91 eqs v92)) ∧
  (∀ v10 v93 v94. P (v10 says v93 eqn v94)) ∧
  (∀ v10 v95 v96. P (v10 says v95 lte v96)) ∧
  (∀ v10 v97 v98. P (v10 says v97 lt v98)) ∧
  (∀ v12 v13. P (v12 speaks_for v13)) ∧
  (∀ v14 v15. P (v14 controls v15)) ∧
  (∀ v16 v17 v18. P (reps v16 v17 v18)) ∧
  (∀ v19 v20. P (v19 domi v20)) ∧
  (∀ v21 v22. P (v21 eqi v22)) ∧
  (∀ v23 v24. P (v23 doms v24)) ∧
  (∀ v25 v26. P (v25 eqs v26)) ∧ (∀ v27 v28. P (v27 eqn v28)) ∧
  (∀ v29 v30. P (v29 lte v30)) ∧ (∀ v31 v32. P (v31 lt v32)) ⇒
  ∀ v. P v
```

[maSensorContext_def]

```
⊢ (maSensorContext [] = TT) ∧
  (∀ xs load.
     maSensorContext
       (Name MunitionAvail says prop (SOME (MA load))::xs) =
     Name MunitionAvail controls prop (SOME (MA load))) ∧
  (∀ xs. maSensorContext (TT::xs) = maSensorContext xs) ∧
  (∀ xs. maSensorContext (FF::xs) = maSensorContext xs) ∧
  (∀ xs v2.
     maSensorContext (prop v2::xs) = maSensorContext xs) ∧
  (∀ xs v3.
     maSensorContext (notf v3::xs) = maSensorContext xs) ∧
  (∀ xs v5 v4.
     maSensorContext (v4 andf v5::xs) = maSensorContext xs) ∧
  (∀ xs v7 v6.
     maSensorContext (v6 orf v7::xs) = maSensorContext xs) ∧
  (∀ xs v9 v8.
     maSensorContext (v8 impf v9::xs) = maSensorContext xs) ∧
  (∀ xs v11 v10.
     maSensorContext (v10 eqf v11::xs) = maSensorContext xs) ∧
  (∀ xs v12.
     maSensorContext (v12 says TT::xs) = maSensorContext xs) ∧
  (∀ xs v12.
     maSensorContext (v12 says FF::xs) = maSensorContext xs) ∧
  (∀ xs v134.
     maSensorContext (Name v134 says prop NONE::xs) =
     maSensorContext xs) ∧
  (∀ xs v146 v144.
     maSensorContext
       (Name (Staff v146) says prop (SOME v144)::xs) =
     maSensorContext xs) ∧
  (∀ xs v147 v144.
     maSensorContext
       (Name (Authority v147) says prop (SOME v144)::xs) =
     maSensorContext xs) ∧
```

```
(∀ xs v148 v144 .
   maSensorContext
     (Name (Role v148) says prop (SOME v144)::xs) =
   maSensorContext xs) ∧
(∀ xs v149 v144 .
   maSensorContext
     (Name (KeyS v149) says prop (SOME v144)::xs) =
   maSensorContext xs) ∧
(∀ xs v150 v144 .
   maSensorContext
     (Name (KeyA v150) says prop (SOME v144)::xs) =
   maSensorContext xs) ∧
(∀ xs v144 .
   maSensorContext (Name C2 says prop (SOME v144)::xs) =
   maSensorContext xs) ∧
(∀ xs v156 .
   maSensorContext
     (Name MunitionAvail says prop (SOME (CMD v156))::xs) =
   maSensorContext xs) ∧
(∀ xs v158 .
   maSensorContext
     (Name MunitionAvail says prop (SOME (KBL v158))::xs) =
   maSensorContext xs) ∧
(∀ xs v159 .
   maSensorContext
     (Name MunitionAvail says prop (SOME (KBT v159))::xs) =
   maSensorContext xs) ∧
(∀ xs v144 .
   maSensorContext (Name GPSKB says prop (SOME v144)::xs) =
   maSensorContext xs) ∧
(∀ xs v144 .
   maSensorContext (Name TimeKB says prop (SOME v144)::xs) =
   maSensorContext xs) ∧
(∀ xs v68 v136 v135 .
   maSensorContext (v135 meet v136 says prop v68::xs) =
   maSensorContext xs) ∧
(∀ xs v68 v138 v137 .
   maSensorContext (v137 quoting v138 says prop v68::xs) =
   maSensorContext xs) ∧
(∀ xs v69 v12 .
   maSensorContext (v12 says notf v69::xs) =
   maSensorContext xs) ∧
(∀ xs v71 v70 v12 .
   maSensorContext (v12 says (v70 andf v71)::xs) =
   maSensorContext xs) ∧
(∀ xs v73 v72 v12 .
   maSensorContext (v12 says (v72 orf v73)::xs) =
   maSensorContext xs) ∧
(∀ xs v75 v74 v12 .
   maSensorContext (v12 says (v74 impf v75)::xs) =
   maSensorContext xs) ∧
(∀ xs v77 v76 v12 .
   maSensorContext (v12 says (v76 eqf v77)::xs) =
   maSensorContext xs) ∧
(∀ xs v79 v78 v12 .
   maSensorContext (v12 says v78 says v79::xs) =
   maSensorContext xs) ∧
(∀ xs v81 v80 v12 .
```

```
    maSensorContext (v12 says v80 speaks_for v81::xs) =
    maSensorContext xs) ∧
  (∀ xs v83 v82 v12.
    maSensorContext (v12 says v82 controls v83::xs) =
    maSensorContext xs) ∧
  (∀ xs v86 v85 v84 v12.
    maSensorContext (v12 says reps v84 v85 v86::xs) =
    maSensorContext xs) ∧
  (∀ xs v88 v87 v12.
    maSensorContext (v12 says v87 domi v88::xs) =
    maSensorContext xs) ∧
  (∀ xs v90 v89 v12.
    maSensorContext (v12 says v89 eqi v90::xs) =
    maSensorContext xs) ∧
  (∀ xs v92 v91 v12.
    maSensorContext (v12 says v91 doms v92::xs) =
    maSensorContext xs) ∧
  (∀ xs v94 v93 v12.
    maSensorContext (v12 says v93 eqs v94::xs) =
    maSensorContext xs) ∧
  (∀ xs v96 v95 v12.
    maSensorContext (v12 says v95 eqn v96::xs) =
    maSensorContext xs) ∧
  (∀ xs v98 v97 v12.
    maSensorContext (v12 says v97 lte v98::xs) =
    maSensorContext xs) ∧
  (∀ xs v99 v12 v100.
    maSensorContext (v12 says v99 lt v100::xs) =
    maSensorContext xs) ∧
  (∀ xs v15 v14.
    maSensorContext (v14 speaks_for v15::xs) =
    maSensorContext xs) ∧
  (∀ xs v17 v16.
    maSensorContext (v16 controls v17::xs) =
    maSensorContext xs) ∧
  (∀ xs v20 v19 v18.
    maSensorContext (reps v18 v19 v20::xs) =
    maSensorContext xs) ∧
  (∀ xs v22 v21.
    maSensorContext (v21 domi v22::xs) = maSensorContext xs) ∧
  (∀ xs v24 v23.
    maSensorContext (v23 eqi v24::xs) = maSensorContext xs) ∧
  (∀ xs v26 v25.
    maSensorContext (v25 doms v26::xs) = maSensorContext xs) ∧
  (∀ xs v28 v27.
    maSensorContext (v27 eqs v28::xs) = maSensorContext xs) ∧
  (∀ xs v30 v29.
    maSensorContext (v29 eqn v30::xs) = maSensorContext xs) ∧
  (∀ xs v32 v31.
    maSensorContext (v31 lte v32::xs) = maSensorContext xs) ∧
  ∀ xs v34 v33.
    maSensorContext (v33 lt v34::xs) = maSensorContext xs
```

[maSensorContext_ind]

```
⊢ ∀ P.
    P [] ∧
    (∀ load xs.
       P (Name MunitionAvail says prop (SOME (MA load))::xs)) ∧
```

$(\forall\, xs.\ P\ xs \Rightarrow P\ (\texttt{TT::}xs)) \wedge (\forall\, xs.\ P\ xs \Rightarrow P\ (\texttt{FF::}xs)) \wedge$
$(\forall\, v_2\ xs.\ P\ xs \Rightarrow P\ (\texttt{prop}\ v_2\texttt{::}xs)) \wedge$
$(\forall\, v_3\ xs.\ P\ xs \Rightarrow P\ (\texttt{notf}\ v_3\texttt{::}xs)) \wedge$
$(\forall\, v_4\ v_5\ xs.\ P\ xs \Rightarrow P\ (v_4\ \texttt{andf}\ v_5\texttt{::}xs)) \wedge$
$(\forall\, v_6\ v_7\ xs.\ P\ xs \Rightarrow P\ (v_6\ \texttt{orf}\ v_7\texttt{::}xs)) \wedge$
$(\forall\, v_8\ v_9\ xs.\ P\ xs \Rightarrow P\ (v_8\ \texttt{impf}\ v_9\texttt{::}xs)) \wedge$
$(\forall\, v_{10}\ v_{11}\ xs.\ P\ xs \Rightarrow P\ (v_{10}\ \texttt{eqf}\ v_{11}\texttt{::}xs)) \wedge$
$(\forall\, v_{12}\ xs.\ P\ xs \Rightarrow P\ (v_{12}\ \texttt{says TT::}xs)) \wedge$
$(\forall\, v_{12}\ xs.\ P\ xs \Rightarrow P\ (v_{12}\ \texttt{says FF::}xs)) \wedge$
$(\forall\, \mathit{v134}\ xs.\ P\ xs \Rightarrow P\ (\texttt{Name}\ \mathit{v134}\ \texttt{says prop NONE::}xs)) \wedge$
$(\forall\, \mathit{v146}\ \mathit{v144}\ xs.$
$\quad P\ xs \Rightarrow$
$\quad P\ (\texttt{Name (Staff}\ \mathit{v146}\texttt{) says prop (SOME}\ \mathit{v144}\texttt{)::}xs)) \wedge$
$(\forall\, \mathit{v147}\ \mathit{v144}\ xs.$
$\quad P\ xs \Rightarrow$
$\quad P\ (\texttt{Name (Authority}\ \mathit{v147}\texttt{) says prop (SOME}\ \mathit{v144}\texttt{)::}xs)) \wedge$
$(\forall\, \mathit{v148}\ \mathit{v144}\ xs.$
$\quad P\ xs \Rightarrow P\ (\texttt{Name (Role}\ \mathit{v148}\texttt{) says prop (SOME}\ \mathit{v144}\texttt{)::}xs)) \wedge$
$(\forall\, \mathit{v149}\ \mathit{v144}\ xs.$
$\quad P\ xs \Rightarrow P\ (\texttt{Name (KeyS}\ \mathit{v149}\texttt{) says prop (SOME}\ \mathit{v144}\texttt{)::}xs)) \wedge$
$(\forall\, \mathit{v150}\ \mathit{v144}\ xs.$
$\quad P\ xs \Rightarrow P\ (\texttt{Name (KeyA}\ \mathit{v150}\texttt{) says prop (SOME}\ \mathit{v144}\texttt{)::}xs)) \wedge$
$(\forall\, \mathit{v144}\ xs.\ P\ xs \Rightarrow P\ (\texttt{Name C2 says prop (SOME}\ \mathit{v144}\texttt{)::}xs)) \wedge$
$(\forall\, \mathit{v156}\ xs.$
$\quad P\ xs \Rightarrow$
$\quad P$
$\qquad (\texttt{Name MunitionAvail says prop (SOME (CMD}\ \mathit{v156}\texttt{))::}$
$\qquad\quad xs)) \wedge$
$(\forall\, \mathit{v158}\ xs.$
$\quad P\ xs \Rightarrow$
$\quad P$
$\qquad (\texttt{Name MunitionAvail says prop (SOME (KBL}\ \mathit{v158}\texttt{))::}$
$\qquad\quad xs)) \wedge$
$(\forall\, \mathit{v159}\ xs.$
$\quad P\ xs \Rightarrow$
$\quad P$
$\qquad (\texttt{Name MunitionAvail says prop (SOME (KBT}\ \mathit{v159}\texttt{))::}$
$\qquad\quad xs)) \wedge$
$(\forall\, \mathit{v144}\ xs.$
$\quad P\ xs \Rightarrow P\ (\texttt{Name GPSKB says prop (SOME}\ \mathit{v144}\texttt{)::}xs)) \wedge$
$(\forall\, \mathit{v144}\ xs.$
$\quad P\ xs \Rightarrow P\ (\texttt{Name TimeKB says prop (SOME}\ \mathit{v144}\texttt{)::}xs)) \wedge$
$(\forall\, \mathit{v135}\ \mathit{v136}\ v_{68}\ xs.$
$\quad P\ xs \Rightarrow P\ (\mathit{v135}\ \texttt{meet}\ \mathit{v136}\ \texttt{says prop}\ v_{68}\texttt{::}xs)) \wedge$
$(\forall\, \mathit{v137}\ \mathit{v138}\ v_{68}\ xs.$
$\quad P\ xs \Rightarrow P\ (\mathit{v137}\ \texttt{quoting}\ \mathit{v138}\ \texttt{says prop}\ v_{68}\texttt{::}xs)) \wedge$
$(\forall\, v_{12}\ v_{69}\ xs.\ P\ xs \Rightarrow P\ (v_{12}\ \texttt{says notf}\ v_{69}\texttt{::}xs)) \wedge$
$(\forall\, v_{12}\ v_{70}\ v_{71}\ xs.\ P\ xs \Rightarrow P\ (v_{12}\ \texttt{says}\ (v_{70}\ \texttt{andf}\ v_{71})\texttt{::}xs)) \wedge$
$(\forall\, v_{12}\ v_{72}\ v_{73}\ xs.\ P\ xs \Rightarrow P\ (v_{12}\ \texttt{says}\ (v_{72}\ \texttt{orf}\ v_{73})\texttt{::}xs)) \wedge$
$(\forall\, v_{12}\ v_{74}\ v_{75}\ xs.\ P\ xs \Rightarrow P\ (v_{12}\ \texttt{says}\ (v_{74}\ \texttt{impf}\ v_{75})\texttt{::}xs)) \wedge$
$(\forall\, v_{12}\ v_{76}\ v_{77}\ xs.\ P\ xs \Rightarrow P\ (v_{12}\ \texttt{says}\ (v_{76}\ \texttt{eqf}\ v_{77})\texttt{::}xs)) \wedge$
$(\forall\, v_{12}\ v_{78}\ v_{79}\ xs.\ P\ xs \Rightarrow P\ (v_{12}\ \texttt{says}\ v_{78}\ \texttt{says}\ v_{79}\texttt{::}xs)) \wedge$
$(\forall\, v_{12}\ v_{80}\ v_{81}\ xs.$
$\quad P\ xs \Rightarrow P\ (v_{12}\ \texttt{says}\ v_{80}\ \texttt{speaks\_for}\ v_{81}\texttt{::}xs)) \wedge$
$(\forall\, v_{12}\ v_{82}\ v_{83}\ xs.$
$\quad P\ xs \Rightarrow P\ (v_{12}\ \texttt{says}\ v_{82}\ \texttt{controls}\ v_{83}\texttt{::}xs)) \wedge$
$(\forall\, v_{12}\ v_{84}\ v_{85}\ v_{86}\ xs.$
$\quad P\ xs \Rightarrow P\ (v_{12}\ \texttt{says reps}\ v_{84}\ v_{85}\ v_{86}\texttt{::}xs)) \wedge$

```
  (∀ v12 v87 v88 xs. P xs ⇒ P (v12 says v87 domi v88::xs)) ∧
  (∀ v12 v89 v90 xs. P xs ⇒ P (v12 says v89 eqi v90::xs)) ∧
  (∀ v12 v91 v92 xs. P xs ⇒ P (v12 says v91 doms v92::xs)) ∧
  (∀ v12 v93 v94 xs. P xs ⇒ P (v12 says v93 eqs v94::xs)) ∧
  (∀ v12 v95 v96 xs. P xs ⇒ P (v12 says v95 eqn v96::xs)) ∧
  (∀ v12 v97 v98 xs. P xs ⇒ P (v12 says v97 lte v98::xs)) ∧
  (∀ v12 v99 v100 xs. P xs ⇒ P (v12 says v99 lt v100::xs)) ∧
  (∀ v14 v15 xs. P xs ⇒ P (v14 speaks_for v15::xs)) ∧
  (∀ v16 v17 xs. P xs ⇒ P (v16 controls v17::xs)) ∧
  (∀ v18 v19 v20 xs. P xs ⇒ P (reps v18 v19 v20::xs)) ∧
  (∀ v21 v22 xs. P xs ⇒ P (v21 domi v22::xs)) ∧
  (∀ v23 v24 xs. P xs ⇒ P (v23 eqi v24::xs)) ∧
  (∀ v25 v26 xs. P xs ⇒ P (v25 doms v26::xs)) ∧
  (∀ v27 v28 xs. P xs ⇒ P (v27 eqs v28::xs)) ∧
  (∀ v29 v30 xs. P xs ⇒ P (v29 eqn v30::xs)) ∧
  (∀ v31 v32 xs. P xs ⇒ P (v31 lte v32::xs)) ∧
  (∀ v33 v34 xs. P xs ⇒ P (v33 lt v34::xs)) ⇒
  ∀ v. P v
```

[tkbSensorContext_def]

```
⊢ (tkbSensorContext [] = TT) ∧
  (∀ xs time.
     tkbSensorContext
       (Name TimeKB says prop (SOME (KBT time))::xs) =
     Name TimeKB controls prop (SOME (KBT time))) ∧
  (∀ xs. tkbSensorContext (TT::xs) = tkbSensorContext xs) ∧
  (∀ xs. tkbSensorContext (FF::xs) = tkbSensorContext xs) ∧
  (∀ xs v2.
     tkbSensorContext (prop v2::xs) = tkbSensorContext xs) ∧
  (∀ xs v3.
     tkbSensorContext (notf v3::xs) = tkbSensorContext xs) ∧
  (∀ xs v5 v4.
     tkbSensorContext (v4 andf v5::xs) = tkbSensorContext xs) ∧
  (∀ xs v7 v6.
     tkbSensorContext (v6 orf v7::xs) = tkbSensorContext xs) ∧
  (∀ xs v9 v8.
     tkbSensorContext (v8 impf v9::xs) = tkbSensorContext xs) ∧
  (∀ xs v11 v10.
     tkbSensorContext (v10 eqf v11::xs) =
     tkbSensorContext xs) ∧
  (∀ xs v12.
     tkbSensorContext (v12 says TT::xs) =
     tkbSensorContext xs) ∧
  (∀ xs v12.
     tkbSensorContext (v12 says FF::xs) =
     tkbSensorContext xs) ∧
  (∀ xs v134.
     tkbSensorContext (Name v134 says prop NONE::xs) =
     tkbSensorContext xs) ∧
  (∀ xs v146 v144.
     tkbSensorContext
       (Name (Staff v146) says prop (SOME v144)::xs) =
     tkbSensorContext xs) ∧
  (∀ xs v147 v144.
     tkbSensorContext
       (Name (Authority v147) says prop (SOME v144)::xs) =
     tkbSensorContext xs) ∧
  (∀ xs v148 v144.
```

```
   tkbSensorContext
     (Name (Role v148) says prop (SOME v144)::xs) =
   tkbSensorContext xs) ∧
(∀xs v149 v144.
   tkbSensorContext
     (Name (KeyS v149) says prop (SOME v144)::xs) =
   tkbSensorContext xs) ∧
(∀xs v150 v144.
   tkbSensorContext
     (Name (KeyA v150) says prop (SOME v144)::xs) =
   tkbSensorContext xs) ∧
(∀xs v144.
   tkbSensorContext (Name C2 says prop (SOME v144)::xs) =
   tkbSensorContext xs) ∧
(∀xs v144.
   tkbSensorContext
     (Name MunitionAvail says prop (SOME v144)::xs) =
   tkbSensorContext xs) ∧
(∀xs v144.
   tkbSensorContext (Name GPSKB says prop (SOME v144)::xs) =
   tkbSensorContext xs) ∧
(∀xs v156.
   tkbSensorContext
     (Name TimeKB says prop (SOME (CMD v156))::xs) =
   tkbSensorContext xs) ∧
(∀xs v157.
   tkbSensorContext
     (Name TimeKB says prop (SOME (MA v157))::xs) =
   tkbSensorContext xs) ∧
(∀xs v158.
   tkbSensorContext
     (Name TimeKB says prop (SOME (KBL v158))::xs) =
   tkbSensorContext xs) ∧
(∀xs v68 v136 v135.
   tkbSensorContext (v135 meet v136 says prop v68::xs) =
   tkbSensorContext xs) ∧
(∀xs v68 v138 v137.
   tkbSensorContext (v137 quoting v138 says prop v68::xs) =
   tkbSensorContext xs) ∧
(∀xs v69 v12.
   tkbSensorContext (v12 says notf v69::xs) =
   tkbSensorContext xs) ∧
(∀xs v71 v70 v12.
   tkbSensorContext (v12 says (v70 andf v71)::xs) =
   tkbSensorContext xs) ∧
(∀xs v73 v72 v12.
   tkbSensorContext (v12 says (v72 orf v73)::xs) =
   tkbSensorContext xs) ∧
(∀xs v75 v74 v12.
   tkbSensorContext (v12 says (v74 impf v75)::xs) =
   tkbSensorContext xs) ∧
(∀xs v77 v76 v12.
   tkbSensorContext (v12 says (v76 eqf v77)::xs) =
   tkbSensorContext xs) ∧
(∀xs v79 v78 v12.
   tkbSensorContext (v12 says v78 says v79::xs) =
   tkbSensorContext xs) ∧
(∀xs v81 v80 v12.
```

```
   tkbSensorContext (v12 says v80 speaks_for v81::xs) =
   tkbSensorContext xs) ∧
(∀ xs v83 v82 v12.
   tkbSensorContext (v12 says v82 controls v83::xs) =
   tkbSensorContext xs) ∧
(∀ xs v86 v85 v84 v12.
   tkbSensorContext (v12 says reps v84 v85 v86::xs) =
   tkbSensorContext xs) ∧
(∀ xs v88 v87 v12.
   tkbSensorContext (v12 says v87 domi v88::xs) =
   tkbSensorContext xs) ∧
(∀ xs v90 v89 v12.
   tkbSensorContext (v12 says v89 eqi v90::xs) =
   tkbSensorContext xs) ∧
(∀ xs v92 v91 v12.
   tkbSensorContext (v12 says v91 doms v92::xs) =
   tkbSensorContext xs) ∧
(∀ xs v94 v93 v12.
   tkbSensorContext (v12 says v93 eqs v94::xs) =
   tkbSensorContext xs) ∧
(∀ xs v96 v95 v12.
   tkbSensorContext (v12 says v95 eqn v96::xs) =
   tkbSensorContext xs) ∧
(∀ xs v98 v97 v12.
   tkbSensorContext (v12 says v97 lte v98::xs) =
   tkbSensorContext xs) ∧
(∀ xs v99 v12 v100.
   tkbSensorContext (v12 says v99 lt v100::xs) =
   tkbSensorContext xs) ∧
(∀ xs v15 v14.
   tkbSensorContext (v14 speaks_for v15::xs) =
   tkbSensorContext xs) ∧
(∀ xs v17 v16.
   tkbSensorContext (v16 controls v17::xs) =
   tkbSensorContext xs) ∧
(∀ xs v20 v19 v18.
   tkbSensorContext (reps v18 v19 v20::xs) =
   tkbSensorContext xs) ∧
(∀ xs v22 v21.
   tkbSensorContext (v21 domi v22::xs) =
   tkbSensorContext xs) ∧
(∀ xs v24 v23.
   tkbSensorContext (v23 eqi v24::xs) =
   tkbSensorContext xs) ∧
(∀ xs v26 v25.
   tkbSensorContext (v25 doms v26::xs) =
   tkbSensorContext xs) ∧
(∀ xs v28 v27.
   tkbSensorContext (v27 eqs v28::xs) =
   tkbSensorContext xs) ∧
(∀ xs v30 v29.
   tkbSensorContext (v29 eqn v30::xs) =
   tkbSensorContext xs) ∧
(∀ xs v32 v31.
   tkbSensorContext (v31 lte v32::xs) =
   tkbSensorContext xs) ∧
∀ xs v34 v33.
  tkbSensorContext (v33 lt v34::xs) = tkbSensorContext xs
```

[tkbSensorContext_ind]

$\vdash \forall P.$
P [] $\wedge$
$(\forall time\ xs.$
P (Name TimeKB says prop (SOME (KBT $time$))::xs)) $\wedge$
$(\forall xs.\ P\ xs \Rightarrow P$ (TT::xs)) $\wedge$ $(\forall xs.\ P\ xs \Rightarrow P$ (FF::xs)) $\wedge$
$(\forall v_2\ xs.\ P\ xs \Rightarrow P$ (prop v_2::xs)) $\wedge$
$(\forall v_3\ xs.\ P\ xs \Rightarrow P$ (notf v_3::xs)) $\wedge$
$(\forall v_4\ v_5\ xs.\ P\ xs \Rightarrow P$ (v_4 andf v_5::xs)) $\wedge$
$(\forall v_6\ v_7\ xs.\ P\ xs \Rightarrow P$ (v_6 orf v_7::xs)) $\wedge$
$(\forall v_8\ v_9\ xs.\ P\ xs \Rightarrow P$ (v_8 impf v_9::xs)) $\wedge$
$(\forall v_{10}\ v_{11}\ xs.\ P\ xs \Rightarrow P$ (v_{10} eqf v_{11}::xs)) $\wedge$
$(\forall v_{12}\ xs.\ P\ xs \Rightarrow P$ (v_{12} says TT::xs)) $\wedge$
$(\forall v_{12}\ xs.\ P\ xs \Rightarrow P$ (v_{12} says FF::xs)) $\wedge$
$(\forall v134\ xs.\ P\ xs \Rightarrow P$ (Name $v134$ says prop NONE::xs)) $\wedge$
$(\forall v146\ v144\ xs.$
$P\ xs \Rightarrow$
P (Name (Staff $v146$) says prop (SOME $v144$)::xs)) $\wedge$
$(\forall v147\ v144\ xs.$
$P\ xs \Rightarrow$
P (Name (Authority $v147$) says prop (SOME $v144$)::xs)) $\wedge$
$(\forall v148\ v144\ xs.$
$P\ xs \Rightarrow P$ (Name (Role $v148$) says prop (SOME $v144$)::xs)) $\wedge$
$(\forall v149\ v144\ xs.$
$P\ xs \Rightarrow P$ (Name (KeyS $v149$) says prop (SOME $v144$)::xs)) $\wedge$
$(\forall v150\ v144\ xs.$
$P\ xs \Rightarrow P$ (Name (KeyA $v150$) says prop (SOME $v144$)::xs)) $\wedge$
$(\forall v144\ xs.\ P\ xs \Rightarrow P$ (Name C2 says prop (SOME $v144$)::xs)) $\wedge$
$(\forall v144\ xs.$
$P\ xs \Rightarrow$
P (Name MunitionAvail says prop (SOME $v144$)::xs)) $\wedge$
$(\forall v144\ xs.$
$P\ xs \Rightarrow P$ (Name GPSKB says prop (SOME $v144$)::xs)) $\wedge$
$(\forall v156\ xs.$
$P\ xs \Rightarrow$
P (Name TimeKB says prop (SOME (CMD $v156$))::xs)) $\wedge$
$(\forall v157\ xs.$
$P\ xs \Rightarrow P$ (Name TimeKB says prop (SOME (MA $v157$))::xs)) $\wedge$
$(\forall v158\ xs.$
$P\ xs \Rightarrow$
P (Name TimeKB says prop (SOME (KBL $v158$))::xs)) $\wedge$
$(\forall v135\ v136\ v_{68}\ xs.$
$P\ xs \Rightarrow P$ ($v135$ meet $v136$ says prop v_{68}::xs)) $\wedge$
$(\forall v137\ v138\ v_{68}\ xs.$
$P\ xs \Rightarrow P$ ($v137$ quoting $v138$ says prop v_{68}::xs)) $\wedge$
$(\forall v_{12}\ v_{69}\ xs.\ P\ xs \Rightarrow P$ (v_{12} says notf v_{69}::xs)) $\wedge$
$(\forall v_{12}\ v_{70}\ v_{71}\ xs.\ P\ xs \Rightarrow P$ (v_{12} says (v_{70} andf v_{71})::xs)) $\wedge$
$(\forall v_{12}\ v_{72}\ v_{73}\ xs.\ P\ xs \Rightarrow P$ (v_{12} says (v_{72} orf v_{73})::xs)) $\wedge$
$(\forall v_{12}\ v_{74}\ v_{75}\ xs.\ P\ xs \Rightarrow P$ (v_{12} says (v_{74} impf v_{75})::xs)) $\wedge$
$(\forall v_{12}\ v_{76}\ v_{77}\ xs.\ P\ xs \Rightarrow P$ (v_{12} says (v_{76} eqf v_{77})::xs)) $\wedge$
$(\forall v_{12}\ v_{78}\ v_{79}\ xs.\ P\ xs \Rightarrow P$ (v_{12} says v_{78} says v_{79}::xs)) $\wedge$
$(\forall v_{12}\ v_{80}\ v_{81}\ xs.$
$P\ xs \Rightarrow P$ (v_{12} says v_{80} speaks_for v_{81}::xs)) $\wedge$
$(\forall v_{12}\ v_{82}\ v_{83}\ xs.$
$P\ xs \Rightarrow P$ (v_{12} says v_{82} controls v_{83}::xs)) $\wedge$
$(\forall v_{12}\ v_{84}\ v_{85}\ v_{86}\ xs.$
$P\ xs \Rightarrow P$ (v_{12} says reps v_{84} v_{85} v_{86}::xs)) $\wedge$
$(\forall v_{12}\ v_{87}\ v_{88}\ xs.\ P\ xs \Rightarrow P$ (v_{12} says v_{87} domi v_{88}::xs)) $\wedge$

```
(∀ v12 v89 v90 xs. P xs ⇒ P (v12 says v89 eqi v90::xs)) ∧
(∀ v12 v91 v92 xs. P xs ⇒ P (v12 says v91 doms v92::xs)) ∧
(∀ v12 v93 v94 xs. P xs ⇒ P (v12 says v93 eqs v94::xs)) ∧
(∀ v12 v95 v96 xs. P xs ⇒ P (v12 says v95 eqn v96::xs)) ∧
(∀ v12 v97 v98 xs. P xs ⇒ P (v12 says v97 lte v98::xs)) ∧
(∀ v12 v99 v100 xs. P xs ⇒ P (v12 says v99 lt v100::xs)) ∧
(∀ v14 v15 xs. P xs ⇒ P (v14 speaks_for v15::xs)) ∧
(∀ v16 v17 xs. P xs ⇒ P (v16 controls v17::xs)) ∧
(∀ v18 v19 v20 xs. P xs ⇒ P (reps v18 v19 v20::xs)) ∧
(∀ v21 v22 xs. P xs ⇒ P (v21 domi v22::xs)) ∧
(∀ v23 v24 xs. P xs ⇒ P (v23 eqi v24::xs)) ∧
(∀ v25 v26 xs. P xs ⇒ P (v25 doms v26::xs)) ∧
(∀ v27 v28 xs. P xs ⇒ P (v27 eqs v28::xs)) ∧
(∀ v29 v30 xs. P xs ⇒ P (v29 eqn v30::xs)) ∧
(∀ v31 v32 xs. P xs ⇒ P (v31 lte v32::xs)) ∧
(∀ v33 v34 xs. P xs ⇒ P (v33 lt v34::xs)) ⇒
∀ v. P v
```

[uavM0_LlRe_discard_MA_inject_RL_thm]

```
⊢ ∀ M Oi Os.
    TR (M,Oi,Os)
      (discard
         (inputList
            [Name MunitionAvail says prop (SOME (CMD RL));
             Name GPSKB says prop (SOME (KBL F));
             Name TimeKB says prop (SOME (KBT T))]))
      (CFG inputOK cmdAuthorizeContext sensorContext
         ([Name MunitionAvail says prop (SOME (CMD RL));
           Name GPSKB says prop (SOME (KBL F));
           Name TimeKB says prop (SOME (KBT T))]::ins) LlRe
         outs)
      (CFG inputOK cmdAuthorizeContext sensorContext ins LlRe
         (exec [NONE]::outs))
```

[uavM0_LlRe_exec_RL_in_KillBox_thm]

```
⊢ ∀ M Oi Os.
    TR (M,Oi,Os)
      (exec
         [SOME (CMD RL); SOME (MA L); SOME (KBL T);
          SOME (KBT T)])
      (CFG inputOK cmdAuthorizeContext sensorContext
         ([Name C2 says prop (SOME (CMD RL));
           Name MunitionAvail says prop (SOME (MA L));
           Name GPSKB says prop (SOME (KBL T));
           Name TimeKB says prop (SOME (KBT T))]::ins) LlRe
         outs)
      (CFG inputOK cmdAuthorizeContext sensorContext ins LeRe
         (exec [SOME (CMD RL)]::outs)) ⟺
    authenticationTest inputOK
      [Name C2 says prop (SOME (CMD RL));
       Name MunitionAvail says prop (SOME (MA L));
       Name GPSKB says prop (SOME (KBL T));
       Name TimeKB says prop (SOME (KBT T))] ∧
    CFGInterpret (M,Oi,Os)
      (CFG inputOK cmdAuthorizeContext sensorContext
         ([Name C2 says prop (SOME (CMD RL));
           Name MunitionAvail says prop (SOME (MA L));
           Name GPSKB says prop (SOME (KBL T));
```

```
          Name TimeKB says prop (SOME (KBT T))]::ins) LlRe
        outs) ∧
    (M,Oi,Os) satList
    [prop (SOME (CMD RL)); prop (SOME (MA L));
     prop (SOME (KBL T)); prop (SOME (KBT T))]
```

[uavM0_LlRe_NOP_in_KillBox_thm]

```
⊢ ∀M Oi Os.
    TR (M,Oi,Os)
      (exec [SOME (MA L); SOME (KBL T); SOME (KBT T)])
      (CFG inputOK cmdAuthorizeContext sensorContext
         ([Name MunitionAvail says prop (SOME (MA L));
           Name GPSKB says prop (SOME (KBL T));
           Name TimeKB says prop (SOME (KBT T))]::ins) LlRe
         outs)
      (CFG inputOK cmdAuthorizeContext sensorContext ins LlRe
         (exec [NONE]::outs)) ⟺
    authenticationTest inputOK
      [Name MunitionAvail says prop (SOME (MA L));
       Name GPSKB says prop (SOME (KBL T));
       Name TimeKB says prop (SOME (KBT T))] ∧
    CFGInterpret (M,Oi,Os)
      (CFG inputOK cmdAuthorizeContext sensorContext
         ([Name MunitionAvail says prop (SOME (MA L));
           Name GPSKB says prop (SOME (KBL T));
           Name TimeKB says prop (SOME (KBT T))]::ins) LlRe
         outs) ∧
    (M,Oi,Os) satList
    [prop (SOME (MA L)); prop (SOME (KBL T));
     prop (SOME (KBT T))]
```

[uavM0_LlRe_trap_RL_outside_KillBox_thm]

```
⊢ ∀M Oi Os.
    TR (M,Oi,Os)
      (trap
         [SOME (CMD RL); SOME (MA L); SOME (KBL F);
          SOME (KBT T)])
      (CFG inputOK cmdAuthorizeContext sensorContext
         ([Name C2 says prop (SOME (CMD RL));
           Name MunitionAvail says prop (SOME (MA L));
           Name GPSKB says prop (SOME (KBL F));
           Name TimeKB says prop (SOME (KBT T))]::ins) LlRe
         outs)
      (CFG inputOK cmdAuthorizeContext sensorContext ins LlRe
         (exec [NONE]::outs)) ⟺
    authenticationTest inputOK
      [Name C2 says prop (SOME (CMD RL));
       Name MunitionAvail says prop (SOME (MA L));
       Name GPSKB says prop (SOME (KBL F));
       Name TimeKB says prop (SOME (KBT T))] ∧
    CFGInterpret (M,Oi,Os)
      (CFG inputOK cmdAuthorizeContext sensorContext
         ([Name C2 says prop (SOME (CMD RL));
           Name MunitionAvail says prop (SOME (MA L));
           Name GPSKB says prop (SOME (KBL F));
           Name TimeKB says prop (SOME (KBT T))]::ins) LlRe
         outs) ∧ (M,Oi,Os) sat prop NONE
```

Appendix C

Modeling Cryptographic Operations in HOL

This chapter describes and algebraic model of cryptographic operations. The model supports the kind of algebraic reasoning behind the use of cryptographic operations for authentication. What is presented here is based on *cipher Theory*, which appears in Appendix D.

The description below is excerpted from Chapter 15, *Certified Security by Design Using Higher Order Logic* [14].

C.1 Properties, Reality, Purposes, and Models

We introduce the basic cryptographic operations of encryption and hashing in much the way computer hardware engineers view logic gates as components, software engineers view system calls and subroutines as building blocks, or users view applications. In what follows, we model cryptographic components and prove properties of the models. All models are simplified, incomplete, and inaccurate descriptions of reality. However, they are useful when their are "close enough" descriptions of reality and simplify the design and verification tasks of engineers and computer scientists.

Properties We view encryption and decryption algorithms as cryptographic components that hide or reveal information using cryptographic keys. *Symmetric*-key cryptographic components use the same key to hide and reveal information. *Asymmetric*-key cryptographic components use different keys to hide and reveal information. These are properties shared by all encryption methods. Encryption algorithms are deemed to be *strong* if the amount of effort required to reveal hidden information or deduce cryptographic keys is impractical.

Cryptographic hash functions take arbitrarily large inputs and produce fixed-sized outputs that are used to uniquely identify the input. A hash function is deemed to be *one way* if it is impractical to reverse, i.e., for any given hash value, it is impractical to determine an input that produces the given hash value.

Reality At the logic design and architecture levels of hardware design, details of signal delays and non-switch-level behavior are omitted. In reality, latches enter meta-stable states, where its outputs are neither zero nor one. Thus, gate level and register-transfer level descriptions are incomplete and imperfect models of reality. Nonetheless, their proven utility makes them essential design and verification tools for hardware design.

Cryptographic components are similar. Hash functions associate an infinite number of inputs with a finite number of hash values. When more than one input has the same hash value, this is called a *collision*. If collisions are impractical to predict and construct, we assume in practice that if two inputs have the same hash value, then the two inputs are identical. In reality, this is a crude approximation and different cryptographic algorithms have different strengths. Nevertheless, just as we ignore the details of meta-stability and time delays in digital hardware, we assume that the cryptographic strength of our components is sufficient for our purposes.

Purposes Cryptographic components are used for two purposes:

1. *Assurance of integrity:* information and commands are authenticated, i.e., their sources are known and their contents are free from corruption, and

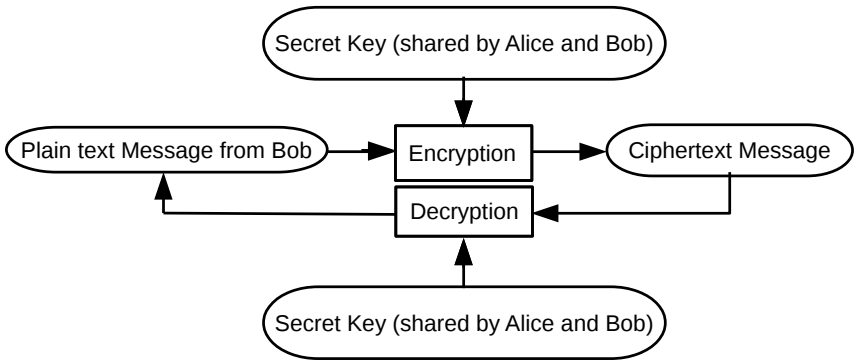


Figure C.1: Symmetric-Key Encryption and Decryption

2. *Assurance of confidentiality:* information and commands are accessible to only those who need to know.

How the above purposes are accomplished using the properties of cryptographic components is the subject of this laboratory. For example, we will show that one method to authenticate information is to combine a cryptographic hash with asymmetric encryption.

Models In what follows, we model cryptographic operations *algebraically*. We use symbols to represent cryptographic functions such as encryption functions, hash functions, keys, plain (unencrypted) text, and cipher (encrypted) text. The use of hash functions and encryption functions is described using algebraic data types. Accessing or decrypting the encrypted information using keys is defined by functions operating on datatypes corresponding to encrypted information.

The above approach leads to theorems describing the behavior of cryptographic operations based on the assumed properties of cryptographic components.

C.2 An Algebraic Model of Symmetric Key Encryption in HOL

Figure C.1 is a schematic of symmetric key encryption and decryption. Suppose Bob wishes to send a message to Alice that only he and Alice can read. Also suppose that Bob and Alice share the same secret key, which is also known as a *symmetric key*. Here are the steps that Bob and Alice take to communicate confidentially.

1. Bob *encrypts* his message in plaintext with the secret key k he shares with Alice. He forwards to encrypted message, i.e., the ciphertext, to Alice.

2. Alice uses symmetric key k to decrypt the ciphertext to retrieve the plaintext message.

C.2.1 Idealized Behavior

Symmetric-key cryptography is used with the following expectations: (1) the same key is the only means to decrypt what is encrypted, (2) if something useful and recognizable is decrypted, then it must mean that the decrypted text and the decryption key are identical to the original text and encryption key, and (3) using anything other than the original encryption key to decrypt will result in an unusable result. We capture these expectations semi-formally by the following statements.

1. Whatever is encrypted with key k is retrieved unchanged by decrypting with the same key k.

2. If key k_1 encrypted any plaintext, and key k_2 decrypted the resulting ciphertext and retrieved the original text, then $k_1 = k_2$.

```
option = NONE | SOME 'a

[option_CLAUSES]

⊢ (∀x y. (SOME x = SOME y) ⟺ (x = y)) ∧
  (∀x. THE (SOME x) = x) ∧ (∀x. NONE ≠ SOME x) ∧
  (∀x. SOME x ≠ NONE) ∧ (∀x. IS_SOME (SOME x) ⟺ T) ∧
  (IS_SOME NONE ⟺ F) ∧ (∀x. IS_NONE x ⟺ (x = NONE)) ∧
  (∀x. ¬IS_SOME x ⟺ (x = NONE)) ∧
  (∀x. IS_SOME x ⇒ (SOME (THE x) = x)) ∧
  (∀x. option_CASE x NONE SOME = x) ∧
  (∀x. option_CASE x x SOME = x) ∧
  (∀x. IS_NONE x ⇒ (option_CASE x e f = e)) ∧
  (∀x. IS_SOME x ⇒ (option_CASE x e f = f (THE x))) ∧
  (∀x. IS_SOME x ⇒ (option_CASE x e SOME = x)) ∧
  (∀v f. option_CASE NONE v f = v) ∧
  (∀x v f. option_CASE (SOME x) v f = f x) ∧
  (∀f x. OPTION_MAP f (SOME x) = SOME (f x)) ∧
  (∀f. OPTION_MAP f NONE = NONE) ∧ (OPTION_JOIN NONE = NONE) ∧
  ∀x. OPTION_JOIN (SOME x) = x
```

Figure C.2: Option Theory in HOL

3. If plaintext is encrypted with key k_1, decrypted with key k_2, and nothing useful results, then $k_1 \neq k_2$.

4. If nothing useful is encrypted using any key, then nothing useful is decrypted using any key.

C.2.2 Modeling Idealized Behavior in HOL

Adding "Nothing Useful" as a Value

One aspect we must model is the notion of "nothing useful" as a value or result. To do this in a general fashion, we use *option* theory in HOL. Figure C.2 shows the type definition of *option* and the properties of *option* types in HOL in the theorem *option_CLAUSES*.

The *option* type is polymorphic. *option* types are created from other types using the type constructor *SOME*. For example, when *SOME* is applied to the natural number 1, i.e., *SOME 1*, the resulting value is of type *num option*. The *num option* type has all the values of *SOME n*, where n is a natural number in HOL, with one added value: *NONE*. We use *NONE* when we want to return a value other than a natural number, e.g., in the case where we return a result of dividing by zero.

In the case of modeling encryption and decryption, we use *option* types to add the value *NONE* to whatever we are encrypting or decrypting. Doing so allows us to handle cases such as what value to return if the wrong key is used to decrypt an encrypted message.

Finally, the accessor function *THE* is used to retrieve the value to which *SOME* is applied. For example, $THE(SOME\ x) = x$, as shown in *option_CLAUSES*.

Symmetric Keys, Encryption, Decryption, and their Properties

Figure C.3 shows the definitions and properties of symmetric-key encryption and decryption. The following is a list of key definitions and properties.

- Symmetric keys are modeled by the algebraic type *symKey*. The type constructor is *sym*. For example, `sym 1234` is a symmetric key. Abstractly, `sym 1234` is the symmetric key which is identified by number 1234.

symKey = sym num

[symKey_one_one]

$\vdash \forall a\ a'.$ (sym a = sym a') $\iff$ (a = a')

symMsg = Es symKey ('message option)

[symMsg_one_one]

$\vdash \forall a_0\ a_1\ a_0'\ a_1'.$
(Es a_0 a_1 = Es a_0' a_1') $\iff$ (a_0 = a_0') $\wedge$ (a_1 = a_1')

[deciphS_def]

$\vdash$ (deciphS k_1 (Es k_2 (SOME x)) =
if k_1 = k_2 **then** SOME x **else** NONE) $\wedge$
(deciphS k_1 (Es k_2 NONE) = NONE)

[deciphS_clauses]

$\vdash$ ($\forall k\ text.$ deciphS k (Es k (SOME $text$)) = SOME $text$) $\wedge$
($\forall k_1\ k_2\ text.$
(deciphS k_1 (Es k_2 (SOME $text$)) = SOME $text$) $\iff$
(k_1 = k_2)) $\wedge$
($\forall k_1\ k_2\ text.$
(deciphS k_1 (Es k_2 (SOME $text$)) = NONE) $\iff$ $k_1 \neq k_2$) $\wedge$
$\forall k_1\ k_2.$ deciphS k_1 (Es k_2 NONE) = NONE

[deciphS_one_one]

$\vdash$ ($\forall k_1\ k_2\ text_1\ text_2.$
(deciphS k_1 (Es k_2 (SOME $text_2$)) = SOME $text_1$) $\iff$
(k_1 = k_2) $\wedge$ ($text_1$ = $text_2$)) $\wedge$
$\forall enMsg\ text\ key.$
(deciphS key $enMsg$ = SOME $text$) $\iff$
($enMsg$ = Es key (SOME $text$))

Figure C.3: Definitions and Properties of Symmetric Encryption and Decryption

- Two symmetric keys are identical if they have the same number to which *sym* is applied. This is shown in theorem *symKey_one_one*.
- Symmetrically encrypted messages are modeled by the algebraic type *symMsg*, whose type constructor is *Es*. Symmetrically encrypted messages have two arguments: (1) a `symKey`, and (2) a `'message option`. For example, `Es (sym 1234) (SOME "This is a string")` is a symmetrically encrypted message using: (1) the symmetric key `sym 1234`, and (2) the *string option* value `SOME "This is a string"`. Abstractly, the type constructor *Es* stands for any symmetric-key encryption algorithm, e.g., DES or AES.
- Two *symMsg* values are identical if their corresponding components are identical. This is shown in theorem *symMsg_one_one*.
- Symmetric-key decryption of *symMsgs* is defined by *deciphS_def*. If the same *symKey* is used to decipher an encrypted *SOME x*, then *SOME x* is returned. Otherwise, *NONE* is returned. If nothing useful is encrypted, then nothing useful is decrypted. Abstractly, *deciphS* represents any symmetric key decryption algorithm.
- Finally, *deciphS_clauses* is the HOL theorem that shows our type definitions for keys and encryption,

digest = hash ('message option)

[digest_one_one]

⊢ ∀ *a* *a'*. (hash *a* = hash *a'*) ⟺ (*a* = *a'*)

Figure C.4: Definition of Digests and their Properties

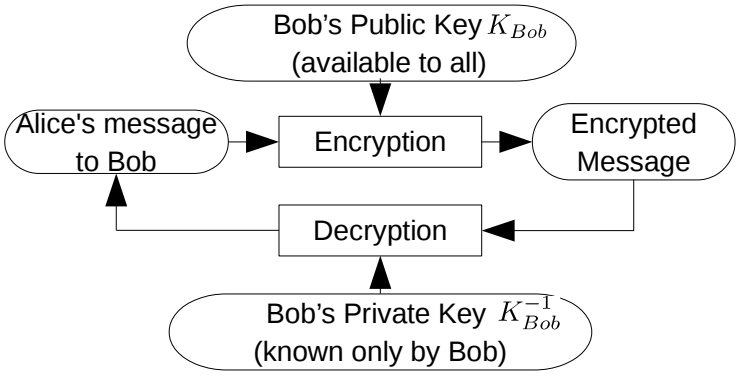


Figure C.5: Asymmetric-Key Encryption and Decryption

coupled with our definition of decryption, has the properties we expect: (1) the same key when used for encryption and decryption returns the original message, (2) if the original message was retrieved, identical keys were used, (3) if a different key is used to decrypt ciphertext, then nothing useful is returned, and (4) garbage in and garbage out holds true.

C.3 Cryptographic Hash Functions

Cryptographic hash functions are used to map inputs of any size into a fixed number of bits. Cryptographic hash functions are one-way functions, (1) the output is easy to compute from the input, and (2) it is computationally infeasible to determine an input when given only a hash value. Hash values are also known as *digests*.

Figure C.4 shows the type definition of *digest* and their properties. The following describes the type definition and its properties.

- Digests or hashes are modeled by the algebraic type *digest*. The type constructor is *hash* and is meant to represent any hash algorithm, e.g., SHA1 and SHA2. Notice that the *hash* is applied to polymorphic arguments of type `'message option`, e.g., `hash (SOME "A string message")`.
- The key property of *ideal* digests is they are one-to-one, as shown by the theorem *digest_one_one*. In reality, hashes cannot be one-to-one due to their fixed-length output. Modeling digests in this way is analogous to abstracting the electrical behavior of transistors as amplifiers away and idealizing them as perfect switches.

C.4 Asymmetric-Key Cryptography

Figure C.5 is a schematic of asymmetric key encryption and decryption. The asymmetric nature of asymmetric key, or public-key cryptography, is two different keys are used instead of the same key. One key, known as a *public key*, may be freely disclosed. The other key, known as a *private key*, must be *known only by one principal*.

```
pKey = pubK 'princ | privK 'princ
```

[pKey_distinct_clauses]

```
⊢ (∀ a' a. pubK a ≠ privK a') ∧ ∀ a' a. privK a' ≠ pubK a
```

[pKey_one_one]

```
⊢ (∀ a a'. (pubK a = pubK a') ⟺ (a = a')) ∧
  ∀ a a'. (privK a = privK a') ⟺ (a = a')
```

```
asymMsg = Ea ('princ pKey) ('message option)
```

[asymMsg_one_one]

```
⊢ ∀ a0 a1 a0' a1'.
    (Ea a0 a1 = Ea a0' a1') ⟺ (a0 = a0') ∧ (a1 = a1')
```

Figure C.6: Definitions and Properties of Asymmetric Keys and Messages

Suppose Alice wishes to send a message to Bob that only Bob can read. Alice encrypts the message to Bob using his public key K_{Bob}. Only Bob, who alone possesses the private key K_{Bob}^{-1}, is able to decrypt the message encrypted with his public key K_{Bob}.

Asymmetric-key cryptography is used with the following expectations: (1) plaintext that is encrypted with a private key and can be retrieved only with the corresponding public key, (2) plaintext that is encrypted with a public key can be retrieved only with the corresponding private key, (3) if plaintext was retrieved that was encrypted with a private key, then the corresponding public key was was used to decrypt the ciphertext, (4) if plaintext was retrieved that was encrypted with a public key, then the corresponding private key was used to decrypt the ciphertext, and (5) nothing useful results if decryption uses anything but the corresponding public or private key used in encryption.

Figure C.6 shows the type definitions for asymmetric keys *pKey*, i.e., public and private keys, and asymmetrically encrypted messages *asymMsg*. Figure C.6 also shows properties of *pKey* and *asymMsg*.

- The type *pKey* has two forms, *pubK P* and *privK P*, public and private, respectively. Asymmetric keys are polymorphic and intended to be associated with principals *P* with variable type *'princ*.
- The private and public keys of any principal are not the same.
- Public and private keys are the same if they have the same parameters.
- The type *asymMsg* represents asymmetrically encrypted messages. The parameters of type constructor *Ea* are a *pKey* and a *'message option*. Abstractly, the type constructor *Ea* stands for any asymmetric-key algorithm, e.g., RSA.
- Two *asymMsgs* are the same if they have the same *pKey* and *'message option* values.

Figure C.7 shows the definition and properties of *deciphP*, which models the decryption of asymmetrically encrypted messages. Similar to symmetric-key encryption, to retrieve the plaintext *SOME x* requires use of the correct key, in this case *privK P* if the message was encrypted using *pubK P*, or *pubK P* if the message was encrypted with *privK P*. As before, garbage in produces garbage out.

The properties of *deciphP* are shown in Figures C.7 and C.8 by theorems *deciphP_clauses* and *deciphP_one_one*. Together, they show the circumstances under which the original plaintext is decrypted, when nothing useful is decrypted, and the conditions that ensure that the expected keys and plaintext messages were in fact, used.

[deciphP_def]

```
⊢ (deciphP key (Ea (privK P) (SOME x)) =
   if key = pubK P then SOME x else NONE) ∧
  (deciphP key (Ea (pubK P) (SOME x)) =
   if key = privK P then SOME x else NONE) ∧
  (deciphP k1 (Ea k2 NONE) = NONE)
```

[deciphP_clauses]

```
⊢ (∀P text.
     (deciphP (pubK P) (Ea (privK P) (SOME text)) =
      SOME text) ∧
     (deciphP (privK P) (Ea (pubK P) (SOME text)) =
      SOME text)) ∧
  (∀k P text.
     (deciphP k (Ea (privK P) (SOME text)) = SOME text) ⟺
     (k = pubK P)) ∧
  (∀k P text.
     (deciphP k (Ea (pubK P) (SOME text)) = SOME text) ⟺
     (k = privK P)) ∧
  (∀x k2 k1 P2 P1.
     (deciphP (pubK P1) (Ea (pubK P2) (SOME x)) = NONE) ∧
     (deciphP k1 (Ea k2 NONE) = NONE)) ∧
  ∀x P2 P1. deciphP (privK P1) (Ea (privK P2) (SOME x)) = NONE
```

Figure C.7: Definitions and Properties of Asymmetric Decryption

C.4.1 Digital Signatures

Digitally signed messages are often a combination of cryptographic hashes of messages encrypted using the *private key* of the sender. This is shown in Figure C.9, which depicts signature generation as the following sequence of operations:

1. A message is hashed, then
2. the message hash is encrypted using the private key of the sender.

The intuition behind signatures is this: (1) the cryptographic hash is a unique pointer to the message (and potentially much smaller than the message), and (2) encrypting using the sender's private key (which is reversible by the sender's public key) is a unique pointer to the sender.

Figure C.10 shows how decrypted messages are checked for integrity using digital signatures. The top-most sequence from left to right shows how the decrypted hash value is retrieved from the received digital signature. The digital signature is decrypted using the sender's public key to retrieve the hash or digest of the original message. The retrieved hash is compared to the hash of the decrypted message. If the two hash values are the same, then the received message is judged to have arrived unchanged from the original.

Figure C.11 shows the function definitions in HOL of *sign* and *signVerify*. *sign* takes as inputs a *pKey* and a digest and returns an asymmetrically encrypted digest using the asymmetric *pKey*. *signVerify* takes as input a *pKey*, digital signature, and a received message and compares the decrypted hash in the signature with the hash of the received message. The properties of *signVerify* and *sign* are in theorems *signVerifyOK* and *signVerify_one_one*.

- *signVerify* is always true for signatures generated as shown in Figure C.9.
- *signVerify* and *sign* combine to have the desired properties that the plaintext must match and the corresponding keys must match.

```
[deciphP_one_one]

⊢ (∀ P1 P2 text1 text2.
     (deciphP (pubK P1) (Ea (privK P2) (SOME text2)) =
      SOME text1) ⟺ (P1 = P2) ∧ (text1 = text2)) ∧
   (∀ P1 P2 text1 text2.
     (deciphP (privK P1) (Ea (pubK P2) (SOME text2)) =
      SOME text1) ⟺ (P1 = P2) ∧ (text1 = text2)) ∧
   (∀ p c P msg.
     (deciphP (pubK P) (Ea p c) = SOME msg) ⟺
     (p = privK P) ∧ (c = SOME msg)) ∧
   (∀ enMsg P msg.
     (deciphP (pubK P) enMsg = SOME msg) ⟺
     (enMsg = Ea (privK P) (SOME msg))) ∧
   (∀ p c P msg.
     (deciphP (privK P) (Ea p c) = SOME msg) ⟺
     (p = pubK P) ∧ (c = SOME msg)) ∧
   ∀ enMsg P msg.
     (deciphP (privK P) enMsg = SOME msg) ⟺
     (enMsg = Ea (pubK P) (SOME msg))
```

Figure C.8: One-to-One Properties of Asymmetric Decryption

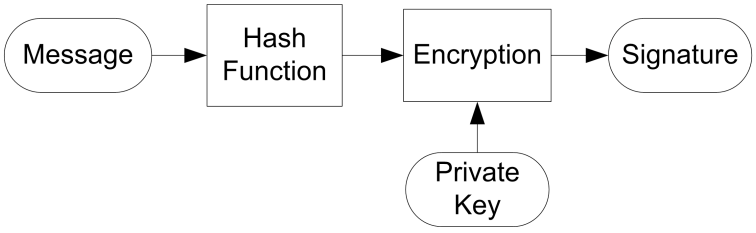


Figure C.9: Digital Signature Generation

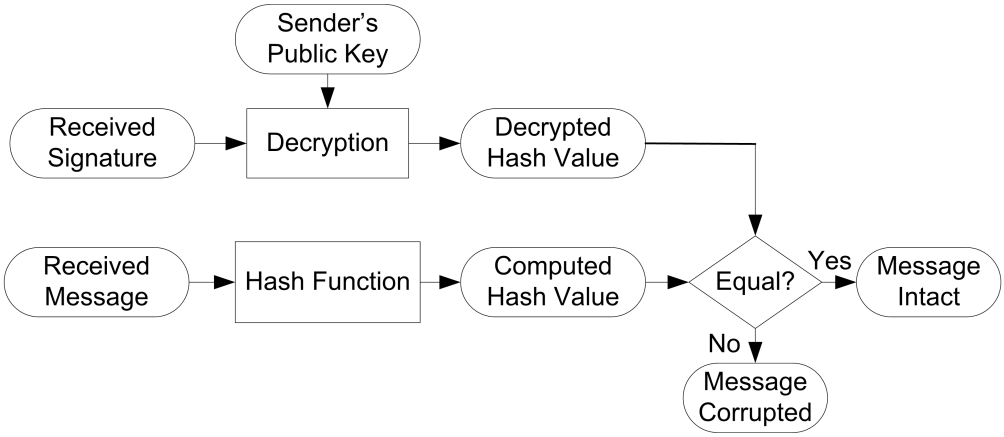


Figure C.10: Digital Signature Verification

[sign_def]

$\vdash \forall\, pubKey\ \ dgst.\ \texttt{sign}\ pubKey\ \ dgst\ \texttt{=}\ \texttt{Ea}\ pubKey\ \ \texttt{(SOME}\ dgst\texttt{)}$

[signVerify_def]

$\vdash \forall\, pubKey\ \ signature\ \ msgContents.$
$\quad \texttt{signVerify}\ pubKey\ \ signature\ \ msgContents\ \iff$
$\quad \texttt{(SOME (hash}\ msgContents\texttt{) = deciphP}\ pubKey\ \ signature\texttt{)}$

[signVerifyOK]

$\vdash \forall\, P\ \ msg.$
$\quad \texttt{signVerify (pubK}\ P\texttt{) (sign (privK}\ P\texttt{) (hash (SOME}\ msg\texttt{)))}$
$\quad\quad \texttt{(SOME}\ msg\texttt{)}$

[signVerify_one_one]

$\vdash (\forall\, P\ \ m_1\ \ m_2.$
$\quad \texttt{signVerify (pubK}\ P\texttt{) (Ea (privK}\ P\texttt{) (SOME (hash (SOME}\ m_1\texttt{))))}$
$\quad\quad \texttt{(SOME}\ m_2\texttt{)}\ \iff\ \texttt{(}m_1\ \texttt{=}\ m_2\texttt{))}\ \wedge$
$\quad (\forall\, signature\ \ P\ \ text.$
$\quad\quad \texttt{signVerify (pubK}\ P\texttt{)}\ signature\ \texttt{(SOME}\ text\texttt{)}\ \iff$
$\quad\quad \texttt{(}signature\ \texttt{=sign (privK}\ P\texttt{) (hash (SOME}\ text\texttt{))))}\ \wedge$
$\quad \forall\, text_2\ \ text_1\ \ P_2\ \ P_1.$
$\quad\quad \texttt{signVerify (pubK}\ P_1\texttt{) (sign (privK}\ P_2\texttt{) (hash (SOME}\ text_2\texttt{)))}$
$\quad\quad\quad \texttt{(SOME}\ text_1\texttt{)}\ \iff\ \texttt{(}P_1\ \texttt{=}\ P_2\texttt{)}\ \wedge\ \texttt{(}text_1\ \texttt{=}\ text_2\texttt{)}$

Figure C.11: Digital Signature Generation, Verification, and Their Properties

Appendix D

cipher Theory

D.1 cipher Theory

Built: 23 March 2018
Parent Theories: indexedLists, patternMatches

D.1.1 Datatypes

asymMsg = Ea ('princ pKey) ('message option)

digest = hash ('message option)

pKey = pubK 'princ | privK 'princ

symKey = sym num

symMsg = Es symKey ('message option)

D.1.2 Definitions

[sign_def]
$\vdash \forall\, pubKey\ dgst.$ sign $pubKey\ dgst$ = Ea $pubKey$ (SOME $dgst$)

[signVerify_def]
$\vdash \forall\, pubKey\ signature\ msgContents.$
signVerify $pubKey\ signature\ msgContents \iff$
(SOME (hash $msgContents$) = deciphP $pubKey\ signature$)

D.1.3 Theorems

[asymMsg_one_one]
$\vdash \forall\, a_0\ a_1\ a_0'\ a_1'.$
(Ea $a_0\ a_1$ = Ea $a_0'\ a_1'$) $\iff$ (a_0 = a_0') $\wedge$ (a_1 = a_1')

[deciphP_clauses]
$\vdash$ ($\forall\, P\ text.$
(deciphP (pubK P) (Ea (privK P) (SOME $text$)) =
SOME $text$) $\wedge$
(deciphP (privK P) (Ea (pubK P) (SOME $text$)) =
SOME $text$)) $\wedge$
($\forall\, k\ P\ text.$
(deciphP k (Ea (privK P) (SOME $text$)) = SOME $text$) $\iff$
(k = pubK P)) $\wedge$
($\forall\, k\ P\ text.$
(deciphP k (Ea (pubK P) (SOME $text$)) = SOME $text$) $\iff$
(k = privK P)) $\wedge$
($\forall\, x\ k_2\ k_1\ P_2\ P_1.$
(deciphP (pubK P_1) (Ea (pubK P_2) (SOME x)) = NONE) $\wedge$
(deciphP k_1 (Ea k_2 NONE) = NONE)) $\wedge$
$\forall\, x\ P_2\ P_1.$ deciphP (privK P_1) (Ea (privK P_2) (SOME x)) = NONE

[deciphP_def]

$$\begin{array}{l}\vdash \texttt{(deciphP } key \texttt{ (Ea (privK } P\texttt{) (SOME } x\texttt{)) =} \\ \quad \textbf{if } key = \texttt{pubK } P \textbf{ then } \texttt{SOME } x \textbf{ else } \texttt{NONE)} \wedge \\ \texttt{(deciphP } key \texttt{ (Ea (pubK } P\texttt{) (SOME } x\texttt{)) =} \\ \quad \textbf{if } key = \texttt{privK } P \textbf{ then } \texttt{SOME } x \textbf{ else } \texttt{NONE)} \wedge \\ \texttt{(deciphP } k_1 \texttt{ (Ea } k_2 \texttt{ NONE) = NONE)}\end{array}$$

[deciphP_ind]

$$\begin{array}{l}\vdash \forall P'. \\ \quad (\forall key\ P\ x.\ P'\ key\ \texttt{(Ea (privK } P\texttt{) (SOME } x\texttt{)))} \wedge \\ \quad (\forall key\ P\ x.\ P'\ key\ \texttt{(Ea (pubK } P\texttt{) (SOME } x\texttt{)))} \wedge \\ \quad (\forall k_1\ k_2.\ P'\ k_1\ \texttt{(Ea } k_2 \texttt{ NONE))} \Rightarrow \\ \quad \forall v\ v_1.\ P'\ v\ v_1\end{array}$$

[deciphP_one_one]

$$\begin{array}{l}\vdash (\forall P_1\ P_2\ text_1\ text_2. \\ \quad \texttt{(deciphP (pubK } P_1\texttt{) (Ea (privK } P_2\texttt{) (SOME } text_2\texttt{)) =} \\ \quad \texttt{SOME } text_1\texttt{)} \iff (P_1 = P_2) \wedge (text_1 = text_2)) \wedge \\ (\forall P_1\ P_2\ text_1\ text_2. \\ \quad \texttt{(deciphP (privK } P_1\texttt{) (Ea (pubK } P_2\texttt{) (SOME } text_2\texttt{)) =} \\ \quad \texttt{SOME } text_1\texttt{)} \iff (P_1 = P_2) \wedge (text_1 = text_2)) \wedge \\ (\forall p\ c\ P\ msg. \\ \quad \texttt{(deciphP (pubK } P\texttt{) (Ea } p\ c\texttt{) = SOME } msg\texttt{)} \iff \\ \quad (p = \texttt{privK } P) \wedge (c = \texttt{SOME } msg)) \wedge \\ (\forall enMsg\ P\ msg. \\ \quad \texttt{(deciphP (pubK } P\texttt{) } enMsg \texttt{ = SOME } msg\texttt{)} \iff \\ \quad (enMsg = \texttt{Ea (privK } P\texttt{) (SOME } msg\texttt{))) } \wedge \\ (\forall p\ c\ P\ msg. \\ \quad \texttt{(deciphP (privK } P\texttt{) (Ea } p\ c\texttt{) = SOME } msg\texttt{)} \iff \\ \quad (p = \texttt{pubK } P) \wedge (c = \texttt{SOME } msg)) \wedge \\ \forall enMsg\ P\ msg. \\ \quad \texttt{(deciphP (privK } P\texttt{) } enMsg \texttt{ = SOME } msg\texttt{)} \iff \\ \quad (enMsg = \texttt{Ea (pubK } P\texttt{) (SOME } msg\texttt{))}\end{array}$$

[deciphS_clauses]

$$\begin{array}{l}\vdash (\forall k\ text.\ \texttt{deciphS } k\ \texttt{(Es } k\ \texttt{(SOME } text\texttt{)) = SOME } text) \wedge \\ (\forall k_1\ k_2\ text. \\ \quad \texttt{(deciphS } k_1\ \texttt{(Es } k_2\ \texttt{(SOME } text\texttt{)) = SOME } text\texttt{)} \iff \\ \quad (k_1 = k_2)) \wedge \\ (\forall k_1\ k_2\ text. \\ \quad \texttt{(deciphS } k_1\ \texttt{(Es } k_2\ \texttt{(SOME } text\texttt{)) = NONE)} \iff k_1 \neq k_2) \wedge \\ \forall k_1\ k_2.\ \texttt{deciphS } k_1\ \texttt{(Es } k_2\ \texttt{NONE) = NONE}\end{array}$$

[deciphS_def]

$$\begin{array}{l}\vdash \texttt{(deciphS } k_1\ \texttt{(Es } k_2\ \texttt{(SOME } x\texttt{)) =} \\ \quad \textbf{if } k_1 = k_2 \textbf{ then } \texttt{SOME } x \textbf{ else } \texttt{NONE)} \wedge \\ \texttt{(deciphS } k_1\ \texttt{(Es } k_2\ \texttt{NONE) = NONE)}\end{array}$$

[deciphS_ind]

$$\begin{array}{l}\vdash \forall P. \\ \quad (\forall k_1\ k_2\ x.\ P\ k_1\ \texttt{(Es } k_2\ \texttt{(SOME } x\texttt{)))} \wedge \\ \quad (\forall k_1\ k_2.\ P\ k_1\ \texttt{(Es } k_2\ \texttt{NONE))} \Rightarrow \\ \quad \forall v\ v_1.\ P\ v\ v_1\end{array}$$

[deciphS_one_one]

$\vdash$ ($\forall\, k_1\ k_2\ text_1\ text_2$.
(deciphS k_1 (Es k_2 (SOME $text_2$)) = SOME $text_1$) $\iff$
(k_1 = k_2) $\wedge$ ($text_1$ = $text_2$)) $\wedge$
$\forall\, enMsg\ text\ key$.
(deciphS key $enMsg$ = SOME $text$) $\iff$
($enMsg$ = Es key (SOME $text$))

[digest_one_one]

$\vdash \forall\, a\ a'$. (hash a = hash a') $\iff$ (a = a')

[option_distinct]

$\vdash \forall\, x$. NONE $\neq$ SOME x

[option_one_one]

$\vdash \forall\, x\ y$. (SOME x = SOME y) $\iff$ (x = y)

[pKey_distinct_clauses]

$\vdash$ ($\forall\, a'\ a$. pubK a $\neq$ privK a') $\wedge$ $\forall\, a'\ a$. privK a' $\neq$ pubK a

[pKey_one_one]

$\vdash$ ($\forall\, a\ a'$. (pubK a = pubK a') $\iff$ (a = a')) $\wedge$
$\forall\, a\ a'$. (privK a = privK a') $\iff$ (a = a')

[sign_one_one]

$\vdash \forall\, pubKey_1\ pubKey_2\ m_1\ m_2$.
(sign $pubKey_1$ (hash m_1) = sign $pubKey_2$ (hash m_2)) $\iff$
($pubKey_1$ = $pubKey_2$) $\wedge$ (m_1 = m_2)

[signVerify_one_one]

$\vdash$ ($\forall\, P\ m_1\ m_2$.
signVerify (pubK P) (Ea (privK P) (SOME (hash (SOME m_1))))
(SOME m_2) $\iff$ (m_1 = m_2)) $\wedge$
($\forall\, signature\ P\ text$.
signVerify (pubK P) $signature$ (SOME $text$) $\iff$
($signature$ = sign (privK P) (hash (SOME $text$)))) $\wedge$
$\forall\, text_2\ text_1\ P_2\ P_1$.
signVerify (pubK P_1) (sign (privK P_2) (hash (SOME $text_2$)))
(SOME $text_1$) $\iff$ (P_1 = P_2) $\wedge$ ($text_1$ = $text_2$)

[signVerifyOK]

$\vdash \forall\, P\ msg$.
signVerify (pubK P) (sign (privK P) (hash (SOME msg)))
(SOME msg)

[symKey_one_one]

$\vdash \forall\, a\ a'$. (sym a = sym a') $\iff$ (a = a')

[symMsg_one_one]

$\vdash \forall\, a_0\ a_1\ a_0'\ a_1'$.
(Es a_0 a_1 = Es a_0' a_1') $\iff$ (a_0 = a_0') $\wedge$ (a_1 = a_1')

Bibliography

[1] IEEE Guide for Information Technology–System Definition–Concept of Operations (ConOps) Document, 19 March 1998. IEEE Computer Society, IEEE Std 1362-1998.

[2] JP 5-0, Joint Operation Planning, 11 August 2011. US Dept of Defense.

[3] Abdulkhaleq, A. *A System-Theoretic Safety Engineering Approach for Software-Intensive Systems.* PhD thesis, University of Stuttgart, 2017.

[4] Agron, J. Domain-specific language for hw/sw co-design for fpgas. In *Proceedings of the IFIP Working Conference on Domain Specific Languages (DSL WC)* (July 2009), vol. 5658 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 262–284.

[5] Bell, D. E., and La Padula, L. J. Secure computer system: Unified exposition and Multics interpretation. Tech. Rep. MTR-2997 Rev. 1, MITRE Corporation, Bedford, MA, March 1975.

[6] Bensoussan, A., Clingen, C. T., and Daley, R. C. The Multics virtual memory: Concepts and design. *Communications of the ACM 15*, 5 (May 1972), 308–318.

[7] Checkland, P. *Systems Thinking, Systems Practice.* John Wiley & Sons, New York, 1981.

[8] Chin, S.-K. Teaching undergraduates certified security by design. In *19^{t}h Colloquium for Information Systems Security Education* (Las Vegas, NV, June 2015).

[9] Chin, S.-K. *Cyber-Assurance for the Internet of Things.* IEEE Press and Wiley, 2017, ch. Certified Security by Design for the Internet of Things.

[10] Chin, S.-K., Devendorf, E., Muccio, S., Older, S., and Royer, J. Formal verification for mission assurance in cyberspace: Education, tools, and results. In *Proceedings 16th Colloquium for Information Systems Security Education* (Lake Buena Vista, FL, June 2012), pp. 75–82.

[11] Chin, S.-K., Muccio, S., Older, S., and Vestal, T. Policy-based design and verification for mission assurance. In *Computer Network Security, Fifth International Conference on Mathematical Methods, Models and Architectures for Computer Network Security* (St. Petersburg, Russia, September 2010).

[12] Chin, S.-K., and Older, S. A rigorous approach to teaching access control. In *Annual Conference on Education in Information Security (ACIS '06)* (Ames, Iowa, September 2006).

[13] Chin, S.-K., and Older, S. *Access Control, Security, and Trust: A Logical Approach.* CRC Press, Boca Raton, FL, 2010.

[14] Chin, S.-K., and Older, S. *Certified Security by Design Using Higher Order Logic.* CRC Press, Forthcoming.

[15] Conway, L. Reminiscences of the vlsi revolution: How a series of failures triggered a paradigm shift in digital design. *IEEE Solid State Circuits Magazine 4*, 4 (2012), 8–31.

[16] Cornea-Hasegan, M. Proving the ieee correctness of iterative floating-point square root, divide, and remainder algorithms. *Intel Technology Journal Q2* (1998), 1–11.

[17] Dill, D. L., and Rushby, J. Acceptance of formal methods: Lessons from hardware design. *IEEE Computer 29*, 4 (April 1996), 16–30.

[18] Gordon, M., and Melham, T. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[19] Harrison, J. Floating-point verification. In *FM 2005: Formal Methods* (2005), T. A. Fitzgerald J., Hayes I.J., Ed., vol. 3582 of *Lecture Notes in Computer Science*, Springer, Berlin, Heidelberg.

[20] Headquarters, Department of the Army. *Ranger Handbook*, tc 3-21.76 ed., April 2017.

[21] Humphrey, W. S. Characterizing the software process: A maturity framework. *IEEE Software* (March 1988), 73–79.

[22] Hüttel, H. *Transitions and Trees: An Introduction to Structural Operational Semantics*. Cambridge University Press, New York, 2010.

[23] Jabbour, K., and Older, S. The advanced course in engineering on cyber security: A learning community for developing cyber-security leaders. In *Proceedings of the Sixth Workshop on Education in Computer Security* (July 2004).

[24] Kc, G. S., Keromytis, A. D., and Previlakis, V. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security* (2003), ACM, pp. 272–280.

[25] Leveson, N. G., and Thomas, J. P. *STPA-Handbook*. MIT, March 2018. Available at http://psas.scripts.mit.edu/home/get_file.php?name=STPA_handbook.pdf.

[26] Older, S., and Chin, S.-K. Engineering assurance at the undergraduate level. *IEEE Security and Privacy 10*, 6 (November/December 2012).

[27] Organick, E. *The Multics System: An Examination of Its Structure*. MIT Press, Cambridge, MA, 1972.

[28] Papadogiannakis, A., Loutsis, L., Papaefstathiou, V., and Ioannidis, S. Asist: architectural support for instruction set randomization. In *Proceedings of the 13th ACM Conference on Computer and Communications Security* (November 4–8 2013), ACM, pp. 981–992.

[29] Popek, G. J., and Goldberg, R. P. Formal requirements for virtualizable third generation architectures. *Communications of the ACM 17*, 7 (July 1974), 412–421.

[30] Ross, R., McEvilley, M., and Oren, J. C. Systems security engineering: Considerations for a multidisciplinary approach in the engineering of trustworthy secure systems. Tech. Rep. SP 800-160, National Institute of Standards and Technology, May 2016. Second Public Draft.

[31] Saltzer, J., and Schroeder, M. The Protection of Information in Computer Systems. *Proceedings IEEE* (1975).

[32] Schroeder, M. D., and Saltzer, J. H. A hardware architecture for implementing protection rings. *Communications of the ACM 15*, 3 (March 1972), 157–170.

[33] Young, W., and Leveson, N. Systems thinking for safety and security. In *ACSAC 2013* (2013).