

_pb-dfd-level

Certified Security by Design Applied to the Patrol Based A Demonstration of CSBD Applicability to Military Operations

Professor Shiu-Kai Chin, Lori Pickering, Jesse Nathaniel Hall, YiHong Guo

August 2017

Contents

List of Figures	4
Preface	5
Acknowledgments	7
1 Summary	11
2 Introduction	13
3 Methods, Assumptions, and Procedures	15
3.1 Methods	15
3.1.1 The Implementation Approach	16
3.1.2 Hierarchy of State Machines	16
3.1.3 The Secure State Machine	17
3.1.4 Authentication	17
3.1.5 Authorization	18
3.1.6 State Interpretation	19
3.1.7 Implementation of the Secure State Machine Model	19
3.1.8 Parameterization of the ssm11 secure state machine	21
3.1.9 Future Work – Some Notes	22
4 Results and Discussions	27
4.1 Overall Results and Discussion	27
4.2 Design Results and Discussion	27
4.3 Implementation Results and Discussion	27
4.3.1 ssm11Script.sml, satListScript.sml, and ssminfRules.sml	27
4.3.2 PBTypeScript.sml and ssmPBScript.sml	32
5 Conclusions	39
5.1 Overall Conclusions	39
5.2 Future Work	39
5.2.1 Design Phase	39
5.2.2 Implementation Phase	39
6 Recommendations	41
6.1 Design Recommendations	41
7 References	43
A State Machine Overview	47
B PB UML Class Diagram	49
C PLAN_PB DFD Level 0	51

D	Next State, Next Output Table	53
E	Source code for ssmPBScript.sml	55
F	Source code for PBTypeScript.sml	61
G	Symbols, Abbreviations, and Acronyms	63

List of Figures

2.1	HazardTable	13
2.2	Organizational Chart	14
3.1	PB Phase Overview	15
3.2	PB Top Level States Overview	22
3.3	PB Top Level Alternate States Overview	23
3.4	PB Sub States Overview	24
3.5	TLP Steps	24
3.6	TLP PLAN_PB unordered sub states	25

Preface

Genesis

Purpose

The purpose of this document is to describe the project at hand. The purpose of the project is to demonstrate the applicability of the certified security by design approach using access-control logic (ACL) and the higher order logic (HOL) theorem prover to a project of the size and type of the patrol base. [Add stuff about how this could be used in stuff that Jesse spoke of. Or, if Jesse is not cooperating, ask professor Chin.]

Limitations

This project has no limitations other than the time needed to represent the design phase of the project in the access-control logic (ACL) and implement that in the higher order logic (HOL) theorem prover.

Scope

This document describes the how the patrol base was represented as state machines and then implemented in the access-control logic (ACL) and higher order logic (HOL). Professor Shiu-Kai Chin is the Principal Investigator and is responsible for the project as a whole. Jesse Nathaniel Hall is the subject matter aspect. Lori Pickering is the access-control logic (ACL) and higher order logic (HOL) expert. YiHong Guo assists with organizing the documentation. This documentation is organized similarly to how the project unfolded. Jesse and I came together to figure out how to represent the patrol base. We decided on a state machine approach to organizing the information in the patrol base. After this initial phase, we each worked separately, Jesse on translating the patrol base into state machine and Lori into implementing this into ACL and HOL, coming together occasionally to clarify areas as necessary.

The METHODS, ASSUMPTIONS, AND PROCEDURES Section is divided into three sections: 3.1 Overall Approach; 3.2 Design; and 3.3 Implementation. Section 3.1 focuses primarily on the initiation of the project. Sections 3.2 and 3.3 discuss the design and implementation phase, respectively, from the person who knows that section the best.

The RESULTS AND DISCUSSIONS section is separated into 3 parts, similarly to section 3. This section shows the results of the METHODS, ASSUMPTIONS, AND PROCEDURES. Section 4.2 includes [description based on what Jesse did]. It includes a step-by-step walk through one of the secure state machine implementations in HOL in section 4.3.

The Conclusions section contains recommendations for future work in both the design phase (section 5.2.1) and the implementation phase (section 5.2.2). 5.1 contains discussions about the project in general.

Acknowledgments

This project involved multiple people at multiple levels. The Principal Investigator was Professor Shiu-Kai Chin of Syracuse University's Department of Engineering and Computer Science. Lori Pickering, Graduate Student at Syracuse University's Department of Engineering and Computer Science, primarily worked on the access-control logic (ACL) and higher order logic (HOL) implementation of the patrol base. Together with Jesse Nathaniel Hall, they constructed the idea of translating the patrol base operations into state machines. Jesse, also a student at Syracuse University in the iSchool and a [rank] in the U.S. Army, was our subject matter expert. Once Jesse and Lori decided on the state machine format, Jesse translated the patrol base operations into this format. YiHong Guo came to the project near the end. Nevertheless, he contributed significantly to the project. YiHong worked on organizing the documentation in LaTeX.

Chapter 1

Summary

Chapter 2

Introduction

Platoon patrol base (PB) operations are an essential part of many military operations. They provide Soldiers with a means to avoid enemy contact, rest, resupply, and conduct Squad level missions (Ranger Handbook 2011, 132). Although some might believe that a PB begins from an objective rally point (ORP), a PB must be planned for. Our PB software includes a state machine model for tasks that include planning, movement, as well as conducting both an ORP and PB.

Any automation of the patrol base operations (for example Jesse? Professor Chin?) would require that the design of such system be certified as secure from the start. This approach should be applied to all systems. Yet, our aim in this project was to show that the certified security by design approach could be applied specifically to the patrol base operations because it is described in a manner that is similar to many other military operations. These operations will, no doubt, be automated in the future. Thus, this project should be viewed as a primer on how to tackle such an automation in a secure manner, from the start.

The certified security by design approach involves taking the design of the automated system and using access-control logic (ACL) to prove that any action that takes place in the automated system is correctly executed and has the appropriate authorizations and authentications of those authorities. Computer-aided reasoning is used to verify that the system is secure. For this project, we used the higher order logic (HOL) theorem prover. HOL is a trusted and reliable system. It is commonly said by the HOL community that HOL is never wrong.

The access-control logic (ACL) was developed by Professor Shiu-Kai Chin and Professor Susan Older of Syracuse University's Department of Engineering and Computer Science. Specifics can be found in their text book titled Access Control, Security, and Trust. The definitions and theorems from the text were implemented in HOL by Professor Shiu-Kai Chin and Lockwood Morris. For reference a report of the datatypes, definitions, and theorems that were implemented in HOL are provided in Appendices ?

Our method for understanding what represented mission failure and mission success was guided by Functional Mission Analysis (Young,?). We identified that our goal was to develop A system to provide mission assurance for execution of patrol base operations by means of logical proof in order to mitigate enemy/civilian contact and mission failure. Unacceptable losses would normally be at the Higher Headquarter (HHQs) Commanders discretion, but for the sake of our project they were the (L1) loss of civilian life, (L2) 20% of our Platoon being rendered Non-Mission Capable (NMC), and (L3) mission failure. Hazards that could lead to unacceptable losses included (H1) contact with civilians, (H2) contact with the enemy, and (H2) loss of adequate mission essential supplies. The table below illustrated our causal links between hazards and losses.

Hazards \ Losses	L1: Loss of Civilian Life	L2: 20% of Platoon NMC	L3: Mission Failure
H1: Civilian Contact	X		X
H2: Enemy Contact		X	X
H3: Loss of Mission Essential Supplies		X	X

Figure 2.1: HazardTable

Our functional control structure is the chain of command illustrated below outlines what individual, in what unit level would be responsible for the actions of the Soldiers below them.

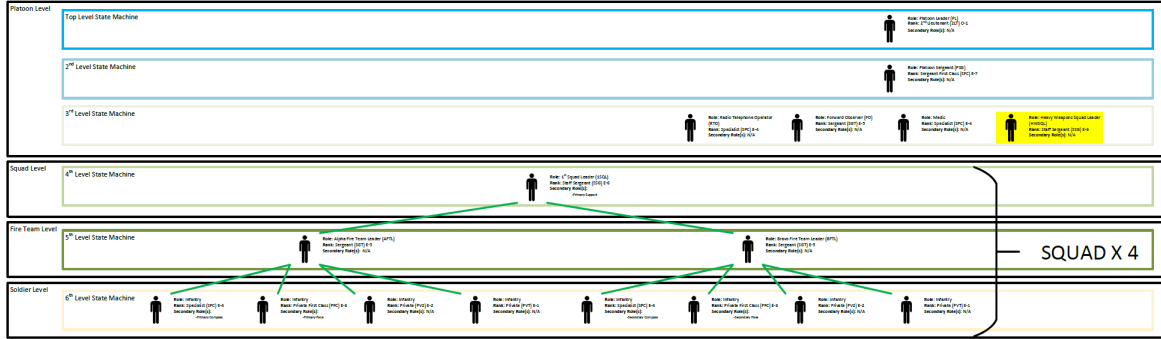




Figure 2.2: Organizational Chart

The organizational chart showed the hierarchy of the Platoon and whom would be in control of each element and responsible for mitigating risks caused by hazards. Our application design recognized each leaders ability to control certain groups/units of Soldiers. With this in mind, leaders within the Platoon would have to (C1) avoid populated areas, (C2) avoid trafficked routes to and from those areas, (C3) maintain supplies within the HHQ standard operating procedure (SOP). These constraints were built into our application with states that performed reconnaissance, uniform and packing list checks, maintenance and cross loading of supplies.

3.1 Methods

Patrol Base Phases (REVISED)



- Chronological Order
 - Planning
 - Move to Objective Rally Point (ORP)
 - ORP
 - Move to Patrol Base (PB)
 - PB
 - PB Complete

The diagram illustrates the Patrol Base Phases (REVISED) in a chronological sequence. It begins with 'Controlled Territory' (blue box) leading to '1. PLT Planning' (green box). This is followed by '2. PLT Move to ORP' (green box), which leads to 'Objective Rally Point (ORP)' (blue oval). From the ORP, the sequence continues to '3. PLT ORP' (green box), then '4. PLT Move to PB' (green box), and finally 'Listening Post/Observation Post (LPOP)' (blue oval). The LPOP leads to '5. PLT PB' (green box), which then leads to '6. PLT PB Complete' (green box). The final step is 'Continue Mission (CM)' (yellow arrow). The diagram also includes a 'Heavy Weapons Team' (blue oval) positioned between the ORP and the LPOP, with arrows indicating movement from the ORP to the team and from the team to the LPOP. A '2nd Squad' (yellow arrow) is shown moving from the ORP to the Heavy Weapons Team. A 'Recon/Emplace' (yellow arrow) is shown moving from the LPOP to the Heavy Weapons Team. A 'Heavy Weapons Team' (blue oval) is also shown at the top right, with an arrow labeled '2nd Squad' pointing to it from the ORP. A 'Heavy Weapons Team' (blue oval) is also shown at the bottom right, with an arrow labeled 'Recon/Emplace' pointing to it from the LPOP. A 'Heavy Weapons Team' (blue oval) is also shown at the top left, with an arrow labeled '2nd Squad' pointing to it from the ORP. A 'Heavy Weapons Team' (blue oval) is also shown at the bottom left, with an arrow labeled 'Recon/Emplace' pointing to it from the LPOP. A 'Heavy Weapons Team' (blue oval) is also shown at the top right, with an arrow labeled '2nd Squad' pointing to it from the ORP. A 'Heavy Weapons Team' (blue oval) is also shown at the bottom right, with an arrow labeled 'Recon/Emplace' pointing to it from the LPOP. A 'Heavy Weapons Team' (blue oval) is also shown at the top left, with an arrow labeled '2nd Squad' pointing to it from the ORP. A 'Heavy Weapons Team' (blue oval) is also shown at the bottom left, with an arrow labeled 'Recon/Emplace' pointing to it from the LPOP.

NOTE: Heavy Weapons Team placement is the experience of Jesse Hall.
CHANGE: Steps now contain "PB Recon," "PB Security," "PB Occupy," due to decisions at meeting 3 on 5JUN17. Reference Pictures.

After the PB was broken into phases, we began creating states that mimicked the phases outlined in the Powerpoint@document (figure 3.1). Microsoft®Visio was used to illustrate the states, their order, and the commands that initiated movement from one state to another (figure 3.2). The decision to adopt a state machine as the method for rendering PB operations came from previous projects Dr. Shiu-Kai Chin had allowed our team to review. Projects programmed in Poly Meta Language (Poly/ML) and compiled with High Order Logic (HOL) seemed to lend themselves well to the state machine structure.

3.1.1 The Implementation Approach

The goal was to represent the patrol base in a way that would be amiable to applying access-control logic (ACL) principles. The motive for this project was, in fact, to demonstrate the applicability of these principles to the mission assurance objective. Representing the patrol base in terms of state machines was easily implemented with computer-aided reasoning, in this case the HOL higher order logic theorem prover. The basic idea was to establish states, commands to transition from one state to another, and people who were authorized to make these commands. To do this, it was first necessary to translate the patrol base into discrete states. This was where the subject matter expertise of Jesse was crucial. The details were described in the section add section title here.

3.1.2 Hierarchy of State Machines

In this project, we decided on a hierarchy of state machines with a top-level state machine, sub-level state machines, and sub-sub-level state machines, etc. as the foundational approach. That is, we began with a top-level state machine consisting of 6 states. Each state (save for the terminal state) was yet another state machine at a sub-level. We called this a sub-level state machine. There was one sub-level state machine for each state in the top-level state machine. This produced a pyramid-like, top down structure. Adding yet another level, the sub-sub-level, produced a state machine for each state machine in the higher-up sub-level state machine. These were called the sub-sub-level state machines. Beyond this level, we considered alternative approaches. Although we discussed and worked out some strategies here, the project was simply too large for one summer. These ideas are discussed in the Future Work section.

The reason for this pyramid-like approach was multi-fold.

- First, it was logical to look at the patrol base as a whole and design a basic, top-level state machine that represented the most basic states. Going into too much detail at first would be distracting and thus error-prone.
- Second, this approach facilitated a test of the state machine approach before going into too much depth. It was easy to see at the top-level what would work and what would not work.
- Third, it allowed for separation of tasks. Jesses expertise was in his knowledge of the patrol base and his ability to work with Visio and other similar technologies. Loris expertise was in her knowledge of the access-control logic (ACL) principles and her ability to work with the higher order logic (HOL) theorem prover. Once a top-level state machine was established, Jesse continued work on the delineation of the patrol base into state machines while Lori began work on implementing the state machine(s) using the ACL principals and HOL.
- Fourth, the modularization facilitated the concept of separation of implementation. For example, if someone were to take the top-level state machine and delegate the sub-level state machines to others, then they could all be integrated together at a later time. The same could be said at the sub-level, and so on. The idea of integration is discussed further in the Future Works section.
- Fifth, approaching the project one level at a time facilitated a step-by-step approach to solving each new level of complexity while capitalizing on previous successes. Because of the nature of HOL, once we solved one problem in the implementation of the patrol base, we could rely on that solution. The next problem could then be isolated and attacked without back-tracking for the most part. We could add complexity as we approached it, in analogy to the complexity of detail in the patrol base itself.
- Sixth, time constraints on the project were tight. As we delved into the sub-level and sub-sub-level state machines, the level of detail that needed to be implemented in the ACL and HOL became apparent. The outlooks for the project were exciting. And, at the same time, we realized that a complete implementation of the patrol base in HOL would require more than one summer. Thus, the level-by-level approach facilitated a view of what could be done based on what had been done, demonstrating the practicality of the application of ACL and HOL to a patrol base-like problem. This was our goal.

3.1.3 The Secure State Machine

Again, the reason for the state machine approach was that it was the simplest way to represent a large project using ACL and implement it using HOL. If the patrol base was represented as a state machine, then the orders/commands could be represented as inputs into the states. This was what we did. We determined at each level, what should be considered a state and what order/command should be given to transition from that state to the next. For a basic state machine, this was all that was needed to represent the given level of complexity. However, in general, we were concerned with authentication and authorization. Who was giving the command and did that person have the authority to give that command? Thus, we needed the appropriate checks on identity (authentication) and authority to make the transition from one state to another. We needed not just a *state machine* but a *secure state machine*.

For a secure state machine, we hiked up the complexity of our state machine model. We now only allowed a transition from one state to another if the order was given by someone (or something) that was authenticated and authorized. We called that someone (or something) in the access-control logic (ACL) a principal. For example, we authenticate (via visual recognition) a soldier as the platoon leader. We establish *a priori* that the platoon leader had the authority to command a change from one state to another. Given those conditions, we wanted our state machine to make that change. More specifically, assume we were in the planning phase of the patrol base. We called this the PLAN_PB state. (To be consistent we denoted all states by capital letters.) In this state, the platoon leader said cross the line of discrimination. We abbreviated this command as crossLD. (We used primarily lower-case letters for commands.) Furthermore, we all knew that the platoon leader was authorized to give the command to cross the line of discrimination and begin the next phase of the patrol base. We called this next phase the MOVE_TO_ORP state. Under these circumstances, the access-control logic dictated that we transition from the PLAN_PB state to the MOVE_TO_ORP state. We did this for each state, using the ACL and HOL to verify the proper authentication and authority for each transition. This was the security by design approach.

3.1.4 Authentication

In the example above, the authentication of the principal (the platoon leader) was assumed to be common knowledge. This was easy for soldiers who could verify the identity of an individual via visual inspection or voice recognition. In the world of computers and/or wherein an individual was not known (such as a security gate check-point or a computer terminal), more formal methods of authentication were required. Although time constraints did not allow us to consider that for this project, the applicability of the secure state machine approach had already been demonstrated elsewhere using cryptographic functions, key cards, etc. This was also discussed in the Future Work section.

For the purposes of this project, the authenticated persons were listed in an *authentication test function*. This function was a list of the form:

- Platoon Leader says someCommand = true,
- Platoon Sergeant says someCommand = true,
- Anything else = false.

Additional principals could have been listed or deleted as needed by each secure state machine. Two principals were the maximum needed to the extent to which this project was implemented in HOL. Most states required only one principal, namely the platoon leader. Each secure state machine had its own *authentication test function* with its own list of principals. In the implementation, if a principal other than those listed in the *authentication test function* were to make a request, HOL would not allow the transition (or any other action) to occur.

The top-level state machine was implemented with only one principal, namely the platoon leader. In other words, the platoon leader was responsible all that commands in the top-level state machine. When

the sub-level state machines were implemented, two state machines included two principals each making commands, namely the platoon leader and the platoon sergeant. This caused some issues with HOL because of the nature of the biconditional. If an authenticated and authorized principal gives the command to change states then we proved in HOL that the transition should occur. On the converse, if the transition occurred then we also proved in HOL that an authenticated and authorized principal must have made the command. But, what if there were two authenticated and authorized principals? Which one made the command? The problem wasn't overly challenging; however, it did require a different strategy than was used for the one principal secure state machines. The simplest approach was to allow each principal to control a specific set of commands. Thus, there were platoon leader commands and platoon sergeant commands. If a transition was made using a platoon leader command, then the platoon leader must have made the command. The same was true for the platoon sergeant. The biconditional problem was solved. Nevertheless, there were several other strategies which were explored and could have been implemented in other secure state machines to demonstrate the flexibility of HOL.

3.1.5 Authorization

In the ACL secure state machine approach, authentication was governed by a list of statements dictating who controls what and under what conditions that control was governed. This was essentially a list of trust assumptions, certificates, authorities, etc. For the level of complexity that was implemented in this project, access was dictated by a prescribed *security context list*, consisting of statements such as Platoon Leader **controls** crossLD. In other words, this was the ACL representation of the policy stating that the Platoon Leader has the authority to execute the command crossLD. Of course, representation of the statement in HOL was more complex. Nevertheless, it amounted to just this declaration of authority.

An additional level of complexity was added to the *security context* list of the ssmPlanPB state machine. It was also planned for other state machines at the sub-sub-level. This was necessary as the complexity of the project grew. In previously implemented secure state machines in the project, transitions from one state to another were sequential. To get from one state, the previous state must have been completed. For the ssmPlanPB state machine, the transition from the WARNO state to the COMPLETE state was to occur after the middle three states were completed. However, these states did not need to be completed in any order. But, all three states *had to be completed* before transitioning to the COMPLETE state. The easiest implementation approach involved allowing a list of inputs to the state machine. For example, the transition from WARNO to COMPLETE was to occur only if a list of requests/commands, say [a,b,c], was given as input, where each letter represented a request (i.e., a = Platoon Leader says state a is complete). The order of this list was irrelevant. There were thus six permutations of the list that could be passed as input to the state machine. It worked.

Nevertheless, the use of lists and permutations was not the most elegant solution and could cause problems in integrating state machines later on (see future work). Thus, a second version of the ssmPlanPB state machine was generated and named testssmPlanPB. This used the approach described next.

Consider the case wherein the Platoon Leader (or other principal) had the authority to execute a command if and only if some set of conditions were met. For example, a, b, and c must have occur before the command could be executed. a, b, and c could have been the following statements: Platoon Leader says state 2 is complete; Platoon Sergeant says state 3 is complete; and Platoon Leader says state 4 is complete. Under those conditions, we wanted the Platoon Leader to have the authority to execute a change of state from WARNO to COMPLETE. To do this in the ACL, a line in the *security context* list would look like the following:

- Platoon Leader **says** state 2 is complete and
- Platoon Sergeant **says** state 3 is complete and
- Platoon Leader **says** state 4 is complete implies

- Platoon leader **controls** complete.

Thus, when the Platoon Leader says complete, our theorem prover would check the *security context* list, check that all the conditions in the list were met, and then execute the transition if and only if all conditions were met. This was the approach implemented in testssmPlanPB.

In addition, we knew because of the reliability of HOL that if the transition was executed, then all of the said conditions above were met, namely.

- If we are now in the state COMPLETE, then it is also true that
- Platoon Leader **says** complete is true (that is, the Platoon Leader made the request), and
- Platoon Leader **says** state 2 is complete is true, and
- Platoon Sergeant **says** state 3 is complete is true, and
- Platoon Leader **says** state 4 is complete is also true.

Other conditions could have been added and were discussed for future work.

3.1.6 State Interpretation

In addition to authentications and authorizations, the behavior of the state machine could have been controlled by the definitions that were state-dependents. This approach was briefly considered, in particular for the two-principal problem discussed above. However, time constraints favored the simpler approach discussed. Nevertheless, it was worth mentioning and could be explored if the project were to be extended.

3.1.7 Implementation of the Secure State Machine Model

For reasons pertaining to simplicity and experience, Professor Shiu-Kai Chins model of a secure state machine was employed to implement a parameterizable secure state machine in HOL. A few modifications were made by Lori Pickering. Parameterization allowed us to construct a functioning secure state machine and then use that for all the secure state machines in our project without re-proving recurring theorems. The secure state machine implementation was implemented in the file ssm11Script.sml. This documentation referred to it as ssm11 or ssm11Theory. *ssm* is an acronym for secure state machine. This nomenclature was used throughout this project to indicate a secure state machine. For example, ssmPlanPB represented the PlanPB secure state machine, which was implemented in the ssmPlanPBScript.sml file. Note that all theory files in HOL were implemented in nameScript.sml. Once compiled with Holmake, they were included in subsequent theories and definitions as nameTheory.

ssm11 had several functions that were necessary for the proper functioning of a secure state machine. First, it defined the types of allowed transition:

- exec: execute a command.
 - This happened if and only if the command was given by an *authenticated* and *authorized* principal.
- trap: trap a command.
 - This happened if and only if the command was given by an *authenticated* principal who was *NOT authorized* to give that command.
- discard: discard the command.
 - This happened if the principal was not authenticated. The command was also discarded if it was not properly formatted.

(The names were selected by Professor Shiu-Kai Chin and were chosen based on his research with virtual machines.)

How did HOL know how to interpret the state of the state machine, the security context, the input, the output, the authentication test, and the state specific instructions? This was done by describing a configuration representing the current state of the machine. The *configuration* had the following components:

- Authentication test function
- Security context list
- State interpretation function (state specific rules)
- Current state
- input
- output

The authentication test function, security context list, and state interpretation function were those described in the discussions above. The current state was simply the current state of the machine, before transition. The input was typically the request, i.e., Platoon Leader says someCommand. The output for this project was kept simple. The output for any transition was the name of the state of the machine after transition. To distinguish it from the state, the outputs were denoted by an initial capital letter followed by mostly lowercase letters, i.e., if the state machine transitioned from PLAN_PB to MOVE_TO_ORP then the output would be MoveToORP. The difference between names of outputs and names of inputs was that the input name because with a lowercase letter.

In addition, HOL needed to know what the configuration meant with respect to the ACL and HOL logic. A *configuration interpretation functions* was also included in ssm11. This function combined everything into a list of ACL statements. HOL used these statements to prove things about the secure state machine. To do this, HOL needed an additional theory named satListTheory, which helped HOL to interpret lists in the ACL logic. This theory was written previously by Professor Shiu-Kai Chin.

Next, HOL needed to know how to deal with the state machine transitions. TR_rules, TR_cases, and TR_ind are the three functions that were defined to do just this. These functions defined the behavior for each of the transition types: exec, trap, and discard. Thus, for the transition type exec, the rules was defined as follows:

- If the following are true
 - *authentication test function* for the input passes, and
 - *configuration test function* for the configuration passes,
- Then, define the following transition behavior for the exec command (note that this takes the secure state machine from *configuration 1* to *configuration 2*)
 - Configuration 1;
 - * *authentication test function*
 - * *configuration test function*
 - * *security context list*
 - * [someComand, next command, etc] (input list)
 - * Initial state
 - * Output
 - Configuration 2

- * *authentication test function*
- * *configuration test function*
- * *security context list*
- * [next command, etc] (input list without initial input)
- * NS initial state (exec someCommand)
- * Out output (exec someCommand)

In essence, this definition told HOL how the configuration of the state machine changed from *configuration 1* to *configuration 2* given the transition type *exec* was followed by a specific command of type *someCommand*. Both explicit and implicit in the definition was the security context (*security context list*) with the proper authentications and authorities. The *configuration test function* includes the *security context list* as a parameter.

The difference between the two configurations was defined by the *NS* (next state) and *Out* (next output) functions. These functions take the current state and current output, respectively, and return the next state and next output, respectively. *NS* and *Out* are the parameterizable functions. These are defined in each secure state machine definition *ssmNameScript.sml* that uses (includes) *ssm11*. In addition, the *someCommand* commands were defined in the parameterizing secure state machine types definition file *nameTypeScript.sml*. Similar definitions were defined for the *trap* and *discard* transition types.

3.1.8 Parameterization of the *ssm11* secure state machine

The parametrizing secure state machines were implemented in two separate files. The first file was the types definition file *nameTypeScript.sml* and the second file was the secure state machine definition *ssmNameScript.sml*. These are the parameterizing secure state machines, not the parameterizable secure state machine *ssm11*. They did the following:

- *nameTypeScript.sml*
 - Define the specific commands for each principal, the states for the state machine, the output, and the principals for the state machine.
- *ssmNameScript.sml*
 - Define the next state relations
 - Define the next output relations
 - Define the *authentication test function*
 - Define the *state interpretation function*
 - Prove that commands without principals are rejected.
 - Define the *security context list*
 - Prove that a transition was made if and only if an authenticated and authorized principal gave the command. This often required one or more lemmas to prove. In addition, for multiple principals or different types of commands, more than one proof was needed.

The later theorem in *ssmNameScript.sml* was the culmination of all the previous theorems, definitions, and lists. It proved that a command was executed by a principal if and only if that principal was authenticated and authorized and gave the request/command to make the transition. HOL performed all the necessary checks based on previous definitions and theorems, which is the beauty of HOL. Additional theorems could be proved as needed. Given more time, this would prove useful depending on the intended application of the project.

3.1.9 Future Work – Some Notes

Integrating the secure state machines

One idea that I had an interest in exploring was integrating the state machines at either the top level or at each individual level. That is, I was interested in writing a sub-level secure state machine ssmSL that would transition from one secure state machine to the next. More specifically, the top-level states were PLAN_PB, MOVE_TO_ORP, CONDUCT_ORP, MOVE_TO_PB, CONDUCT_PB, and COMPLETE. Each of these states, save for the terminal state COMPLETE, was implemented at the sub-level as a secure state machine. Thus, each state in the top-level secure state machine was itself a secure state machine at the sub-level. These were each implemented in separate folders and separate *.sml files. Each of these were sub-folders in the sub level folder. The idea would be to generate a separate folder herein and a .sml file that would transition from the COMPLETE state of one sub-level secure state machine to the next sub-level secure state machine. And, so on. This could be done at the sub-sub-level as well. An alternative approach to the integration problem was to create an integrating secure state machine above the top-level state machine, in what was called the OMNI level state machine.

Alternatives to the sub-sub-sub-level state machine

At the sub-sub-level we began to sway from the sub levels approach and consider defining platoon, squad, and soldier theories. These theories would define Boolean functions or datatypes that would indicated the degree of readiness of a platoon, squad, or soldier. For example, the platoon in a sub-sub-level secure state machine may require that the orders be read back to the headquarters to verify receipt and correctness. The platoon theory may contain a Boolean function *ordersReadyReady* = false (by default). To transition to the next state in the sub-sub-level state machine, *ordersReadyReady* = true, would be a condition required. This would likely be added to the *security context list*, but defined in the platoon theory file. Other such functions would be defined in the platoon, squad, and soldier theories as required to adequately represent the patrol base. The ideas were hashed out and would be reasonably straight forward to implement given sufficient time.

Authentication

Our approach at assuming that everybody knows whos who works for this project. However, if the patrol base were implemented in a [add a reference to what Jesse discussed], then authentication would require a password, key-card, or even a bar code on a soldiers iPhone. In this case, we would need to take our secure state machine to the next level and require cryptographic operations on identity. A parameterizable secure state machine that includes these features has already been worked out by Professor Shiu-Kai Chin. In anaology to the ssm11 that I modified from Professor Chins ssm1, there already exists an ssm2. ssm2 extends ssm1 to include cryptographic checks on identify. It could be used in the same manner as ssm1 (my modified version was ssm11) without appropriate changes to the project. That is, it wouldnt be too much work to make the transition from the authenticate-by-visual to authenticate-by-password (or other form). This approach would lend itself well to the idea of [add a reference to what Jesse discussed], discussed in section name of section. The states were organized into discrete subtasks, illustrated in figure 3.2 with blue



Figure 3.2: PB Top Level States Overview

circles and the states name in the center. Each state had easily discernable actions for entering and exiting

the state, given above the line connecting adjacent states. For simplicity, the majority of commands were just named after the state that would be initiated when the command was given (i.e. moveToPB initiates the MOVE_TO_PB state), with the exceptions of the PLAN_PB state which would be initiated by Higher Head Quarters (HHQ) when they transmitted an operations order (OPORD) and the MOVE_TO_ORP state which would be initiated by the command given by the Platoon Leader (PL) to cross the line of departure (LD).

Each state was given a loop, labeled incomplete, which would simply allow the Platoon time to execute the tasks associated with each state.

We recognized that the state machine we initially designed, here after referred to as top-level state machine, required alternate states that would represent conclusions other than a successful completion of a PB. Three scenarios were derived:

1. The Platoon received a new mission.
2. The Platoon came under attack.
3. The Platoon was rendered non-mission capable (NMC) by either casualty (i.e. too many Soldiers fell ill or died), or lack of supplies.

States corresponding to these scenarios were developed: change mission, react to contact, and return to base respectively (illustrated in figure 3.3 with red circles). At any state in the PB top-level state machine, save for the complete state which represents a successful PB operation, these alternate termination states could be reached and essentially terminate the machines execution (see figure 3.3).

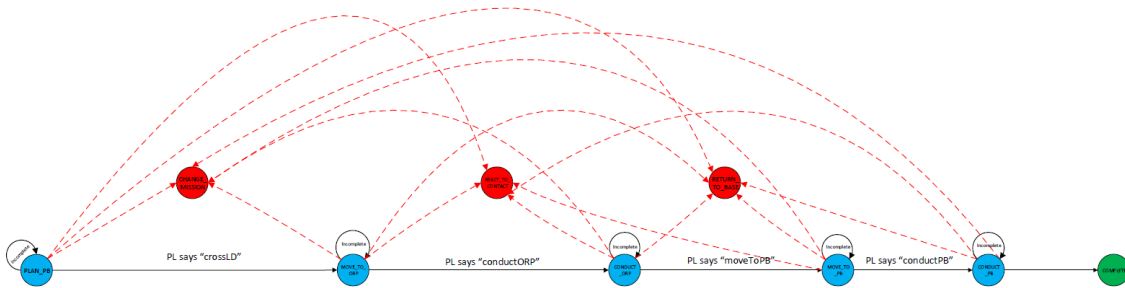


Figure 3.3: PB Top Level Alternate States Overview

Each of the top-level states represented a number of explicit tasks that we extrapolated from the Ranger Handbook (Ranger Handbook 2011):

- The planning state for the PB (PLAN_PB) contained sub states corresponding to the Troop Leading Procedures (TLPs) outlined in chapter 2 section 1 of the Ranger Handbook (Ranger Handbook 2011, 23).
- Movement to the objective rally point (MOVE_TO_ORP) sub states were crafted from chapter 6 (Ranger Handbook 2011, 101).
- Sub states corresponding to the were found in chapter 7, section 20 (Ranger Handbook 2011, 131).
- Move to PB sub states were formed similarly to those in move to ORP.

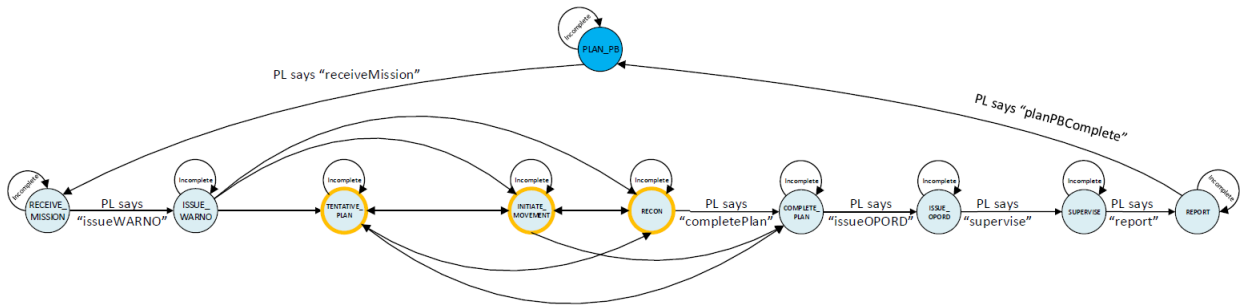


Figure 3.6: TLP PLAN_PB unordered sub states

Chapter 4

Results and Discussions

4.1 Overall Results and Discussion

The design phase of the patrol base was extensive.

4.2 Design Results and Discussion

Jesse should discuss the design as a whole. Or indicate that the discussion here would follow along the same lines as the discussion in the Methods section.

4.3 Implementation Results and Discussion

This section contains a more detailed description of one of the secure state machines. The ssmPB state machine was the first state machine implemented. It parameterized the ssm11 secure state machine. Subsequent state machines were designed in a similar matter, in almost a cut-and-paste manner. Exception were discussed in section 3.3. This discussion is intended to give the reader an understanding of the HOL implementation In more detail by providing a walk-through of the ssmPBScript.sml and PBTypeScript.sml files.

4.3.1 ssm11Script.sml, satListScript.sml, and ssminfRules.sml

General Description of Files

These files include the parameterizable secure state machine along with supporting theorems.

ssm11Script.sml

This is the parameterizable secure state machine. ssm11 is an acronym for secure state machine version 1.1. The first version was written by Professor Shiu-Kai Chin. ssm11 makes a few modifications to Dr. Chins original ssm1Thoery.

This file defines the basic functions needed to implement a secure state machine in the Higher Order Logic (HOL) theorem prover.

What follows, follows the order in the script file itself. For a list of datatypes and theorems based on datatype of theorem classification, see Appendix [add appendix for the HOLReportsOMNI pdf(OMNIReport.pdf)].

- All theorems starting with val save_thm save the theorem to the name in quotes and defined by the name following.

- order

order = SOME 'command | NONE

- order_distinct_clauses

$$\vdash \forall a. \text{SOME } a \neq \text{NONE}$$

- order_one_one

$$\vdash \forall a \ a'. (\text{SOME } a = \text{SOME } a') \iff (a = a')$$

- trType

$$\text{trType} = \text{discard 'command} \mid \text{trap 'command} \mid \text{exec 'command}$$

- trType_distinct_clauses

$$\begin{aligned} &\vdash (\forall a' \ a. \text{discard } a \neq \text{trap } a') \wedge \\ &(\forall a' \ a. \text{discard } a \neq \text{exec } a') \wedge \\ &\forall a' \ a. \text{trap } a \neq \text{exec } a' \end{aligned}$$

- tyType_one_one

$$\begin{aligned} &\vdash (\forall a \ a'. (\text{discard } a = \text{discard } a') \iff (a = a')) \wedge \\ &(\forall a \ a'. (\text{trap } a = \text{trap } a') \iff (a = a')) \wedge \\ &\forall a \ a'. (\text{exec } a = \text{exec } a') \iff (a = a') \end{aligned}$$

- configuration

$$\begin{aligned} \text{configuration} = &\text{CFG } ((\text{'command order, 'principal, 'd, 'e) Form } -i \text{ bool}) \\ &(\text{'state } -i \text{ ('command order, 'principal, 'd, 'e) Form}) \\ &((\text{'command order, 'principal, 'd, 'e) Form list}) \\ &((\text{'command order, 'principal, 'd, 'e) Form list}) \text{'state} \\ &(\text{'output list}) \end{aligned}$$

- configuration_one_one

$$\begin{aligned} &\vdash \forall a_0 \ a_1 \ a_2 \ a_3 \ a_4 \ a_5 \ a'_0 \ a'_1 \ a'_2 \ a'_3 \ a'_4 \ a'_5. \\ &(\text{CFG } a_0 \ a_1 \ a_2 \ a_3 \ a_4 \ a_5 = \text{CFG } a'_0 \ a'_1 \ a'_2 \ a'_3 \ a'_4 \ a'_5) \iff \\ &(a_0 = a'_0) \wedge (a_1 = a'_1) \wedge (a_2 = a'_2) \wedge (a_3 = a'_3) \wedge (a_4 = a'_4) \wedge \\ &(a_5 = a'_5) \end{aligned}$$

- CFGInterpret_def

$$\begin{aligned} &\vdash \text{CFGInterpret } (M, Oi, Os) \\ &(\text{CFG authenticationTest stateInterp securityContext } (\text{input}::\text{ins}) \\ &\text{state outputStream}) \iff \\ &(M, Oi, Os) \text{ satList securityContext } \wedge (M, Oi, Os) \text{ sat input } \wedge \\ &(M, Oi, Os) \text{ sat stateInterp state} \end{aligned}$$

- TR_rules

$$\begin{aligned} &\vdash (\forall \text{authenticationTest } P \ NS \ M \ Oi \ Os \ Out \ s \ \text{securityContext } \text{stateInterp} \\ &\text{cmd } \text{ins } \text{outs}. \\ &\text{authenticationTest } (P \text{ says prop } (\text{SOME } \text{cmd})) \wedge \\ &\text{CFGInterpret } (M, Oi, Os) \\ &(\text{CFG authenticationTest stateInterp securityContext} \\ &(P \text{ says prop } (\text{SOME } \text{cmd}::\text{ins}) \ s \ \text{outs}) \Rightarrow \\ &\text{TR } (M, Oi, Os) (\text{exec } \text{cmd}) \\ &(\text{CFG authenticationTest stateInterp securityContext} \\ &(P \text{ says prop } (\text{SOME } \text{cmd}::\text{ins}) \ s \ \text{outs}) \\ &(\text{CFG authenticationTest stateInterp securityContext } \text{ins } (NS \ s \ (\text{exec } \text{cmd})) (\text{Out } s \ (\text{exec } \text{cmd}::\text{outs}))) \\ &\wedge \\ &(\forall \text{authenticationTest } P \ NS \ M \ Oi \ Os \ Out \ s \ \text{securityContext } \text{stateInterp } \text{cmd } \text{ins } \text{outs}. \end{aligned}$$

$\text{authenticationTest } (P \text{ says prop } (\text{SOME cmd})) \wedge$
 $\text{CFGInterpret } (M, Oi, Os)$
 $(\text{CFG authenticationTest stateInterp securityContext}$
 $(P \text{ says prop } (\text{SOME cmd}::\text{ins}) s \text{ outs}) \Rightarrow$
 $\text{TR } (M, Oi, Os) (\text{trap cmd})$
 $(\text{CFG authenticationTest stateInterp securityContext}$
 $(P \text{ says prop } (\text{SOME cmd}::\text{ins}) s \text{ outs})$
 $(\text{CFG authenticationTest stateInterp securityContext ins } (NS s (\text{trap cmd})) (\text{Out } s (\text{trap cmd}::\text{outs})))$
 \wedge
 $\forall \text{ authenticationTest NS M Oi Os Out s securityContext stateInterp cmd x ins outs.}$
 $\neg \text{ authenticationTest } x \Rightarrow$
 $\text{TR } (M, Oi, Os) (\text{discard cmd})$
 $(\text{CFG authenticationTest stateInterp securityContext } (x::\text{ins}) s \text{ outs})$
 $(\text{CFG authenticationTest stateInterp securityContext ins}$
 $(NS s (\text{discard cmd})) (\text{Out } s (\text{discard cmd}::\text{outs})))$

- TR_ind

$\vdash \forall TR'.$
 $(\forall \text{ authenticationTest P NS M Oi Os Out s securityContext stateInterp cmd ins outs.}$
 $\text{authenticationTest } (P \text{ says prop } (\text{SOME cmd})) \wedge$
 $\text{CFGInterpret } (M, Oi, Os)$
 $(\text{CFG authenticationTest stateInterp securityContext}$
 $(P \text{ says prop } (\text{SOME cmd}::\text{ins}) s \text{ outs}) \Rightarrow$
 $TR' (M, Oi, Os) (\text{exec cmd})$
 $(\text{CFG authenticationTest stateInterp securityContext}$
 $(P \text{ says prop } (\text{SOME cmd}::\text{ins}) s \text{ outs})$
 $(\text{CFG authenticationTest stateInterp securityContext ins}$
 $(NS s (\text{exec cmd})) (\text{Out } s (\text{exec cmd}::\text{outs}))) \wedge$
 $(\forall \text{ authenticationTest P NS M Oi Os Out s securityContext stateInterp}$
 cmd ins outs.
 $\text{authenticationTest } (P \text{ says prop } (\text{SOME cmd})) \wedge$
 $\text{CFGInterpret } (M, Oi, Os)$
 $(\text{CFG authenticationTest stateInterp securityContext}$
 $(P \text{ says prop } (\text{SOME cmd}::\text{ins}) s \text{ outs}) \Rightarrow$
 $TR' (M, Oi, Os) (\text{trap cmd})$
 $(\text{CFG authenticationTest stateInterp securityContext}$
 $(P \text{ says prop } (\text{SOME cmd}::\text{ins}) s \text{ outs})$
 $(\text{CFG authenticationTest stateInterp securityContext ins}$
 $(NS s (\text{trap cmd})) (\text{Out } s (\text{trap cmd}::\text{outs}))) \wedge$
 $(\forall \text{ authenticationTest NS M Oi Os Out s securityContext stateInterp}$
 cmd x ins outs.
 $\neg \text{ authenticationTest } x \Rightarrow$
 $TR' (M, Oi, Os) (\text{discard cmd})$
 $(\text{CFG authenticationTest stateInterp securityContext } (x::\text{ins}) s$
 $\text{outs})$
 $(\text{CFG authenticationTest stateInterp securityContext ins}$
 $(NS s (\text{discard cmd})) (\text{Out } s (\text{discard cmd}::\text{outs}))) \Rightarrow$
 $\forall a_0 a_1 a_2 a_3. \text{ TR } a_0 a_1 a_2 a_3 \Rightarrow TR' a_0 a_1 a_2 a_3$

- TR_cases

$\vdash \forall a_0 a_1 a_2 a_3.$
 $\text{TR } a_0 a_1 a_2 a_3 \iff$
 $(\exists \text{ authenticationTest P NS M Oi Os Out s securityContext stateInterp}$

$cmd\ ins\ outs.$
 $(a_0 = (M, Oi, Os)) \wedge (a_1 = \text{exec } cmd) \wedge$
 $(a_2 = \text{CFG authenticationTest stateInterp securityContext}$
 $(P \text{ says prop (SOME } cmd)::ins) s\ outs) \wedge$
 $(a_3 =$
 $\text{CFG authenticationTest stateInterp securityContext ins (NS } s (\text{exec } cmd)) (Out\ s (\text{exec } cmd)::outs))$
 \wedge
 $\text{authenticationTest } (P \text{ says prop (SOME } cmd)) \wedge$
 $\text{CFGInterpret } (M, Oi, Os)$
 $(\text{CFG authenticationTest stateInterp securityContext}$
 $(P \text{ says prop (SOME } cmd)::ins) s\ outs)) \vee$
 $(\exists \text{ authenticationTest } P\ NS\ M\ Oi\ Os\ Out\ s\ securityContext\ stateInterp$
 $cmd\ ins\ outs.$
 $(a_0 = (M, Oi, Os)) \wedge (a_1 = \text{trap } cmd) \wedge$
 $(a_2 = \text{CFG authenticationTest stateInterp securityContext}$
 $(P \text{ says prop (SOME } cmd)::ins) s\ outs) \wedge$
 $(a_3 =$
 $\text{CFG authenticationTest stateInterp securityContext ins (NS } s (\text{trap } cmd)) (Out\ s (\text{trap } cmd)::outs))$
 \wedge
 $\text{authenticationTest } (P \text{ says prop (SOME } cmd)) \wedge$
 $\text{CFGInterpret } (M, Oi, Os)$
 $(\text{CFG authenticationTest stateInterp securityContext}$
 $(P \text{ says prop (SOME } cmd)::ins) s\ outs)) \vee$
 $\exists \text{ authenticationTest } NS\ M\ Oi\ Os\ Out\ s\ securityContext\ stateInterp$
 $cmd\ x\ ins\ outs.$
 $(a_0 = (M, Oi, Os)) \wedge (a_1 = \text{discard } cmd) \wedge$
 $(a_2 =$
 $\text{CFG authenticationTest stateInterp securityContext } (x::ins) s$
 $outs) \wedge$
 $(a_3 =$
 $\text{CFG authenticationTest stateInterp securityContext ins (NS } s (\text{discard } cmd)) (Out\ s (\text{discard}$
 $cmd)::outs)) \wedge$
 $\neg \text{authenticationTest } x$

- rule0, rule1, rule2

- TR_EQ_rules.thm

$\vdash (\text{TR } (M, Oi, Os) (\text{exec } cmd)$
 $(\text{CFG authenticationTest stateInterp securityContext}$
 $(P \text{ says prop (SOME } cmd)::ins) s\ outs)$
 $(\text{CFG authenticationTest stateInterp securityContext ins}$
 $(NS\ s (\text{exec } cmd)) (Out\ s (\text{exec } cmd)::outs)) \iff$
 $\text{authenticationTest } (P \text{ says prop (SOME } cmd)) \wedge$
 $\text{CFGInterpret } (M, Oi, Os)$
 $(\text{CFG authenticationTest stateInterp securityContext}$
 $(P \text{ says prop (SOME } cmd)::ins) s\ outs)) \wedge$
 $(\text{TR } (M, Oi, Os) (\text{trap } cmd) (\text{CFG authenticationTest stateInterp securityContext}$
 $(P \text{ says prop (SOME } cmd)::ins) s\ outs)$
 $(\text{CFG authenticationTest stateInterp securityContext ins}$
 $(NS\ s (\text{trap } cmd)) (Out\ s (\text{trap } cmd)::outs)) \iff$
 $\text{authenticationTest } (P \text{ says prop (SOME } cmd)) \wedge$
 $\text{CFGInterpret } (M, Oi, Os)$
 $(\text{CFG authenticationTest stateInterp securityContext}$

$$\begin{aligned}
& (P \text{ says prop } (\text{SOME } cmd)::ins) s \text{ outs}) \wedge \\
& (\text{TR } (M, Oi, Os) (\text{discard } cmd)) \\
& (\text{CFG authenticationTest stateInterp securityContext } (x::ins) s \text{ outs}) \\
& (\text{CFG authenticationTest stateInterp securityContext } ins \\
& (\text{NS } s (\text{discard } cmd)) (\text{Out } s (\text{discard } cmd)::outs)) \iff \\
& \neg \text{authenticationTest } x
\end{aligned}$$

- TRrule0

$$\begin{aligned}
& \vdash \text{TR } (M, Oi, Os) (\text{exec } cmd) \\
& (\text{CFG authenticationTest stateInterp securityContext } \\
& (P \text{ says prop } (\text{SOME } cmd)::ins) s \text{ outs}) \\
& (\text{CFG authenticationTest stateInterp securityContext } ins \\
& (\text{NS } s (\text{exec } cmd)) (\text{Out } s (\text{exec } cmd)::outs)) \iff \\
& \text{authenticationTest } (P \text{ says prop } (\text{SOME } cmd)) \wedge \\
& \text{CFGInterpret } (M, Oi, Os) \\
& (\text{CFG authenticationTest stateInterp securityContext } \\
& (P \text{ says prop } (\text{SOME } cmd)::ins) s \text{ outs})
\end{aligned}$$

- TRrule1

$$\begin{aligned}
& \vdash \text{TR } (M, Oi, Os) (\text{trap } cmd) \\
& (\text{CFG authenticationTest stateInterp securityContext } \\
& (P \text{ says prop } (\text{SOME } cmd)::ins) s \text{ outs}) \\
& (\text{CFG authenticationTest stateInterp securityContext } ins \\
& (\text{NS } s (\text{trap } cmd)) (\text{Out } s (\text{trap } cmd)::outs)) \iff \\
& \text{authenticationTest } (P \text{ says prop } (\text{SOME } cmd)) \wedge \\
& \text{CFGInterpret } (M, Oi, Os) \\
& (\text{CFG authenticationTest stateInterp securityContext } \\
& (P \text{ says prop } (\text{SOME } cmd)::ins) s \text{ outs})
\end{aligned}$$

- TRrule2

- TR_discard_cmd_rule

$$\begin{aligned}
& \vdash \text{TR } (M, Oi, Os) (\text{discard } cmd) \\
& (\text{CFG authenticationTest stateInterp securityContext } (x::ins) s \text{ outs}) \\
& (\text{CFG authenticationTest stateInterp securityContext } ins \\
& (\text{NS } s (\text{discard } cmd)) (\text{Out } s (\text{discard } cmd)::outs)) \iff \\
& \neg \text{authenticationTest } x
\end{aligned}$$

- TR_exec_cmd_rule

$$\begin{aligned}
& \vdash \forall \text{authenticationTest securityContext stateInterp } P \text{ cmd } ins \text{ s outs.} \\
& (\forall M \text{ Oi } Os. \\
& \text{CFGInterpret } (M, Oi, Os) \\
& (\text{CFG authenticationTest stateInterp securityContext } \\
& (P \text{ says prop } (\text{SOME } cmd)::ins) s \text{ outs}) \Rightarrow \\
& (M, Oi, Os) \text{ sat prop } (\text{SOME } cmd)) \Rightarrow \\
& \forall \text{NS Out } M \text{ Oi } Os. \\
& \text{TR } (M, Oi, Os) (\text{exec } cmd) \\
& (\text{CFG authenticationTest stateInterp securityContext } \\
& (P \text{ says prop } (\text{SOME } cmd)::ins) s \text{ outs}) \\
& (\text{CFG authenticationTest stateInterp securityContext } ins \\
& (\text{NS } s (\text{exec } cmd)) (\text{Out } s (\text{exec } cmd)::outs)) \iff \\
& \text{authenticationTest } (P \text{ says prop } (\text{SOME } cmd)) \wedge
\end{aligned}$$

```

CFGInterpret (M, Oi, Os)
  (CFG authenticationTest stateInterp securityContext
   (P says prop (SOME cmd)::ins) s outs) ∧
  (M, Oi, Os) sat prop (SOME cmd)

```

- TR_trap_cmd_rule

satListScript.sml

This file contains supporting theorems needed to work with sat lists (sat represents satisfies in the access-control logic (ACL)).

- satList_def

```

⊢ ∀ M Oi Os formList.
  (M, Oi, Os) satList formList ⇔
  FOLDR (λ x y. x ∧ y) T (MAP (λ f. (M, Oi, Os) sat f) formList)

```

- satList_nil

```

⊢ (M, Oi, Os) satList []

```

- satList_conj

```

⊢ ∀ l1 l2 M Oi Os.
  (M, Oi, Os) satList l1 ∧ (M, Oi, Os) satList l2 ⇔
  (M, Oi, Os) satList (l1 ++ l2)

```

- satList_CONS

```

⊢ ∀ h t M Oi Os.
  (M, Oi, Os) satList (h::t) ⇔
  (M, Oi, Os) sat h ∧ (M, Oi, Os) satList t

```

ssminfRules.sml

This file contains supporting functions for ssm11.

- flip_imp
- flip_TR_rules
- TR_EQ_rules
- Distinct_clauses

4.3.2 PBTypeScript.sml and ssmPBScript.sml

General description of files

PBTypeScript.sml

This file defines the basic datatypes for the ssmPB secure state machine.

What follows, follows the order in the script file itself. For a list of datatypes and theorems based on datatype of theorem classification, see Appendix [add appendix for the HOLReportsOMNI pdf(OMNIReport.pdf)]

- `slCommand`

```
slCommand = crossLD
           |conductORP
           |moveToPB
           |conductPB
           |completePB
           |incomplete
```

State-level command. This datatype includes all the allowable state transition commands. Each command takes the state machine from one state to another. The one exception is the incomplete command which takes the state machine from one state to that same state. This applies to all states except for the terminal state COMPLETE.

- `slState`

```
slState = PLAN_PB
         |MOVE_TO_ORP
         |CONDUCT_ORP
         |MOVE_TO_PB
         |CONDUCT_PB
         |COMPLETE_PB
```

State-level state. This datatype describes all the states that are specific to this state machine.

- `slOutput`

```
slOutput = PlanPB
          |MoveToORP
          |ConductORP
          |MoveToPB
          |ConductPB
          |CompletePB
          |unAuthenticated
```

State-level output. This datatype includes all outputs for the state machine. Each output is description of the current state (or next state if a transition occurred).

- `stateRole`

```
stateRole = PlatoonLeader
```

State role. This datatype is a list of all principals in this state machine. This particular state has only one principals, namely PlatoonLeader.

ssmPBScript.sml

This file describes the theorems and definitions for the ssmPB secure state machine. This is the top-level state machine for the patrol base. ssmPB is an acronym for secure state machine patrol base.

What follows, follows the order in the script file itself. For a list of datatypes and theorems based on datatype of theorem classification, see Appendix [add appendix for the HOLReportsOMNI pdf(OMNIReport.pdf)]

- PBNS_def

$$\begin{aligned}
&\vdash (\text{PBNS PLAN_PB (exec (SLc crossLD))} = \text{MOVE_TO_ORP}) \wedge \\
&(\text{PBNS PLAN_PB (exec (SLc incomplete))} = \text{PLAN_PB}) \wedge \\
&(\text{PBNS MOVE_TO_ORP (exec (SLc conductORP))} = \text{CONDUCT_ORP}) \wedge \\
&(\text{PBNS MOVE_TO_ORP (exec (SLc incomplete))} = \text{MOVE_TO_ORP}) \wedge \\
&(\text{PBNS CONDUCT_ORP (exec (SLc moveToPB))} = \text{MOVE_TO_PB}) \wedge \\
&(\text{PBNS CONDUCT_ORP (exec (SLc incomplete))} = \text{CONDUCT_ORP}) \wedge \\
&(\text{PBNS MOVE_TO_PB (exec (SLc conductPB))} = \text{CONDUCT_PB}) \wedge \\
&(\text{PBNS MOVE_TO_PB (exec (SLc incomplete))} = \text{MOVE_TO_PB}) \wedge \\
&(\text{PBNS CONDUCT_PB (exec (SLc completePB))} = \text{COMPLETE_PB}) \wedge \\
&(\text{PBNS CONDUCT_PB (exec (SLc incomplete))} = \text{CONDUCT_PB}) \wedge \\
&(\text{PBNS } s \text{ (trap (SLc cmd))} = s) \wedge \\
&(\text{PBNS } s \text{ (discard (SLc cmd))} = s)
\end{aligned}$$

PBNS is the next state function for the PB (patrol base) top-level secure state machine. The next state, next output table is shown in table 4.1. This table shows the current state in the header column. The header row shows the commands. At the intersection of each row and column is the next state and next output that results from giving that command while in that state. For example, while in the PLAN_PB state, the command crossLD will take the state machine from the PLAN_PB state to the MOVE_TO_ORP state. The corresponding output will be MoveToORP. In general, the next output is the name of the state, formatted as an initial capital letter with mostly lower-case letters. Where there is no entry, the given command does not cause a change in the state machine. I.e., this is not an allowable command. For example, while in the PLAN_PB state, if the command moveToORP is given, then no transition occurs. I.e., this is not an allowable transition. There is also no change in the output.

Next State, Next Output Table							
Commands/ input	State	crossLD	conductORP	moveToPB	conductPB	completePB	incomplete
	PLAN_PB	MOVE_TO_ORP, MoveToORP					PLAN_PB, PlanPB
	MOVE_TO_ORP		CONDUCT_ORP, ConductORP				MOVE_TO_ORP, MoveToORP
	CONDUCT_ORP			MOVE_TO_PB, MoveToPB			CONDUCT_ORP, ConductORP
	MOVE_TO_PB				CONDUCT_PB, ConductPB		MOVE_TO_PB, MoveToPB
	CONDUCT_PB					COMPLETE_PB, CompletePB	CONDUCT_PB, ConductPB
	COMPLETE_PB						

Table 4.1: Next State, Next Output Table

- PBOut_def

$$\begin{aligned}
&\vdash (\text{PBOut PLAN_PB (exec (SLc crossLD))} = \text{MoveToORP}) \wedge \\
&(\text{PBOut PLAN_PB (exec (SLc incomplete))} = \text{PlanPB}) \wedge \\
&(\text{PBOut MOVE_TO_ORP (exec (SLc conductORP))} = \text{ConductORP}) \wedge \\
&(\text{PBOut MOVE_TO_ORP (exec (SLc incomplete))} = \text{MoveToORP}) \wedge \\
&(\text{PBOut CONDUCT_ORP (exec (SLc moveToPB))} = \text{MoveToPB}) \wedge \\
&(\text{PBOut CONDUCT_ORP (exec (SLc incomplete))} = \text{ConductORP}) \wedge \\
&(\text{PBOut MOVE_TO_PB (exec (SLc conductPB))} = \text{ConductPB}) \wedge \\
&(\text{PBOut MOVE_TO_PB (exec (SLc incomplete))} = \text{MoveToPB}) \wedge \\
&(\text{PBOut CONDUCT_PB (exec (SLc completePB))} = \text{CompletePB}) \wedge
\end{aligned}$$

$$\begin{aligned}
& (\text{PBOut CONDUCT_PB (exec (SLc incomplete))} = \text{ConductPB}) \wedge \\
& (\text{PBOut } s \text{ (trap (SLc cmd))} = \text{unAuthorized}) \wedge \\
& (\text{PBOut } s \text{ (discard (SLc cmd))} = \text{unAuthenticated})
\end{aligned}$$

PBNS is the next output function for the PB (patrol base) top-level secure state machine. The next state, next output table is shown in table 4.1. See description for PBNS.

- authenticationTest_def

$$\begin{aligned}
& \vdash (\text{authenticationTest (Name PlatoonLeader says prop cmd)} \iff T) \wedge \\
& (\text{authenticationTest TT} \iff F) \wedge \\
& (\text{authenticationTest FF} \iff F) \wedge \\
& (\text{authenticationTest (prop } v) \iff F) \wedge \\
& (\text{authenticationTest (notf } v_1) \iff F) \wedge \\
& (\text{authenticationTest (} v_2 \text{ andf } v_3) \iff F) \wedge \\
& (\text{authenticationTest (} v_4 \text{ orf } v_5) \iff F) \wedge \\
& (\text{authenticationTest (} v_6 \text{ impf } v_7) \iff F) \wedge \\
& (\text{authenticationTest (} v_8 \text{ eqf } v_9) \iff F) \wedge \\
& (\text{authenticationTest (} v_{10} \text{ says TT)} \iff F) \wedge \\
& (\text{authenticationTest (} v_{10} \text{ says FF)} \iff F) \wedge \\
& (\text{authenticationTest (} v_{133} \text{ meet } v_{134} \text{ says prop } v_{66}) \iff F) \wedge \\
& (\text{authenticationTest (} v_{135} \text{ quoting } v_{136} \text{ says prop } v_{66}) \iff F) \wedge \\
& (\text{authenticationTest (} v_{10} \text{ says notf } v_{67}) \iff F) \wedge \\
& (\text{authenticationTest (} v_{10} \text{ says (} v_{68} \text{ andf } v_{69}) \iff F) \wedge \\
& (\text{authenticationTest (} v_{10} \text{ says (} v_{70} \text{ orf } v_{71}) \iff F) \wedge \\
& (\text{authenticationTest (} v_{10} \text{ says (} v_{72} \text{ impf } v_{73}) \iff F) \wedge \\
& (\text{authenticationTest (} v_{10} \text{ says (} v_{74} \text{ eqf } v_{75}) \iff F) \wedge \\
& (\text{authenticationTest (} v_{10} \text{ says } v_{76} \text{ says } v_{77}) \iff F) \wedge \\
& (\text{authenticationTest (} v_{10} \text{ says } v_{78} \text{ speaks_for } v_{79}) \iff F) \wedge \\
& (\text{authenticationTest (} v_{10} \text{ says } v_{80} \text{ controls } v_{81}) \iff F) \wedge \\
& (\text{authenticationTest (} v_{10} \text{ says reps } v_{82} \text{ } v_{83} \text{ } v_{84}) \iff F) \wedge \\
& (\text{authenticationTest (} v_{10} \text{ says } v_{85} \text{ domi } v_{86}) \iff F) \wedge \\
& (\text{authenticationTest (} v_{10} \text{ says } v_{87} \text{ eqi } v_{88}) \iff F) \wedge \\
& (\text{authenticationTest (} v_{10} \text{ says } v_{89} \text{ doms } v_{90}) \iff F) \wedge \\
& (\text{authenticationTest (} v_{10} \text{ says } v_{91} \text{ eqs } v_{92}) \iff F) \wedge \\
& (\text{authenticationTest (} v_{10} \text{ says } v_{93} \text{ eqn } v_{94}) \iff F) \wedge \\
& (\text{authenticationTest (} v_{10} \text{ says } v_{95} \text{ lte } v_{96}) \iff F) \wedge \\
& (\text{authenticationTest (} v_{10} \text{ says } v_{97} \text{ lt } v_{98}) \iff F) \wedge \\
& (\text{authenticationTest (} v_{12} \text{ speaks_for } v_{13}) \iff F) \wedge \\
& (\text{authenticationTest (} v_{14} \text{ controls } v_{15}) \iff F) \wedge \\
& (\text{authenticationTest (reps } v_{16} \text{ } v_{17} \text{ } v_{18}) \iff F) \wedge \\
& (\text{authenticationTest (} v_{19} \text{ domi } v_{20}) \iff F) \wedge \\
& (\text{authenticationTest (} v_{21} \text{ eqi } v_{22}) \iff F) \wedge \\
& (\text{authenticationTest (} v_{23} \text{ doms } v_{24}) \iff F) \wedge \\
& (\text{authenticationTest (} v_{25} \text{ eqs } v_{26}) \iff F) \wedge \\
& (\text{authenticationTest (} v_{27} \text{ eqn } v_{28}) \iff F) \wedge \\
& (\text{authenticationTest (} v_{29} \text{ lte } v_{30}) \iff F) \wedge \\
& (\text{authenticationTest (} v_{31} \text{ lt } v_{32}) \iff F)
\end{aligned}$$

This definition defines who the authenticated principals are in this secure state machine. For ssmPB, the only authenticatable principal is PlatoonLeader. This is the first part of the definition. The second part of the definition indicates that all other principals are F or false. That is, no other principals will be authenticated.

- `ssmPBStateInterp_def`

$$\vdash \forall state. \text{ssmPBStateInterp } state = \text{TT}$$

This definition describes conditions that are specific to each state. There are no specific conditions based on state in `ssmPB`. Therefore, this function is true (TT) by default.

- `authenticationTest_cmd_reject_lemma`

$$\vdash \forall cmd. \neg \text{authenticationTest } (\text{prop } (\text{SOME } cmd))$$

This theorem proves that any command not given by a principal is rejected. This follows directly from `authenticationTest_def` which sets `PlatoonLeader` says `someCommand` to true and anything else to false.

- `secContext_def`

$$\vdash \forall cmd. \text{secContext } cmd = [\text{Name } \text{PlatoonLeader } \text{controls } \text{prop } (\text{SOME } (\text{SLc } cmd))]$$

This is the heart of the security context of the secure state machine. This is where trust relations, policies, and conditions for authority are defined. `PlatoonLeader` is the only authenticated principal to have authority on state transitions. Therefore, there is only one entry in the security context list. This entry states that the `PlatoonLeader` has the authority (controls) to make a change in states using the `slCommand` type.

- `PlatoonLeader_slCommand_lemma`

$$\begin{aligned} &\vdash \text{CFGInterpret } (M, Oi, Os) \\ &(\text{CFG authenticationTest ssmPBStateInterp } (\text{secContext } \text{slCommand}) \\ &(\text{Name } \text{PlatoonLeader } \text{says prop } (\text{SOME } (\text{SLc } \text{slCommand}))::\text{ins}) \text{ s outs}) \Rightarrow \\ &(M, Oi, Os) \text{ sat prop } (\text{SOME } (\text{SLc } \text{slCommand})) \end{aligned}$$

This is a lemma developed to assist in proving the following theorem. It proves that, given the input `PlatoonLeader` says `slCommand`, then `slCommand` is true. If `slCommand` is true, then we can use this to prove that we are justified in making the transition from one state to another.

- `PlatoonLeader_exec_slCommand_justified_thm`

$$\begin{aligned} &\vdash \forall NS \text{ Out } M \text{ Oi } Os. \\ &\text{TR } (M, Oi, Os) (\text{exec } (\text{SLc } \text{slCommand})) \\ &(\text{CFG authenticationTest ssmPBStateInterp } (\text{secContext } \text{slCommand}) (\text{Name } \text{PlatoonLeader} \\ &\text{says prop } (\text{SOME } (\text{SLc } \text{slCommand}))::\text{ins}) \text{ s outs}) \\ &(\text{CFG authenticationTest ssmPBStateInterp } (\text{secContext } \text{slCommand}) \text{ ins } (NS \text{ s } (\text{exec } (\text{SLc} \\ &\text{slCommand})))) (Out \text{ s } (\text{exec } (\text{SLc } \text{slCommand}))::\text{outs})) \iff \\ &\text{authenticationTest } (\text{Name } \text{PlatoonLeader } \text{says prop } (\text{SOME } (\text{SLc } \text{slCommand}))) \wedge \\ &\text{CFGInterpret } (M, Oi, Os) (\text{CFG authenticationTest ssmPBStateInterp } (\text{secContext } \text{slCommand}) \\ &(\text{Name } \text{PlatoonLeader } \text{says prop } (\text{SOME } (\text{SLc } \text{slCommand}))::\text{ins}) \text{ s outs}) \wedge \\ &(M, Oi, Os) \text{ sat prop } (\text{SOME } (\text{SLc } \text{slCommand})) \end{aligned}$$

This theorem proves that a transition from one state to another is executed IF AND ONLY IF `PlatoonLeader` says `slCommand`

ConductORP *Next State*, Next Output Table

Commands/ input	State					
	next state, next output	secure	actionsIn	withdraw	complete	plIncomplete
	CONDUCT_ORP	SECURE, Secure				CONDUCT_ORP, ConductORP
	SECURE		ACTIONS_IN, ActionsIn			SECURE, Secure
	ACTIONS_IN			WITHDRAW, Withdraw		ACTIONS_IN, ActionsIn
	WITHDRAW				COMPLETE, Complete	WITHDRAW, Withdraw

ConductPB *Next State*, Next Output Table

Commands/ input	State					
	next state, next output	securePB	actionsInPB	withdrawPB	completePB	plIncompletePB
	CONDUCT_PB	SECURE_PB, SecurePB				CONDUCT_PB, ConductPB
	SECURE_PB		ACTIONS_IN_PB, ActionsInPB			SECURE_PB, Sec- urePB
	ACTIONS_IN_PB			WITHDRAW_PB, WithdrawPB		ACTIONS_IN_PB, ActionsInPB
	WITHDRAW_PB				COMPLETE_PB, CompletePB	WITHDRAW_PB, WithdrawPB

MoveToORP *Next State*, Next Output Table

Commands/ input	State					
	next state, next output	pltForm	pltMove	pltSecureHalt	complete	incomplete
	PLAN_PB	PLT_FORM, PLTForm				PLAN_PB, PLT- Plan
	PLT_FORM		PLT_MOVE, PLTMove			PLT_FORM, PLT- Form
	PLT_MOVE			PLT_SECURE_- HALT, PLTSecure- Halt		PLT_MOVE, PLT- Move
	PLT_SECURE_HALT				COMPLETE, Complete	PLT_SECURE_- HALT, PLTSecure- Halt

MoveToPB *Next State*, Next Output Table

Commands/ input	State					
	next state, next output	pltForm	pltMove	pltHalt	complete	incomplete
	PLAN_PB	PLT_FORM, PLTForm				PLAN_PB, PLTPlan
	PLT_FORM		PLT_MOVE, PLTMove			PLT_FORM, PLTForm
	PLT_MOVE			PLT_HALT, PLTSecureHalt		PLT_MOVE, PLTMove
	PLT_HALT				COMPLETE, Complete	PLT_HALT, PLTHalt

Chapter 5

Conclusions

5.1 Overall Conclusions

Overall, the project was a great idea. The initial challenges were in translating the patrol base operations into a design that could be implemented using ACL and then verified in HOL. At first, this seemed like a daunting task. Jesse was well-versed in the patrol base operations and the methods of the military. Lori was well-versed in the ACL and HOL. Yet, merging the two was not obvious at first. Once we decided on a state machine approach to solving the problem, everything ran smoothly. The beauty of the project was that it could be designed and built into any system that does check on equipment and the state of readiness of the patrol base operations at the platoon, squad, and soldier levels.

5.2 Future Work

This section discusses future work and other ideas from the perspective of each phase, the design phase and the implementation phase.

5.2.1 Design Phase

PB operations were well defined by a state machine. The well-defined chain of command, roles, and associated tasks yielded easily identifiable states and sub states that have discrete beginning and ending points. Design constraints inherent in state machines are manageable at this level of granularity (i.e. top-level and sub state level).

5.2.2 Implementation Phase

This section extrapolates on ideas for future work noted in the various sections above.

Integration

One idea that I had an interest in exploring was integrating the state machines at either the top level or at each individual level. That is, I was interested in writing a sub-level secure state machine ssmSL that would transition from one secure state machine to the next. More specifically, the top-level states were PLAN_PB, MOVE_TO_ORP, CONDUCT_ORP, MOVE_TO_PB, CONDUCT_PB, and COMPLETE. Each of these states, save for the terminal state COMPLETE, was implemented at the sub-level as a secure state machine. Thus, each state in the top-level secure state machine was itself a secure state machine at the sub-level. These were each implemented in separate folders and separate *.sml files. Each of these were sub-folders in the sub level folder. The idea would be to generate a separate folder herein and a .sml file that would transition from the COMPLETE state of one sub-level secure state machine to the next sub-level secure state machine. And, so on. This could be done at the sub-sub-level as well. An alternative approach to the integration problem was to create an integrating secure state machine above the top-level state machine, in what was called the OMNI level state machine.

Alternatives and Additions

At the sub-sub-level we began to sway from the sub levels approach and consider defining platoon, squad, and soldier theories. These theories would define Boolean functions or datatypes that would indicated the degree of readiness of a platoon, squad, or soldier. For example, the platoon in a sub-sub-level secure state machine may require that the orders be read back to the headquarters to verify receipt and correctness. The platoon theory may contain a Boolean function `ordersReadyReady = false` (by default). To transition to the next state in the sub-sub-level state machine, `ordersReadyReady = true`, would be a condition required. This would likely be added to the security context list, but defined in the platoon theory file. Other such functions would be defined in the platoon, squad, and soldier theories as required to adequately represent the patrol base. The ideas were hashed out and would be reasonably straight forward to implement given sufficient time.

Cryptographic Authentication

Our approach at assuming that everybody knows whos who works for this project. However, if the patrol base were implemented in a [add a reference to what Jesse discussed], then authentication would require a password, key-card, or even a bar code on a soldiers iPhone. In this case, we would need to take our secure state machine to the next level and require cryptographic operations on identity. A parameterizable secure state machine that includes these features has already been worked out by Professor Shiu-Kai Chin. In analogy to the `ssm11` that I modified from Professor Chins `ssm1`, there already exists an `ssm2`. `ssm2` extends `ssm1` to include cryptographic checks on identify. It could be used in the same manner as `ssm1` (my modified version was `ssm11`) without appropriate changes to the project. That is, it wouldnt be too much work to make the transition from the authenticate-by-visual to authenticate-by-password (or other form). This approach would lend itself well to the idea of [add a reference to what Jesse discussed], discussed in section name of section.

Chapter 6

Recommendations

6.1 Design Recommendations

The current level of documented states is limited to the top-level and first sub-level state machines. Exploration of further sub states to both ensure that the state machine is able to adequately model patrol base (PB) operations. A state machine model should include states down to the Soldier level, where applicable. Further sub states have been identified but, due to time constraints, were not able to be implemented (see APPENDIX A).

Additional assurance could be attained by constructing types in Poly/ML for entities within PB operations. Constructing a Unified Modeling Language (UML) class diagram with types for Soldiers, units, roles, missions and their objectives, as well as equipment and supplies could provide a structure that functions in Poly/ML could check for additional confirmation that a state is complete (see APPENDIX B). These types would hold Boolean values as members of types that, when used with logical operations, would prove whether a state is complete or not. As the state diagram is being completed down to the individual Soldier level, referencing the UML class diagram and noting what Boolean members within types would indicate completion of the state will allow for more complete data structures when implementing the types in Poly/ML.

Construction of a Data Flow Diagram (DFD) for sub state machines, i.e. those below the top-level state machine, provide design cues and illuminate possible problems. One problem that can be illuminated by use of a DFD is the need for iterative state execution: i.e. deficiencies are found in a later state that require a return to a previously completed state. For example, in our state machine the sub states of PLAN_PB require that INITIATE_MOVEMENT occur before SUPERVISE. If in the SUPERVISE state and the PL notices during rehearsals that additional supplies will be needed to complete the mission, he or she might have to order that the Platoon go back to the INITIATE_MOVEMENT state and collect up those supplies (see APPENDIX C, red line labeled Deficiencies).

Chapter 7

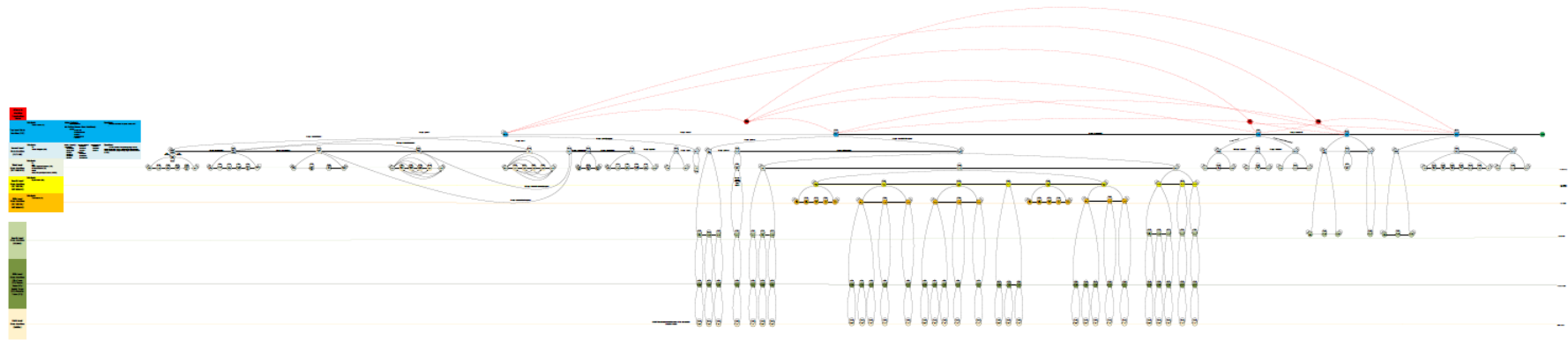
References

Bibliography

- [1] U.S. Army. *Ranger Handbook*. Department of Army. SH 21-76.(Washington D.C.: Government Printing Office, 2011). URL: <http://www.benning.army.mil/infantry/rtb/4thrtb/content/PDF/Handbook.pdf>, last modified February, 2011. Accessed June 1, 2017.
- [2] Shiu-Kai Chin and Susan Older *Certified Security by Design Using Higher Order Logic*. Department of Electrical Engineering and Computer Science Syracuse University, Syracuse, New York 13244, 1.4 edition, Fall 2016.

ASSURANCE FUNDAMENTALS

State Machine Overview

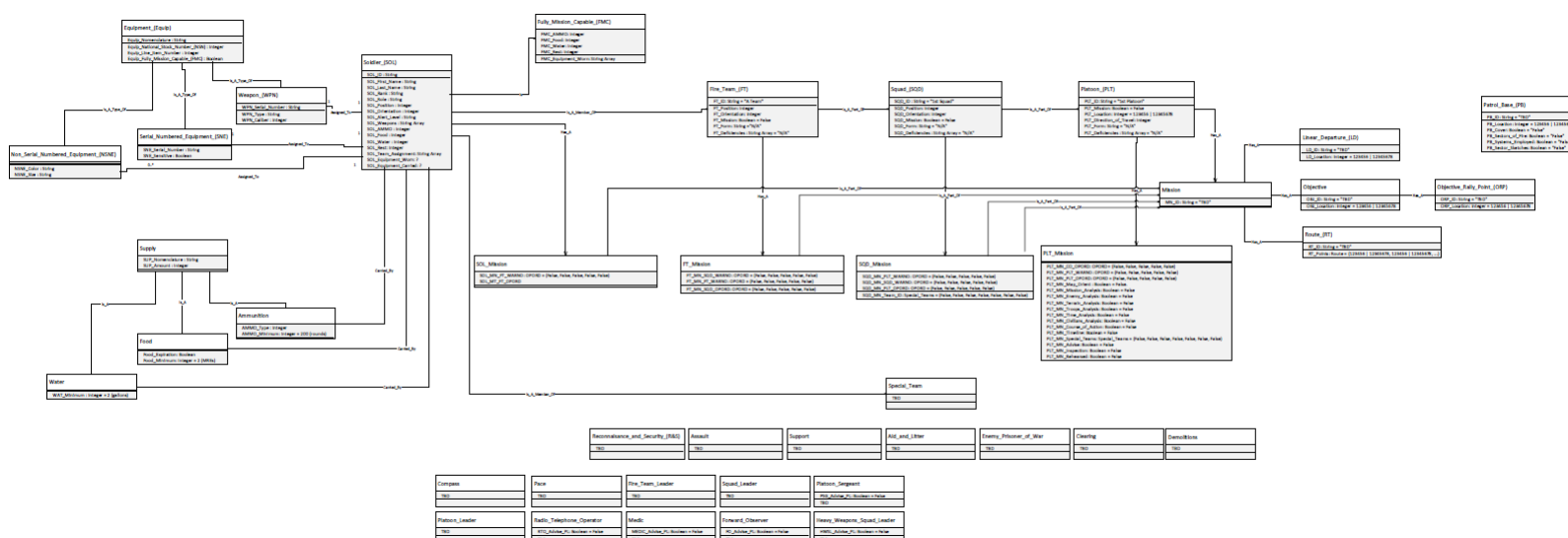


NOTE: A digital copy is available on the Syracuse University Google®Drive in both pdf and visio formats. The visio formatted document contains comments with descriptions, possible HOL implementation, and references to the Ranger Handbook (where applicable).

ASSURANCE FUNDAMENTALS

Appendix B

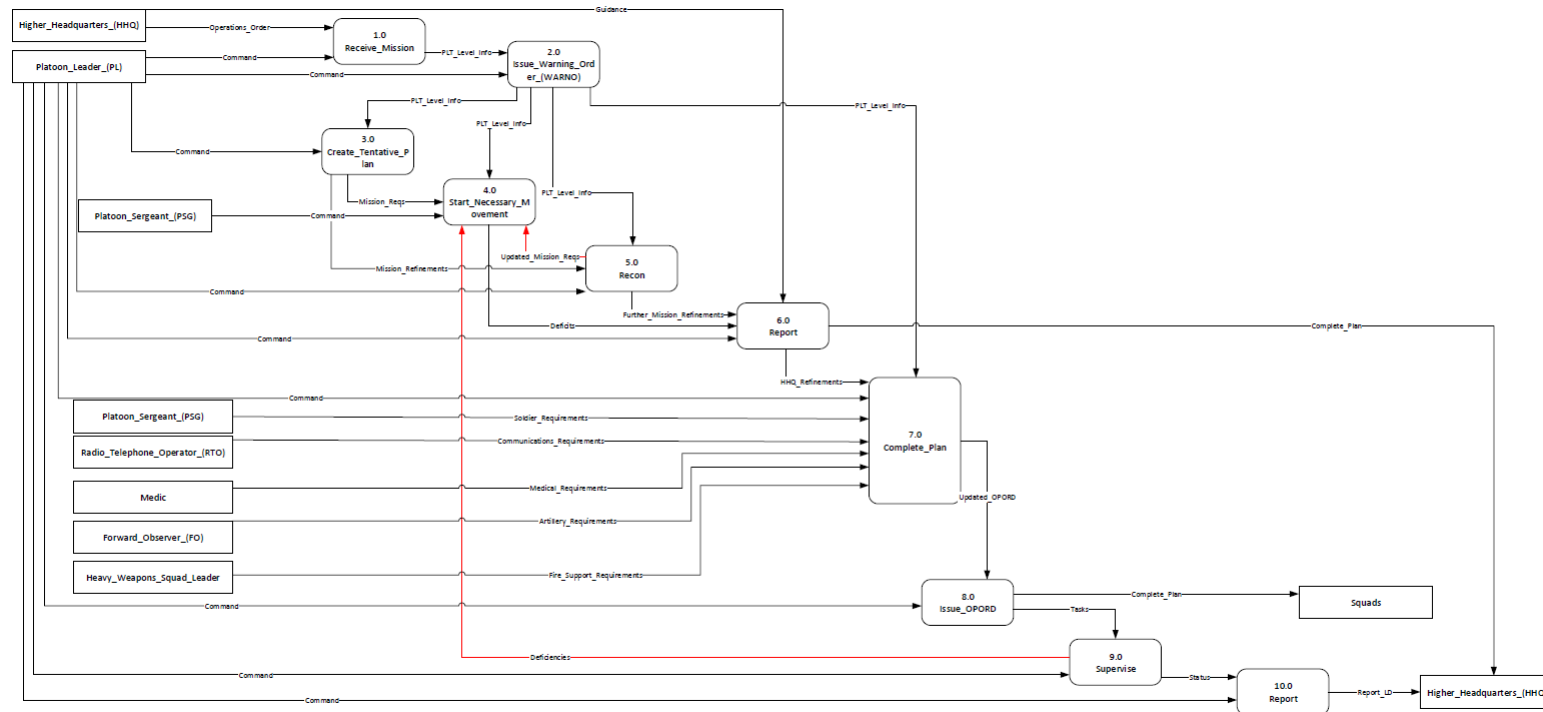
PB UML Class Diagram



NOTE: A digital copy is available on the Syracuse University Google®Drive in both pdf and visio formats. The visio formatted document contains comments with descriptions, possible HOL implementation, and references to the Ranger Handbook (where applicable).

ASSURANCE FUNDAMENTALS

PLAN_PB DFD Level 0



NOTE: A digital copy is available on the Syracuse University Google®Drive in both pdf and visio formats. The visio formatted document contains comments with descriptions, possible HOL implementation, and references to the Ranger Handbook (where applicable).

Appendix D

Next State, Next Output Table

Next State, Next Output Table							
Commands/ input	State						
	next state, next output	crossLD	conductORP	moveToPB	conductPB	completePB	incomplete
	PLAN_PB	MOVE_TO_ORP, MoveToORP					PLAN_PB, PlanPB
	MOVE_TO_ORP		CONDUCT_ORP, ConductORP				MOVE_TO_ORP, MoveToORP
	CONDUCT_ORP			MOVE_TO_PB, MoveToPB			CONDUCT_ORP, ConductORP
	MOVE_TO_PB				CONDUCT_PB, ConductPB		MOVE_TO_PB, MoveToPB
	CONDUCT_PB					COMPLETE_PB, CompletePB	CONDUCT_PB, ConductPB
	COMPLETE_PB						

Appendix E

Source code for ssmPBScript.sml

The following code is from *ssmPBScript.sml*

```
(*****)
(* ssmPBTheory defines the top level state machine for the patrol base. *)
(* Each state, save for the end states, have sub-level state machines, and *)
(* some have sub-sub-level state machines. These are implemented in separate *)
(* theories. *)
(* Author: Lori Pickering in collaboration with Jesse Nathaniel Hall *)
(* Date: 20 June 2017 *)
(*****)
structure ssmPBScript = struct

(* ===== Interactive Mode =====
app load ["TypeBase", "listTheory", "optionTheory",
          "acl_infRules", "aclDrulesTheory", "aclrulesTheory",
          "satListTheory", "ssm11Theory", "ssminfRules",
          "OMNITypeTheory", "PBTypeTheory", "ssmPBTheory"];
open TypeBase listTheory optionTheory
      acl_infRules aclDrulesTheory aclrulesTheory
      satListTheory ssm11Theory ssminfRules
      OMNITypeTheory PBTypeTheory ssmPBTheory
===== end Interactive Mode ===== *)

open HolKernel Parse boolLib bossLib;
open TypeBase listTheory optionTheory
open acl_infRules aclDrulesTheory aclrulesTheory
open satListTheory ssm11Theory ssminfRules
open OMNITypeTheory PBTypeTheory

val _ = new_theory "ssmPB";
(* ----- *)
(* Define the next state function for the state machine. *)
(* ----- *)
val PBNS_def =
Define
'
(* executing *)
(PBNS PLAN_PB      (exec (SLc crossLD))      = MOVE_TO_ORP) /\
(PBNS PLAN_PB      (exec (SLc incomplete))    = PLAN_PB) /\
(PBNS MOVE_TO_ORP  (exec (SLc conductORP))    = CONDUCT_ORP) /\
(PBNS MOVE_TO_ORP  (exec (SLc incomplete))    = MOVE_TO_ORP) /\
(PBNS CONDUCT_ORP  (exec (SLc moveToPB))      = MOVE_TO_PB) /\
(PBNS CONDUCT_ORP  (exec (SLc incomplete))    = CONDUCT_ORP) /\
(PBNS MOVE_TO_PB   (exec (SLc conductPB))     = CONDUCT_PB) /\
```

```

(PBNS MOVE_TO_PB (exec (SLc incomplete)) = MOVE_TO_PB) /\
(PBNS CONDUCT_PB (exec (SLc completePB)) = COMPLETE_PB) /\
(PBNS CONDUCT_PB (exec (SLc incomplete)) = CONDUCT_PB) /\
(* trapping *)
(PBNS (s:slState) (trap (SLc (cmd:slCommand)))) = s) /\
(* discarding *)
(PBNS (s:slState) (discard (SLc (cmd:slCommand)))) = s)‘

(* ----- *)
(* Define next-output function for ssmPB *)
(* ----- *)

val PBOut_def =
Define
‘
  (* executing *)
  (PBOut PLAN_PB (exec (SLc crossLD)) = MoveToORP) /\
  (PBOut PLAN_PB (exec (SLc incomplete)) = PlanPB) /\
  (PBOut MOVE_TO_ORP (exec (SLc conductORP)) = ConductORP) /\
  (PBOut MOVE_TO_ORP (exec (SLc incomplete)) = MoveToORP) /\
  (PBOut CONDUCT_ORP (exec (SLc moveToPB)) = MoveToPB) /\
  (PBOut CONDUCT_ORP (exec (SLc incomplete)) = ConductORP) /\
  (PBOut MOVE_TO_PB (exec (SLc conductPB)) = ConductPB) /\
  (PBOut MOVE_TO_PB (exec (SLc incomplete)) = MoveToPB) /\
  (PBOut CONDUCT_PB (exec (SLc completePB)) = CompletePB) /\
  (PBOut CONDUCT_PB (exec (SLc incomplete)) = ConductPB) /\
  (* trapping *)
  (PBOut (s:slState) (trap (SLc (cmd:slCommand)))) = unauthorized) /\
  (* discarding *)
  (PBOut (s:slState) (discard (SLc (cmd:slCommand)))) = unauthenticated)‘

(* ----- *)
(* Input Authentication *)
(* ----- *)
val authenticationTest_def =
Define
‘(authenticationTest
  ((Name PlatoonLeader) says (prop (cmd:(slCommand command)order))
    :((slCommand command)order , stateRole , 'd, 'e)Form) = T) /\
  (authenticationTest _ = F)‘

(* ----- *)
(* "State Interpretation: this is the trivial assumption TT, as the machine *)
(* state has no influence on access privileges"—Prof. Chin, SM0Script.sml *)
(* ----- *)
val ssmPBStateInterp_def =
Define
‘ssmPBStateInterp (state:slState state) =
  (TT:((slCommand command)order , stateRole , 'd, 'e)Form)‘

(* ----- *)

```

```

(* "A theorem showing commands without a principal are rejected."--Prof *)
(* Chin'e SM0Script.sml *)
(*****)
val authenticationTest_cmd_reject_lemma =
TACPROOF(
  ([],
    '(!cmd. ~(authenticationTest
      ((prop (SOME cmd)):((slCommand command)order ,stateRole , 'd, 'e)Form))) ' '),
  PROVE_TAC[authenticationTest_def])

val _ = save_thm("authenticationTest_cmd_reject_lemma",
  authenticationTest_cmd_reject_lemma)

(* ----- *)
(* securityContext definition: PlatoonLeader authorized on any slCommand *)
(* (defined in PBTypeScript.sml) *)
(* ----- *)
val secContext_def =
Define
'secContext cmd =
  [((Name PlatoonLeader) controls
    (prop (SOME (SLc cmd))) )
    :((slCommand command)order ,stateRole , 'd, 'e)Form] '
(*****)
(* PlatoonLeader is authorized on any slCommand *)
(*****)
val PlatoonLeader_slCommand_lemma =
TACPROOF(
  ([],
    'CFGInterpret ((M:((slCommand command)order , 'b, stateRole , 'd, 'e)Kripke), Oi, Os)
      (CFG
        authenticationTest
        ssmPBStateInterp
        (secContext slCommand)
        (((Name PlatoonLeader) says (prop (SOME (SLc slCommand))))::ins)
        (s:slState state)
        (outs:(slOutput output) list) ) ==>
      ((M, Oi, Os) sat (prop (SOME (SLc slCommand)))) ' '),
  REWRITE_TAC[CFGInterpret_def, secContext_def, ssmPBStateInterp_def, satList_CONS,
    satList_nil, sat_TT] THEN
  PROVE_TAC[Controls])

val _ = save_thm("PlatoonLeader_slCommand_lemma",
  PlatoonLeader_slCommand_lemma)

(* ----- *)
(* exec slCommand occurs if and only if PlatoonLeaders's command is *)
(* authenticated and authorized *)
(* ----- *)
val PlatoonLeader_exec_slCommand_justified_thm =
let
  val th1 =
    ISPECL

```

```
[
  'authenticationTest:((slCommand command)order, stateRole, 'd, 'e)Form -> bool',
  '(secContext slCommand):((slCommand command)order, stateRole, 'd, 'e)Form list',
  'ssmPBStateInterp:(slState state)->
    ((slCommand command)order, stateRole, 'd, 'e)Form',
  'Name PlatoonLeader', 'SLc slCommand:(slCommand command)',
  'ins:((slCommand command)order, stateRole, 'd, 'e)Form list',
  's:(slState state)', 'outs:(slOutput output) list']
TR_exec_cmd_rule
```

in

```
TACPROOF([
  '!(NS :(slState state) -> (slCommand command)trType -> (slState state))
    (Out :(slState state) -> (slCommand command)trType -> (slOutput output))
    (M :(slCommand command)order, 'b, stateRole, 'd, 'e) Kripke)
    (Oi : 'd po)
    (Os : 'e po).
  TR (M, Oi, Os) (exec (SLc slCommand):((slCommand command)trType))
    (CFG
      (authenticationTest
        :((slCommand command)order, stateRole, 'd, 'e) Form -> bool)
      (ssmPBStateInterp :(slState state)
        -> ((slCommand command)order, stateRole, 'd, 'e) Form)
      ((secContext slCommand)
        :((slCommand command)order, stateRole, 'd, 'e)Form list)
      (Name PlatoonLeader says
        (prop (SOME (SLc slCommand):(slCommand command)order):
          ((slCommand command)order, stateRole, 'd, 'e)Form)::
          (ins :((slCommand command)order, stateRole, 'd, 'e)Form list))
        (s:(slState state))
        (outs:(slOutput output) list) )
    (CFG
      (authenticationTest
        :((slCommand command)order, stateRole, 'd, 'e) Form -> bool)
      (ssmPBStateInterp :(slState state)
        -> ((slCommand command)order, stateRole, 'd, 'e) Form)
      ((secContext slCommand)
        :((slCommand command)order, stateRole, 'd, 'e)Form list)
      ins
      (NS
        (s:(slState state))
        (exec (SLc slCommand):((slCommand command)trType)) )
      (Out
        (s:(slState state))
        (exec (SLc slCommand):((slCommand command)trType)):: outs ) )
  <=>
```

```
authenticationTest
  (Name PlatoonLeader says
    (prop (SOME (SLc slCommand):(slCommand command)order):
      ((slCommand command)order, stateRole, 'd, 'e)Form))
  /\
CFGInterpret (M, Oi, Os)
  (CFG
    (authenticationTest
      :((slCommand command)order, stateRole, 'd, 'e) Form -> bool)
```

```

(ssmPBStateInterp :(slState state)
  -> ((slCommand command)order , stateRole , 'd, 'e) Form)
((secContext slCommand)
  :((slCommand command)order , stateRole , 'd, 'e)Form list)
(Name PlatoonLeader says
  (prop (SOME (SLc slCommand):(slCommand command)order):
    ((slCommand command)order , stateRole , 'd, 'e)Form)::
    (ins :((slCommand command)order , stateRole , 'd, 'e)Form list))
(s:(slState state))
(outs:(slOutput output) list) )
/\
(M,Oi,Os) sat
  (prop (SOME (SLc slCommand):(slCommand command)order):
    ((slCommand command)order , stateRole , 'd, 'e)Form)‘‘),
PROVE_TAC[th1 , PlatoonLeader_slCommand_lemma])
end

val _ =
  save_thm("PlatoonLeader_exec_slCommand_justified_thm",
    PlatoonLeader_exec_slCommand_justified_thm)

val _ = export_theory ();
end

```


Appendix F

Source code for PBTypeScript.sml

The following code is from *PBTypeScript.sml*

```
(*****)
(* PBType contains definitions for datatypes that are used in the PB state *)
(* machine ssmPB. *)
(* Author: Lori Pickering in collaboration with Jesse Nathaniel Hall *)
(* Date: 19 June 2017 *)
(*****)
structure PBTypeScript = struct

(* ===== Interactive Mode =====
app load ["TypeBase"]
open TypeBase
===== end Interactive Mode ===== *)

open HolKernel Parse boolLib bossLib;
open TypeBase

val _ = new_theory "PBType";
(* ----- *)
(* slcommand, slState, slOutput, and slLeader *)
(* ----- *)
val _ =
Datatype 'slCommand = crossLD (* Move to MOVE_TO_ORP state *)
    | conductORP
    | moveToPB
    | conductPB
    | completePB
    | incomplete'

val slCommand_distinct_clauses = distinct_of '':slCommand'
val _ = save_thm("slCommand_distinct_clauses",slCommand_distinct_clauses)

val _ =
Datatype 'slState = PLAN_PB
    | MOVE_TO_ORP
    | CONDUCT_ORP
    | MOVE_TO_PB
    | CONDUCT_PB
    | COMPLETE_PB'

val slState_distinct_clauses = distinct_of '':slState'
val _ = save_thm("slState_distinct_clauses",slState_distinct_clauses)
```

```

val _ =
Datatype 'slOutput = PlanPB
          | MoveToORP
          | ConductORP
          | MoveToPB
          | ConductPB
          | CompletePB
          | unAuthenticated
          | unAuthorized '

val slOutput_distinct_clauses = distinct_of '':slOutput ''
val _ = save_thm("slOutput_distinct_clauses",slOutput_distinct_clauses)

val _ =
Datatype 'stateRole = PlatoonLeader '

val _ = export_theory ();

end

```


Appendix G

Symbols, Abbreviations, and Acronyms

<i>DFD</i>	<i>DataFlowDiagram</i>
<i>HHQ</i>	<i>HigherHeadquarters</i>
<i>HOL</i>	<i>HighOrderLogic</i>
<i>LD</i>	<i>LineofDeparture/LinearDeparture</i>
<i>NMC</i>	<i>Non – MissionCapable</i>
<i>OPORD</i>	<i>OperationOrder</i>
<i>PB</i>	<i>PatrolBase</i>
<i>PL</i>	<i>PlatoonLeader</i>
<i>PolyML</i>	<i>PolyMetalanguage</i>
<i>PSG</i>	<i>PlatoonSergeant</i>
<i>TLP</i>	<i>TroopLeadingProcedure</i>
<i>UML</i>	<i>UnifiedModelingLanguage</i>