
AFRL-RI-RS-TR-2019-191

# EMBEDDING VERIFICATION AWARENESS IN SELF-REPAIRING SYSTEMS

THE UNIVERSITY OF TULSA

*OCTOBER 2019*

FINAL TECHNICAL REPORT

***APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED***

STINFO COPY

**AIR FORCE RESEARCH LABORATORY**
**INFORMATION DIRECTORATE**

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

# NOTICE AND SIGNATURE PAGE

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

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nations. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2019-191 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

**/ S /**
WILMAR SIFRE
Work Unit Manager

**/ S /**
BRIAN ROMANO
Acting Technical Advisor
Computing & Communications Division
Information Directorate

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

# REPORT DOCUMENTATION PAGE

*Form Approved*
**OMB No. 0704-0188**



| 1. REPORT DATE *(DD-MM-YYYY)* | 2. REPORT TYPE | 3. DATES COVERED *(From - To)* |
|---|---|---|
| OCTOBER 2019 | FINAL TECHNICAL REPORT | OCT 2016 – APR 2019 |

**4. TITLE AND SUBTITLE**

EMBEDDING VERIFICATION AWARENESS IN SELF-REPAIRING SYSTEMS

**5a. CONTRACT NUMBER**
N/A

**5b. GRANT NUMBER**
FA8750-16-1-0248

**5c. PROGRAM ELEMENT NUMBER**
62788F

**6. AUTHOR(S)**

Rose Gamble

**5d. PROJECT NUMBER**
T2RS

**5e. TASK NUMBER**
EV

**5f. WORK UNIT NUMBER**
AS

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

The University of Tulsa
800 South Tucker Drive
Tulsa, OK 74104

**8. PERFORMING ORGANIZATION REPORT NUMBER**

**9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**

Air Force Research Laboratory/RITA
525 Brooks Road
Rome NY 13441-4505

**10. SPONSOR/MONITOR'S ACRONYM(S)**
AFRL/RI

**11. SPONSOR/MONITOR'S REPORT NUMBER**
AFRL-RI-RS-TR-2019-191

**12. DISTRIBUTION AVAILABILITY STATEMENT**
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09

**13. SUPPLEMENTARY NOTES**

**14. ABSTRACT**

This report details our approach to develop technology that investigates the re-verification potential of critical properties affected by the self-repair plan of an adaptive system within an automated framework that also measures the risk of violated properties should the self-repair take place. From this investigation, we 1) devised a framework that makes component-based systems both architecture and verification aware and enables a dynamic re-verification status assessment in response to a self-repair plan 2) designed a risk assessment mechanism that alerts where a system could enter into a potentially risky state if the self-repair plan causes the violation of a critical requirement, and 3) tested the application of the Genprog self-repair program within the framework. The framework has been embedded into and evaluated using two in-house case studies and a testbed developed to mimic wearable operation and communication.

**15. SUBJECT TERMS**

Self-repair, verification workflow, risk assessment

| 16. SECURITY CLASSIFICATION OF: | | | 17. LIMITATION OF ABSTRACT | 18. NUMBER OF PAGES | 19a. NAME OF RESPONSIBLE PERSON |
|---|---|---|---|---|---|
| a. REPORT | b. ABSTRACT | c. THIS PAGE | | | **WILMAR SIFRE** |
| U | U | U | UU | 132 | 19b. TELEPHONE NUMBER (*Include area code*) **N/A** |

# Table of Contents

## List of Figures

## List of Tables

# 1 Summary

Performing dynamic *compliance guaranteeing processes*, such as verification and security certification of functional and security requirements at run time on an adaptive system is exceedingly complex and stretches current formal model-checking capabilities. We use the generic term *adaptive system* for a system that satisfies any self-* property, such as self-repair or self-protection. We advocate for technology that abstracts key properties from compliance guaranteeing processes performed prior to system deployment, embeds the properties within the adaptive system, and assesses these properties against planned adaptations or repairs at run time. The results of the assessment determine the potential for repeating a compliance guaranteeing process post adaptation and ascertain the resulting risks to the system if the adaptation is performed given a compliance guaranteeing process may be compromised.

The goal of the project is to develop this technology within a framework to determine the potential for reusing compliance guaranteeing processes of critical requirements that may be affected by the adaptive plan (i.e. code repair, functional change, security change) of a self-adaptive system. The framework will translate the reuse potential into a risk measurement that, should the adaptive plan be deployed, it will violate one or more requirements. The risk will be calculated without performing full re-verification or re-certification. We use the term *compliance aware* to represent the dual goals to achieve *verification awareness* – the embedded key properties needed for verification of critical functional requirements – and *certification awareness* the embedded key properties needed for certification of critical security requirements in the form of security controls (NIST, 2013). Thus, the overall research objective is to devise a framework that makes an adaptive system both architecture and compliance aware and enables a run-time risk assessment of an adaptive plan. The techniques developed that underpin the framework can be used to better inform adaptive system design to improve resiliency.

# 2 Introduction

Resilience requires automatically adapting to a situation that impedes a mission. An adaptive software system can monitor itself, analyze a failure occurrence, and recover by altering its state, logic, or architecture. Such autonomous systems rely on continuous monitoring of the system and environment, analyzing performance anomalies, planning the most viable adaptation strategy given the context and available resources, and executing the adaptation on the deployed system. Substantial research exists in self-adaptive systems at all levels, e.g., architectural reconfiguration, interface alterations, and changes to embedded systems, but it focuses primarily on performing a functional or architectural change without disruption (Cheng, 2009), (Lemos, 2013). Once altered, the same requirements compliance guaranteeing processes that were imposed on the originally deployed system should be similarly imposed on the adapted system. Research in self-verification during and subsequent to system adaptation is severely lacking, especially for distributed systems, service-oriented architectures, and embedded systems (Calinescu, 2012), (Tamura, 2013). Software verification and validation, along with security certification, are difficult and tedious processes, demanding well-defined requirements, clear evaluation strategies, and automated methods that should not require more code than the functionality being assessed (Zuo, 2011). This project defines a framework for technology to determine if an adaptation can inhibit the reuse of

the original compliance guaranteeing processes used for verification or certification, where *verification* refers to a formal methods process and *certification* is specific to guaranteeing compliance with security controls such as those in the NIST SP800-53 (NIST, 2013). It extracts critical properties and the flow of assessment or examination from those processes to determine how the adaptation will affect them (Jahan, 2017). The more devastating the effect, the lower the likelihood that a compliance guaranteeing process can be reused. The lower the likelihood of process reuse, the higher the likelihood of compliance violation because rarely can an alternate compliance guaranteeing process be used for a requirement (Marshall, 2018).

There are multiple research issues to be addressed to construct and deploy this technology. One major research issue is to define a strategy to determine if a requirement can be re-guaranteed through verification or certification of the adapted system at the time the adaptation plan is designed and selected. The challenge is to go beyond identifying the critical properties and determining the system's level of compliance through verification or certification to capturing and modeling the compliance guaranteeing process by which verification or certification was initially performed. Formal processes of verification and certification are far too costly in resources to deploy at run time. Though model-checking introduces a level of abstraction into the compliance guaranteeing process, it too is limited in what can be expressed and assessed (Sharifloo, 2015), (Calinescu, 2012), (Cordy, 2013), (Weyns, 2012). To address this research issue, we advocate for modeling the verification and certification processes (i.e. compliance guaranteeing processes) in a way that can be embedded into the system to make it compliance aware. With this awareness, technology can be developed to assess *how it was guaranteed to be compliant* with *how an adaptation affects the reuse of the compliance guaranteeing process*. This report demonstrates an initial framework that embodies this technology.

Another research issue is that metrics need to be defined that associate a risk level with the determination of how an adaptive plan inhibits the reuse of the compliance guaranteeing process. In addition, once a risk factor is calculated, it is essential to understand how that risk propagates throughout the system requirements. This project aligns risk assessment with re-verification and re-certification status assessment. With the framework, the risk metrics are directly tied to the criticality of the affected properties and the extent to which those effects impact reuse of one or more compliance guaranteeing processes.

A third research issue is how to encode compliance awareness and risk assessment into the dynamic adaptive plan analysis at runtime (Abie, 2012), (Almeida, 2011), (Camara, 2013), (Cheng, 2009). The framework developed defines modeling abstractions that express internal processing and external interaction performance parameters and their dependencies. It embeds an executable Colored Petri Net (CPN) (Jensen, 2009), (Jensen, 2015) for each critical requirement that represents the architecture of the compliance guaranteeing process and the properties used for verification and certification. The output of each CPN is aggregated to calculate the risk of an adaptation plan against alternative plans. Overall, the modeling and assessment mechanisms developed for this project will inform the capturing of relevant meta-data at design time to lead to better compliance awareness representations and manipulations in resilient systems.

# 3 Methods, Assumptions, and Procedures

## 3.1 Perspective and Objectives

Not all system requirements can be formally verified, validated, or certified. But many critical properties, especially those that are relevant across the system landscape, would benefit from formal approaches applied to the safety and security of the system, given the environments in which it operates. Autonomous systems must continually comply with certain properties and maintain behavior guarantees. When these systems adapt to a situation, the resulting change should fall within the boundaries of the existing property and behavior guarantees or provide notification as to the extent and risk of the failure as part of compliance awareness (Cotroneo, 2014).

Our perspective is that the compliance guaranteeing process for a critical requirement, such as an invariant (safety property) or behavior expectation over time (liveness property), should be represented in some form to an adaptive system as part of its compliance awareness, which includes architecture awareness and situational awareness. That is, we should be able to capture and model the meta-data that represents the variables, state changes, and communications among systems, components, and internal processes that are needed to guarantee a requirement. Because initially we focused on formal verification of requirements (and later addressed security certification), we call the resulting model a *verification workflow* or VFlow. If an adaptation alters any of the meta-data, then it has the potential to invalidate the verification or certification process, signaling a potential issue with the reuse of the same process within the repaired system. As stated in Section 2, assessing process reuse provides a computationally efficient and effective method of predetermining the potential level of requirements violation within the system at run time. Even if the adaptation is not performed because of the potential for violation, substantive knowledge can be retained to inform the heuristics that help plan the next adaptation.

The main objectives of the project are as follows.

- Develop technology that investigates the reuse of compliance guaranteeing processes.
- Define utility functions that express the risk of inhibiting the reuse of compliance guaranteeing processes due to changing properties on which the processes rely.
- Devise a framework for run-time compliance awareness.
- Derive, model and embed VFlows for computationally efficient assessment of formal verification process reuse.
- Design security assurance cases (SACs) as representative of certification processes for compliance with security controls.
- Devise case studies to evaluate the framework for assessing verification and certification process reuse during run-time adaptation.
- Evaluate the use of GenProg/Darjeeling within the framework,

The approach uses a combination of techniques to embed compliance awareness into the case studies and compute the overall risk of an adaptation plan with respect to functional and security requirements. For functional requirements, techniques include manual proof and the use of the KIV theorem prover (Ernst, 2015) to find verification concerns that are paramount to the proof

process, a Colored Petri Net [(Jensen, 2009), (Jensen, 2015) representation for the VFlow that assesses the potential violations to reusing a compliance guaranteeing process, and an analysis technique to compare adaptations to determine the least risky. For security requirements, manual proof and the KIV theorem prover are shown to be valid techniques. Additional techniques for security certification employ representations of security controls from the NIST SP800-53 (NIST, 2013) as SACs. Achievement weights are deployed at assurance case goals to provide a level of confidence in the claims of compliance with a security control given the effect of an adaptation. Satisficing scores are calculated across a network of related security controls as detailed within the NIST SP800-53 to provide a risk assessment across all requirements when an adaptation is planned. Adaptation operators are defined for SACs to inform the risk assessment.

Four cases studies are defined and used to demonstrate the techniques. The Multi-modal Traveler System (MMTS) is a basic program of a traveler moving within a grid to conserve fuel and avoid enemies. The Smart Inventory Management Systems (SIMS) represents a component-based system that must comply with security controls. The wearable security experimentation testbed (Walter, 2018a) provides a platform to exercise the use of the CPNs. The Cozmo experimentation testbed provides a platform to exercise the use of the SACs.

The assumption is that investigating a means for the dynamic assessment of resilient systems when adapted at run time will inform design practices for adaptive systems. In addition, such technology should reduce the resources needed for re-verification and re-certification by providing automated strategies that distinguish the components and properties directly and indirectly affected by an adaptation from those whose compliance is more likely to remain intact.

## 3.2 General Approach: Embedding Compliance Awareness in Adaptive Systems

We develop a run-time, metadata-driven verification and certification framework within which adaptation plans or patches can be compared to determine the least risky choice at runtime shown in Figure 1. This framework fits within the four phases of the traditional MAPE loop: Monitor, Analyze, Plan and Execute (Kephart, 2003).

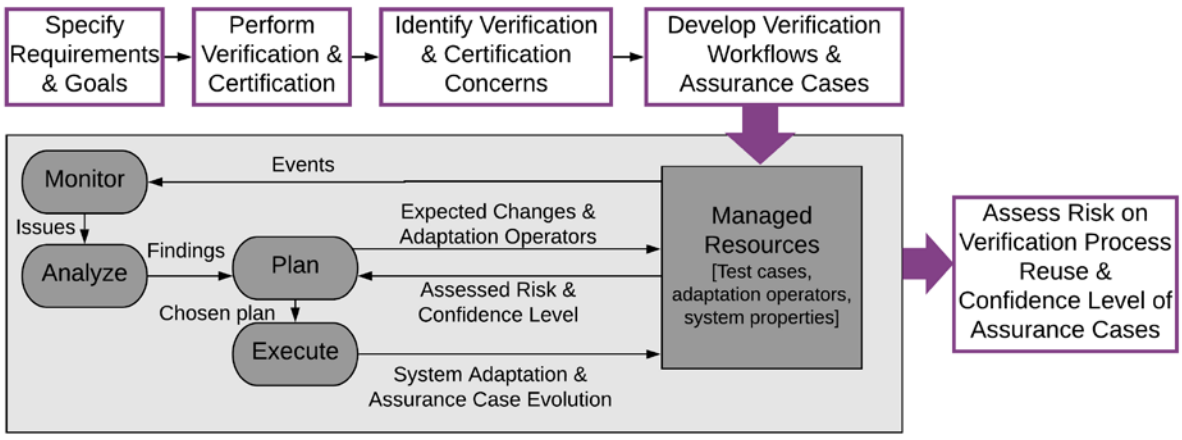



**Figure 1: Metadata based Verification and Certification Framework**

We perform formal proof process and certification of system's requirement compliance for original deployed code of the system and develop argumentation that system maintains a justified confidence level to comply with specified requirements and goals. We construct the structure of argumentation of system's compliance as a form of assurance case by linking the claims of system's intended behavioral goals and evidences of system's compliance with goals as a part of certification. This formal verification and certification process extracts metadata which contain information about potential state variables, their association with system's functions, methods, and components, their condition and impact on original verification and certification process. We model the metadata extracted from verification process of each functional requirement as verification workflow and certification process as assurance cases and deployed them a part of knowledge of MAPE-K loop. The benefit of this approach is having precise knowledge about system and proof process provides assurance about correct behavior of system.

In the monitor phase, the framework allows for examination of states of local and global variables extracted from metadata of formal verification and certification process and events of interest based on situational awareness. Potential problems that can occur due to the events are determined, along with their importance to the system operation, within the analyze phase.

The planning phase is where our effort is mainly concentrated. In this phase, the planner eventually chooses adaptation plan to resolve the problem and creates expected change set for plan. But adaptation may add/delete/modify system's functionality which inhibit to reuse the original verification process and have negative impact on confidence level of certification process. Our approach determines the risk to reuse the original requirement compliance verification process due to adaption changes and recalculate the confidence level of assurance case at runtime. The risk is represented using probability estimates to violate requirement from which we compute the expected utility of each adaptive plan. Probability estimates are calculated based on state variables and their conditions elevated from the original proof process, along with their impact on the proofs and knowledge supplied by the MAPE-K planner. We introduce a utility function that uses probability estimate to perform risk assessment. We employ rule-based adaptation operation to evolve assurance cases with the changes provided by MAPE-K planner and determine impact on state variables calculated from their conditions elevated from the original certification process. These impacts are accumulated and passed through higher level goals. We introduce two metrics: achievement weights and satisficing levels to realize the condition of goals within assurance cases and calculate the confidence level of system's compliance with certification process.

After determining the least risky plan by assessing risk and confidence level, we deploy the modification planned in planning step to system and evolve assurance cases to reflect the changes on model in execution step.

The following sections detail several investigations performed during the contract period to provide a broad understanding of compliance awareness and run-time adaptation risk assessment using the various case studies and platforms. Each investigation works with a set of dedicated requirements and potential adaptations that may be different from those used in another section for the same case study or platform. These differences are due to the focus of the research, approach, implementation, and expected outcome for that investigation. We use the terms self-adaptive system and adaptive system interchangeably, along with adaptation and self-adaptation. Because

we were originally focused on verification of functional requirements, verification awareness appears instead of the more generic term of compliance awareness. Similarly, certification awareness is introduced when examining security controls as a part of the larger effort.

### 3.2.1 MMTS Test Case with Checkpoint Experimentation

Self-adaptive systems must comply with requirements even after adaptation. Compliance to system requirements after adaptation can be assured by performing re-verification process against employed system code. Re-verification requires human intervention which is not feasible for autonomous system and dynamic compliance assessment is very resources intensive. On the other hand, if the adaptation does not impact state variables associated with system requirement, performing dynamic compliance assessment is not necessary. We abstract original verification process and extract metadata representing how the variables, state changes, and communications among systems, components, and internal processes are assessed to establish the compliance status of a requirement. From these metadata, we identify potential state variables associated with system requirement. Identified state variables are organized according to verification process and embed them into system code as checkpoints to realize the state of verification process for each requirement. But adaptation changes functionality, and it has potentiality to effect on state variables associated with original verification process, which subsequently arises deviation to the original verification process. If deviations to the verification process are detected, the adaptation can be considered to have a higher risk of violating critical requirements. We simulate adaptation plan and collect checkpoint logs. If adaptation affects requirements, simulation arises flags even if the issues do not immediately cause a requirement violation. Thus, failure to complete a path through the checkpoints indicates that the verification process may not be reusable.

Our first case study is Multi-Mode Traveler System (MMTS) where a "traveler" moves on a grid with enemies statically placed in random positions. Traveler have to move by avoiding enemies and maintain a fuel threshold range. The traveler's fuel level along with the minimum and maximum fuel levels are set initially and maintained the range. But a move may keep fuel same as prior or increase or decrease a unit from the traveler's reserves depending on which next move is chosen. That is, the traveler's fuel is not associated with its current position, but how it has traveled on the grid. We select three MMTS requirements for verification shown in Table 1. Let $p = (x, y)$ be the traveler's position.

**Table 1: MMTS Requirements and their LTL representation**

| | Statement | LTL Expression | Type |
|---|---|---|---|
| **MR1** | The fuel level must satisfy the defined threshold | $\square(minFuel \leq fuel \leq maxFuel)$ | Invariant |
| **MR2** | The traveler must not enter an enemy position | $\square(p \notin enemyPos)$ | Invariant |
| **MR3** | If the traveler can move, it must move | $\square(canMove(p) \Longrightarrow \diamondsuit[\exists q : q \neq p : moveTo(q)]$ | Progress |

The component-based system architecture including self-adaptive MAPE-K loop as related to the verification of MMTS's three requirements is shown in Table 1. In Figure 2 the MAPE-K loop components are shown as purple ellipse and three system components are shown as light blue boxes. Checkpoints associated with components related with verification process are included within the components and shown as dark blue boxes. The fuel threshold invariant, MR1, and avoiding enemy position invariant, MR2 is checked with respect to traveler's current position using checkpoint C1.A and C1.B at component *getCurrentStatus*. Requirement, MR1 implies that fuel level cannot go below minimum fuel and cannot exceed maximum fuel level. MR1 is an invariant property ($\square$) for MMTS and checked against current fuel level of the traveler. Requirement, MR2 implies that traveler is never in a position where there is an enemy, and it's also an invariant property for MMTS. If $p$ is current position of the traveler on the grid and $enemyPos$ is the set of positions indicating placement of enemies on the grid, then $p$ cannot be an element of $enemyPos$.

If MR1 and MR2 are satisfied, traveler computes available valid moves that traveler can make from current position, and generates set of $validPositions$. This computation is part of progress property, MR3 which implies that if the traveler can move to a new position, then eventually ($\diamondsuit$) the traveler moves to a new (distinct) position. The computation considers the current state so that current position, $p$ is not element of $validPositions$, intersection of $validPositions$ and $enemyPos$ is null and all the element of validPositions satisfy the fuel constraints. Being validPositions nonempty indicates that travel can move and is checked at C1.C. If traveler can move, a next position is chosen randomly from $validPositions$ set in *getNextPosition.* Checkpoint C2.A checks whether next position is chosen or not. Adjusted fuel level for the chosen position will be checked at C3.A again before changing position in *setPosition*.

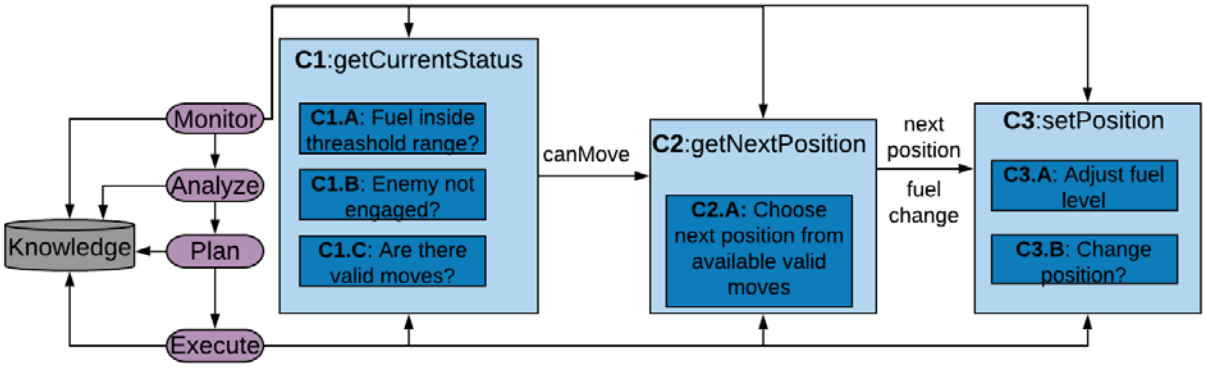



**Figure 2: MMTS architecture Diagram**

### 3.2.1.1 Initial Extraction of Verification Concerns

While verifying the requirements against the system's code, we followed the following method and documented the process to identify verification concerns.

Given that a requirement may be expressed as a safety or progress property, we focus on manual techniques to prove them given an implementation. One type of safety property is an invariant. To prove that properties A and B are invariant, as written in Linear Temporal Logic (LTL) (Lichtenstein, 1985) below

$$\Box (A \wedge B)$$

It must be the case that the properties expressed by A and B hold in all reachable states. That can be restated in terms of Hoare triples to express proofs and the potential for proof reuse (Damiani, 2011) and read as "for all statements s in program F, A and B must be preserved by the execution of s."

$$[\forall s \in F \mid \{A \wedge B\}\ s\ \{A \wedge B\}]$$

This immediately makes A and B verification concerns. Sometimes, other state variables may affect A or B, either by being embedded into a condition that may cause them to be *false* or by having a side effect that can impact their state. Assume that during the verification process, it is determined that

$$C \Rightarrow A$$

Then the state of C may impact the state of A. Thus, C becomes a verification concern.

Progress property proofs can follow a similar verification process use for safety property proofs, except that progress crosses states and, therefore, dependencies exist between verification concerns and must be captured. For example, the progress property below written in LTL, means that "if D becomes *true* in some state, then in the next state D will be changed to D ∧ K."

$$\Box (D \Rightarrow \bigcirc (D \wedge K))$$

This property can be stated in terms of Hoare triples, such as

$$[\forall s \in F \mid \{ D \}\ s\ \{ (D \wedge K) \}]$$

which clarifies that in a state when D holds, the next state will have D and K both *true*, since all statements must ensure this is the case.

With progress properties, the verification concerns still reflect the initial conditions expressed in the Hoare triples, making both D and K verification concerns. Other properties may need to hold to ensure that none of the statements that can execute inhibit K from being true in the next state. Often time functions are assumed to be atomic, meaning they cannot be interrupted. Thus, a statement in F may be a function G. When proving a property, it may be necessary to examine the functions that G comprises resulting in an examination of intermediate (internal) state changes. For example, assume that $G = (f \circ g)$ and the following statements for f and g can be proven.

$$[\forall t \in f \mid \{ D \}\ t\ \{ (J) \}]$$
$$[\forall w \in g \mid \{ J \}\ w\ \{ (D \wedge K) \}]$$

Since the next complete state change occurs after G, the progress property is not violated. But if an adaptive plan did not abide by the atomicity assumptions, then the intermediate state transition could be problematic. Therefore, these properties, such as J above, are also captured as verification concerns.

In the MMTS case study, we have manually proven the requirements against Java code and extracted the verification concerns from those proofs. To verify a goal Hoare triple, we analyze the code to identify subgoal lemmas, usually in the form of additional Hoare triples that imply the goal. The fact that the identified lemmas imply the Hoare triple is dependent on the analyzed code rather than being tautological, so we reify it as an additional lemma. We generate three types of lemmas. The first two are Hoare triples and implications from which we extract verification concerns. The third lemma type states that a given method is a pure function and does not generate verification concerns. Since Hoare triple lemmas specify Boolean preconditions and postconditions, they formulate the verification concerns. For implication lemmas, extraction of verification concerns is less straightforward. We identify them based on the argument used to verify an implication lemma.

We assume the correctness of certain methods that are not part of the main controlling function. These methods are typically “getter” and “setter” functions, such as *traveler.getFuel* or *traveler.setMaxFuel*, which function as expected. We also assume the correctness of standard Java library methods. With these assumptions, our verification process focuses on defining subgoals involving methods that are part of the main controller.

Even with these assumptions, we found that similar verification concerns can emerge from the proof lemmas, leading to the creation of an excessive number of checkpoints. Our objective is to maintain a high level of abstraction and reduce overhead. Therefore, we filter the lemmas to obtain a smaller subset of verification concerns from which checkpoints are generated. Our current approach identifies a *domain of influence* for each method in the main controller, consisting of the set of state variables the method may modify. We assume this becomes part of the documentation associated with the verification process. We filter out any Hoare triple lemmas for which the postcondition does not contain state variables in the method’s domain of influence. For example, if

*computeResult* does not have *numFailures* in its domain of influence, we would not generate any verification concerns from the Hoare triple.

$\{numFailures = 0\}\ computeResult(\ )\ \{numFailures = 0\}$

One disadvantage is that proof violations may go unnoticed if an adaptation modifies a method to affect state variables outside its domain of influence. This scenario will be addressed by future work.

As requirement MR3 relies on requirement MR1 and MR2 as subgoals, so proof process of MR3 contains proofs of MR1 and MR2. Here, we provide proof discussion only for MR3. We specified MR3 as a form of following Hoare triple.

**L1**: $\forall x, y : \{pre(x,y)\}\ update(state, \dots)\ \{post(x,y)\}$

where, $pre(x,y) \triangleq validPositions \neq \emptyset \wedge p = (x,y)$

and $post(x,y) \triangleq p \neq (x,y)$

L1 means that, whenever $validPositions$ is nonempty, the traveler eventually move to new position and update will be set at setPosition. To satisfy the Hoare triple, $validPositions \neq \emptyset$, and $newPos$ must not equal $(x,y)$ at that point need to be satisfied, which relies on following subgoal lemmas.

**L2**: $\{fuelThresh\}\ update(state, \dots)\ \{fuelThresh\}$,

where $fuelThresh \triangleq minFuel \leq fuel \leq maxFuel$

**L3**: {*safePos*} *update*(*state*, …) {*safePos*}

where *safePos* ≜ $p \notin$ *enemyPos*

**L4**: *CheckEnemy* returns false if the traveler is not in an enemy position

{*safePos*} *CheckEnemy*(*state*, …) {*retVal* = *false*}

**L5**: *computeValidPositions* is pure.

**L6**: *CheckEnemy* does not modify the position

$\forall q$ : {$p = q$} *CheckEnemy*(*state*, …) {$p = q$}

**L7**: *computeValidPositions* never returns the current position

$\forall q$ : {$p = q$} *computeValidPositions*(*state*) {$q \notin$ *retVal*}

**L8**: *generateRandomNextMove* returns one of the positions passed to it

$\forall s$ : {$s \neq \emptyset$} *generateRandomNextMove*(*s*) {*retVal* $\in s$}

**L9**: (**L2** $\wedge$ **L3** $\wedge$ … $\wedge$ **L8**) $\Rightarrow$ **L1**

Verification process of three requirements of MMTS generates verification concerns which needs to be checked by analyzing checkpoint logs to realize adaptation impact on reusing verification process. Verification concerns associated with two invariant requirements of MMTS, MR1 and MR2 are *fuel*, *minFuel*, *maxFuel*, *position*, and *enemyPos*. But verification concerns associated with MR3 are *validPositions*, *newPos* along with the above-mentioned verification concerns. The proof lemmas that generate verification concerns and associated conditions set as checkpoints are shown in Table 2.

**Table 2: Checkpoint conditions derived from lemmas**

| Lemma | Checkpoint conditions |
|---|---|
| **L1** | $validPositions \neq \emptyset$; $p \neq (x, y)$ |
| **L2** | $minFuel \leq fuel \leq maxFuel$ |
| **L3** | $p \notin enemyPos$ |
| **L4** | $p \notin enemyPos$; $enemyCheckPassed = true$ |
| **L7** | $p \notin validPositions$ |
| **L8** | $validPositions \neq \emptyset$; $newPos \in validPositions$ |
| **L9** | $fuelCheckPassed = true$; $enemyCheckPassed = true$; $canMove = true$ |

### 3.2.1.2 Embedding Verification Concerns as Checkpoints for MMTS

To realize adaptation plan's potentiality to invalidate original verification process, we generate a restricted Petri Net for each requirement using ProM by embedding checkpoints into the code that relate to

- Identified verification concerns
- Their use in the code that affected the requirement proof, which may require multiple checkpoints for the same verification concern
- Progress property dependencies among verification concerns.

We organize identified verification concerns according to the verification process and embed associated checkpoints within underlying code as a part of verification workflow. Figure 3 shows a checkpoint for fuel consistency associated with MR1 at line 34-38 and checkpoint for current position is not at enemy position associated with MR2 at line 39-40. Other checkpoints are also inserted within the code and this insertion of checkpoints has no interference on the original functionality of MMTS. Flag has been raised if violation has been detected. We also include a log generation method which collects the logs of checkpoints, assuming that logging has not interfered with system operation and also adaptation doesn't cause stop logging. Analysis of the checkpoint for the simulation of adaptation plan will pinpoint where potential deviations to the original verification process may occur because of the adaptation. If deviations to the verification process are detected, the adaptation can be considered to have a higher risk of violating critical requirements.

```
21  public void update(State state, LogGeneration logger, int caseID)
22  {
23      if(getCurrentStatus(state,logger,caseID)) {
24          Position newPos = getNextPosition(state,logger,caseID);
25          if(newPos!=null){
26              setPosition(state,newPos,logger,caseID);
27          }
28      }
29  }
30
31  private boolean getCurrentStatus(State state, LogGeneration logger, int caseID)
32  {
33      boolean proceed=false;
34      boolean fuelCheckPassed = state.getTraveler().Check_FuelConsistency();
35
36      logger.GenerateLog( _methodName: "1st Check Fuel", caseID,  _result: ""+fuelCheckPassed);
37
38      if (fuelCheckPassed) {
39          boolean enemyCheckPassed = !CheckEnemy(state);
40          logger.GenerateLog( _methodName: "1st Check Enemy", caseID,  _result: "" + enemyCheckPassed)
41          if (enemyCheckPassed) {proceed=true;}
42          else{
43              proceed=false;
44              logger.GenerateLog( _methodName: "Flag", caseID,  _result: ""+"false");
45              state.getTraveler().Set_Traveler_InActive();
46          }
47      } else{
48          proceed=false;
49          logger.GenerateLog( _methodName: "Flag", caseID,  _result: ""+"false");
50          state.getTraveler().Set_Traveler_InActive();
51      }
52      return proceed;
53  }
54
55  private Position getNextPosition(State state, LogGeneration logger, int caseID)
56  {
57      Position initPos = state.getTraveler().Get_Position();
58      Position newPos = state.getTraveler().Get_Position();
59
60      Set<Position> validPositions = computeValidMoves(state);
61      boolean checkCurrentPositionFromValidMoves=checkCurrentPositionFromValidMoves(validPositions,initPos);
62
63      if(!checkCurrentPositionFromValidMoves) {
64          logger.GenerateLog( _methodName: "Current Position Check", caseID,  _result: "" + "true");
65          boolean canMove = !validPositions.isEmpty();
66          if (canMove) {
67              logger.GenerateLog( _methodName: "Valid Positions NonEmpty", caseID,  _result: "" + "true");
68              newPos = generateRandomNextMove(validPositions);
69              boolean checkNewPositionFromValidMoves = checkNewPositionFromValidMoves(validPositions, newPos);
70              if (checkNewPositionFromValidMoves) {
71                  logger.GenerateLog( _methodName: "New Position Check", caseID,  _result: "" + "true");}
72              else{
73                  logger.GenerateLog( _methodName: "Flag", caseID,  _result: ""+"false");
74                  state.getTraveler().Set_Traveler_Active();
75              }
76          }
77          else{
78              newPos=null;
79              logger.GenerateLog( _methodName: "Valid Positions Empty", caseID,  _result: ""+"true");
80              state.getTraveler().Set_Traveler_Active();}
81      }else{
82          newPos=null;
83          logger.GenerateLog( _methodName: "Flag", caseID,  _result: ""+"false");
84          state.getTraveler().Set_Traveler_Active();}
85      return newPos;
86  }
87
88  private void setPosition(State state, Position newPos, LogGeneration logger, int caseID) {
89      Position initPos = state.getTraveler().Get_Position();
90      int changeFuel = getChangeFuel(state.getTraveler().Get_Position(), newPos);
91      state.getTraveler().Set_ChangeFuel(changeFuel);
92
93      boolean logFuel = state.getTraveler().Check_FuelConsistency();
94      logger.GenerateLog( _methodName: "2nd Check Fuel", caseID,  _result: ""+logFuel);
95
96      if(logFuel){
97          state.getTraveler().Set_Position(newPos);
98          boolean logPosCheck = !initPos.equals(state.getTraveler().Get_Position());
99          logger.GenerateLog( _methodName: "Position Changed", caseID,  _result: ""+logPosCheck);
100         if (!logPosCheck){logger.GenerateLog( _methodName: "Flag", caseID,  _result: ""+false);}
101         boolean logEnemyCheck = !CheckEnemy(state);
102         logger.GenerateLog( _methodName: "2nd Check Enemy", caseID,  _result: ""+logEnemyCheck);
103         if(logEnemyCheck){state.getTraveler().Set_Traveler_Active();}
104         else{
105             logger.GenerateLog( _methodName: "Flag", caseID,  _result: ""+"false");
106             state.getTraveler().Set_Traveler_InActive();
107         }
108     }
109     else{
110         logger.GenerateLog( _methodName: "Flag", caseID,  _result: ""+"false");
111         state.getTraveler().Set_Traveler_InActive();}
112 }
```

**Figure 3: MMTS Code with Logging of Verification Concerns as Checkpoints**

Figure 4 shows the ProM visualization of a successful, non-adaptive execution of MMTS with 100 moves. Figure 4 shows that for given 100 moves, there are 100 paths that proceed through all the checkpoints and has no deviation from verification workflow which indicates the requirements are not violated.

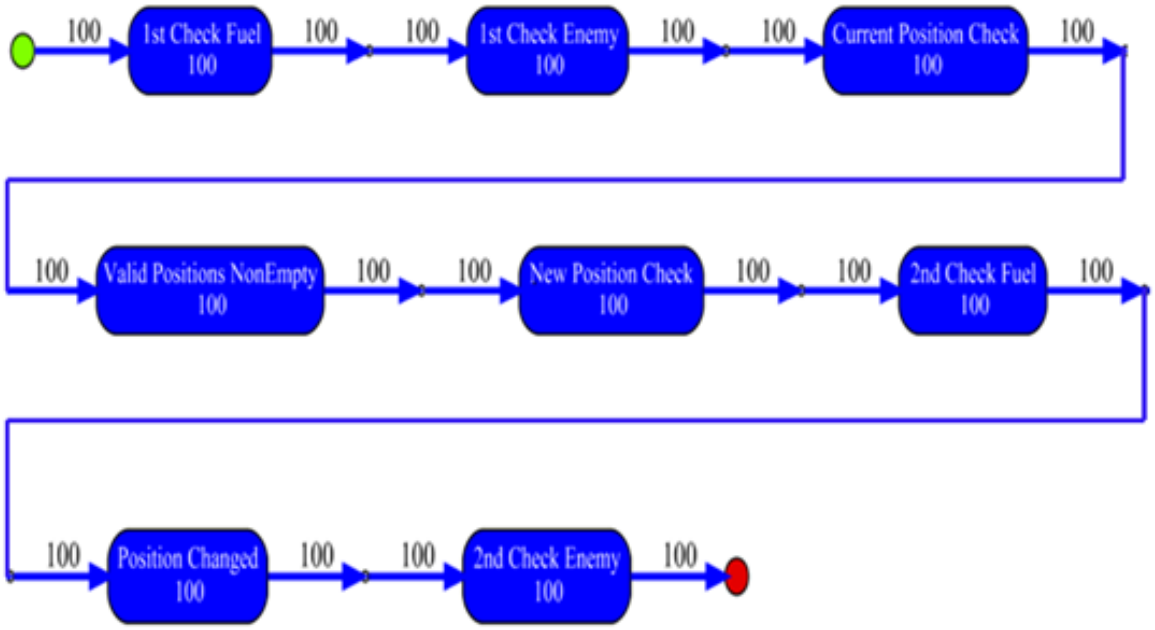



**Figure 4: Visualization of a successful, non-adaptive execution of MMTS**

### 3.2.1.3 Analyzing Adaptations to the MMTS

We simulate two adaptation plans that could occur if the Monitor observes fuel level and "fuel line anomaly" has been detected at Analyze process.

A1. Reduce the maximum fuel value so that the traveler maintains a lower overall fuel value.
A2. Force the traveler to stop for a number of "moves" for repair.

Figure 5 shows the ProM visualization of Adaptation A1, where a flag has been raised at 1st Check Fuel checkpoint. Investigation on this flag issue indicates that the current fuel level exceed to the maxFuel value as maxFuel is reduced by half of its prior value shown in Figure 6 (line 96).

Therefore, when the maxFuel value was decreased the verification processes for MR1 and MR3 were impacted because the precondition for the fuel consistency is not satisfied.

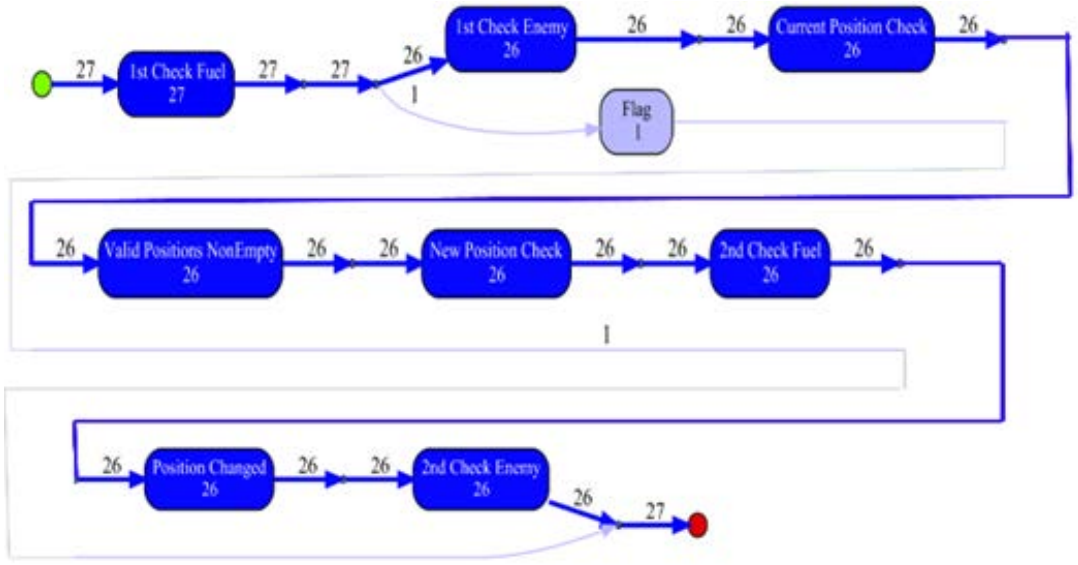



**Figure 5: ProM Visualization of the Verification Workflow for Plan A1**

```
94      private void reduceMaxFuel(State state)
95      {
96          state.getTraveler().Set_MaxFuel( _max_fuel: state.getTraveler().Get_MaxFuel() / 2);
97          hasReducedMaxFuel = true;
98      }
```

**Figure 6: Adaptation A1 implementation Code for MMTS**

Figure 7 is ProM visualization for adaptation plan A2. Adaptation A2 allows traveler to stop for 5 moves to repair by forcefully making $validPositions$ set empty shown in line 96 in Figure 8. Figure 7 shows that 5 paths during the 100 iterations found valid positions to be empty. Those 5 paths were the repair paths and the verification process for MR3, as well as for MR1 and MR2, was not impacted because according to MR3, the traveler is only required to move if $validPositions$ is non-empty.

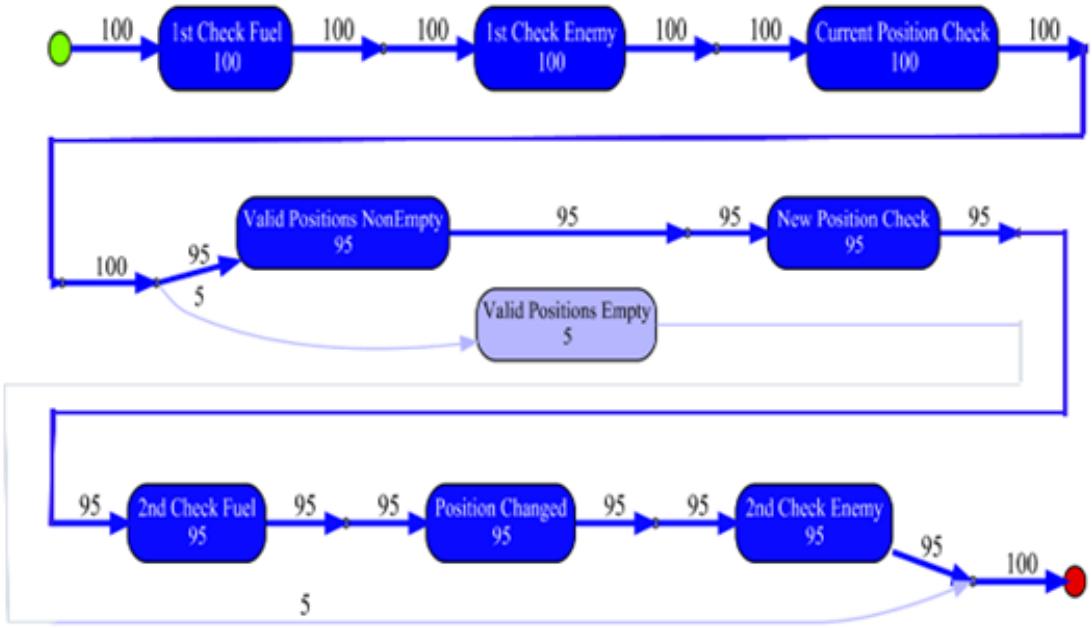



**Figure 7: ProM Visualization of the Verification Workflow for Plan A2**

```
93  @Override
94      public Set<Position> computeValidMoves(State state)
95      {
96          if(repMove>0){
97              repMove--;
98              return Collections.emptySet();
99          }
100
101         else{
102             Position currPos = state.getTraveler().Get_Position();
103             int minFuel = state.getTraveler().Get_MinFuel();
104             int maxFuel = state.getTraveler().Get_MaxFuel();
105             int currFuel = state.getTraveler().Get_Fuel();
106
107             Set<Position> moves = new HashSet<>(Arrays.asList(
                        new Position( x currPos.Get_X()+0,  y currPos.Get_Y()+1),

                                .....
                                .....
                                .....

127
128             return Collections.unmodifiableSet(moves);
129     }
```

**Figure 8: Adaptation A2 implementation Code for MMTS**

### 3.2.2 SIMS Test Case with Checkpoint Experimentation

Our next case study is Smart Inventory Management System (SIMS) which tracks inventory's stock condition using a pressure sensor and adjusts a pressure threshold depending on the inventory movement. SIMS architecture, implemented as multiple threads in Java. SIMS has three components with local MAPE-K control loops: Measure reads the pressure sensor and sends data to Process, Process adjusts the threshold calculated from those readings and sends new threshold value to Measure, and Audit stores audit records from Measure and Process for security certification. The Measure and Process components has capability to support Audit and accountability policy as security requirement. This capability includes sensing auditable events, generating audit records, and sending audit records on a channel to be received by the Audit component and appropriately placed in the audit trail.

The SIMS case study will serve multiple purposes. First, it will provide a mechanism to test the efficacy of the methodologies crafted for the first case study. Second, it will allow us to explore security certification issues. Finally, in its final form, it will be a distributed service-oriented architecture that broadens the scope of the systems we are analyzing. Currently, it is fully simulated in Java so that we can work with the findings from the 1st case study, MMTS. During this simulation, we refined the architecture and processes, which are shown in Figure 9. The simulation of the distributed system includes:

- A MAPE loop component that examines the information stored in the audit trail, as well as other information, to signify if an event of interest has occurred that requires an adaptive plan to be assessed. Adaptation can be imposed on any of the components
- A connection service that checks that the network connection is active and not compromised
- A measurement service that reads the state of a pressure sensor measuring the amount of inventory
- A cloud service that examine the sensor data over time, forecasts the best inventory weight, and changes the threshold values so that the sensor reports adhere to the forecast
- An audit component that house an audit trail of data collected from the system components.

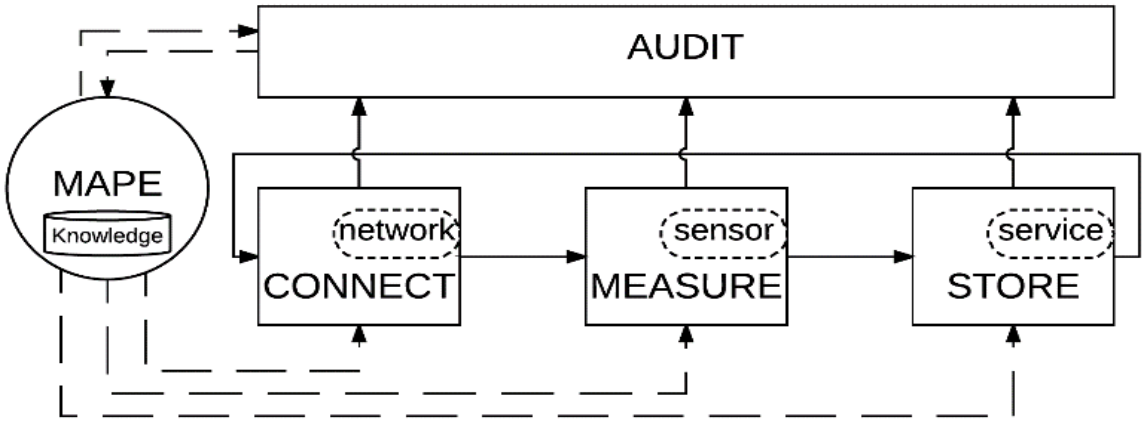



**Figure 9: Architecture for the Smart Inventory Management System**

The base level requirements for SIMS appear in Table 3, where they are associated with their formal LTL expression. In this case, we introduce two versions of the same requirement, SR2a and SR2b, where SR2a is much more restrictive than SR2b. The reason for examining these two versions is to show how flexibility in the requirements may allow for safe adaptations that can be re-verified, whereas more restrictive requirements result in a higher risk.

**Table 3: SIMS Requirements and their LTL representation**

| | Statement | LTL Expression | Type |
|---|---|---|---|
| **SR1** | Sensor signals high unless the pressure is below a predefined threshold | $\Box$(signal = 1 $\Leftrightarrow$ (pressure $\geq$ threshold)) | Invariant |
| **SR2a** | Sensor status is appended to the cloud status on the next step after the measurement when the network connection is good | $\Box$(connection = *true* $\wedge$ threshold = t $\wedge$ pressure = p $\wedge$ cloudDB = c $\Rightarrow$ ($\bigcirc$cloudDB = c ^ (t,p)) | Progress |
| **SR2b** | Sensor eventually sends its status to the cloud when the connection is good | $\Box$(connection = *true* $\wedge$ threshold = t $\wedge$ pressure = p $\Rightarrow$ ($\Diamond$(t,p) $\in$ cloudDB) | Progress |

We have implemented a simulation of the hardware/software aspects of the SIMS system in Java and proven that the code meets the requirements in Table 3. While verifying the requirements against the SIMS Java code, we followed the similar methods used for the MMTS case study. While doing so, we documented the process to identify verification concerns. From the verification

process for SR1, the verification concerns that emerged were *signal* (is it measurable), pressure and threshold. Since signal subsumes the state of *pressure* and *threshold* (i.e., if it is measurable, the values can be ascertained), we retain *signal* as the only verification concern for SR1. For the proof of requirement SR2a and SR2b, the verification concerns identified were *signal*, *pressure*, *threshold*, *cloud availability*, and *connection status*.

#### 3.2.2.1 Embedding Verification Concerns as Checkpoints for SIMS

Figure 10 shows a checkpoint for the first of four signal checks at lines 14-19. This code can be inserted without any interference on the original functionality of SIMS.

```
11  public class BaseComponent implements Component{
12      public void update(Sensor sensor, Connection connection, Cloud cloud,LocalDB localDB,AuditI
13      {
14          boolean signalCheck1 = (checkSensorHigh(sensor,caseId,logger) == (sensor.getPressure()
15          logger.GenerateLog( _methodName: "Signal Check 1", caseId);
16          if (!signalCheck1)
17          {
18              logger.GenerateLog( _methodName: "Signal Check 1 Failed", caseId);
19          }
20
21          String initialThreshold = Double.toString(sensor.getThresholdValue());
22          String initialPressure = Double.toString(sensor.getPressure());
23          Path initialCloudDB = null;
```

**Figure 10: Sample checkpoints within SIMS case study code**

Figure 11 shows a static image of the executed base code with the embedded checkpoints for SIMS after ProM analyzed the log files for 100 paths based on the verification process for SR2a and SR2b which were similar enough to combine into a single process. In addition, the verification concerns for SR2a and SR2b subsume those of SR1, so SR1's checkpoints are used as well. Normally, ProM shows each path using a token that flows through the places indicated by blue ellipses. Each place represents a verification concern that produced an embedded checkpoint. In Figure 11, 85 paths have the signal with the pressure meeting the threshold (Signal Check 2) and 15 paths have the signal with the pressure below the expected threshold (Signal Check 3) likely due to changes by the cloud feedback. Signal Check 1, as seen in Figure 10, ensures that at the start of the measuring the sensor is functioning properly (from requirement R1), while Signal Check 4 determines that at the end of the measurement, the sensor is still functioning properly. These two checks are necessary to ensure the invariant is not violated during the entire process. Also in Figure 11 are the dedicated checkpoints for the cloud database contents for SR2a and SR2b. The difference between the two requirements is that SR2a requires the sensor output to be appended to the cloud database on the next step (as denoted by ○) after the measurement is taken and provided the network connection is good. But SR2b requires only that eventually (as denoted by ◇) the measurement appears in the cloud database given a good network connection.

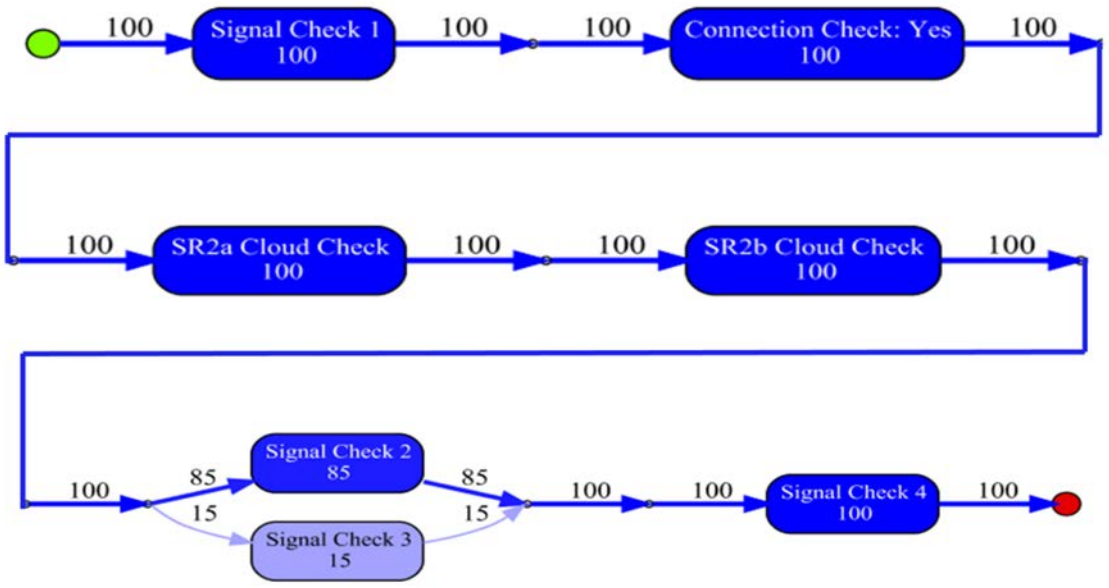



**Figure 11: ProM visualization of 100 paths through SIMS with embedded checkpoints**

### 3.2.2.2 Analyzing Adaptions to the SIMS

We have configured four adaptive plans for the SIMS case study given a sensor problem or a network problem. These self-adaptations will allow initial investigation into comparison approaches and risk assessment. They are:

- **Plan-1**: Replace the sensor, when the primary sensor is not sending signals
- **Plan-2**: Store data into local database when the connection is lost and then send to cloud once connection is re-established
- **Plan-3**: Ignore sensor data and always send low signal
- **Plan-4**: Throw away data and don't store it any database.

Given the embedded checkpoints, we deploy a simulation of the plans with SIMS to see the logged results. Currently, ProM provides a visual comparison of the how the checkpoints were reached if 100 measurements were attempted. Figure 12 illustrates the checkpoint logs when Plan-2 is executed. We use this plan because it also shows the differences between requirement R2a and R2b.

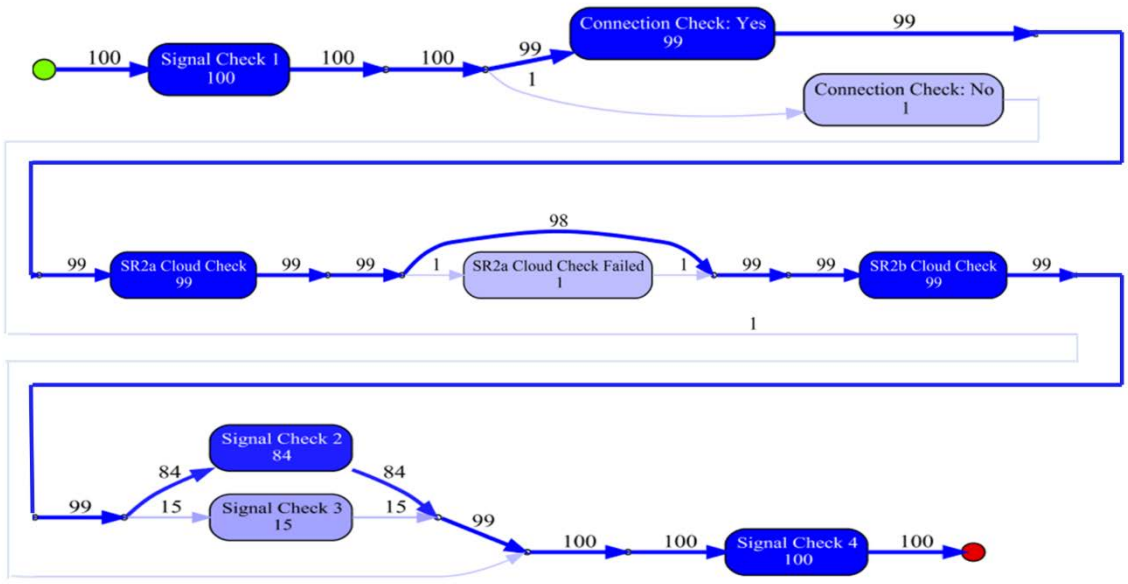



**Figure 12: ProM visualization of checkpoint logs for Plan-2**

When Plan-2 is execution, Figure 12 shows that SR2a's verification process may be inhibited because the associated cloud check fails. The failure does not occur because the adaptive plan forces the sensor output to be stored locally when the network connection is bad. Rather the failure occurs because when the network connection is restored (i.e. good), the local database is pushed to the cloud. But the requirement say that the cloud must be its previous state plus the latest measurement. However, its previous state does not include the local database. Thus, requirement SR2a is restrictive in its statement such that its verification process uses the "before" state of the cloud database to verify the "after" state of the cloud database when the most recent measurement is added.

Figure 12 also shows that SR2b's verification process is unaffected by Plan-2. This is because the verification process only examines the cloud database after the connection is deemed good to find that the latest sensor measurement is available. So, when the local database is pushed to the cloud, as long as it contains the latest measurement, the verification process is not affected. This process, along with the embedded checkpoints, suggest that requirements that have some flexibility and less details within their verification process can withstand adaptation better then restrictively stated requirements that may have the same semantics.

### 3.2.3 Building in Risk Assessment

In this section, we extend the lessons learned in the embedded checkpoint work to designing the overall framework so that it can incorporate adaptation plan risk assessment with respect to verification and certification constraints provided at design time. The extension includes (1) the introduction of the verification workflow specified as a Colored Petri Net, (2) the expression of an initial utility function for risk assessment and plan comparison, and (3) an example of risk analysis using the MMTS case study and additional adaptive plans.

#### 3.2.3.1 Specifying the Verification Workflow as a Colored Petri Net

Each verified requirement has a related verification workflow or *VFlow* that models the verification process using the following information:

- **Verification concerns** as derived from the verification process.
- **Components or processes in the system architecture** where the verification concerns were of interest, which may involve multiple components and processes.
- **Conditions related to the impact or prominence of verification concerns in the verification process**. These conditions describe change types or ranges that fall in one of three sets related to expected impact: devastating, worrisome, or unconcerned.

The system architecture, as part of the VFlow construction, underlies the verification process and self-adaptive system. Components are generally examined independently for compliance and, then, as part of the larger system architecture through their interfaces. Within the components are the state variables, functions, and dependencies that the verification process must also examine, in the form of lemma proofs. Thus, the VFlow should represent the proper granularity of the architecture description that best fits the verification process perspective and flow.

#### 3.2.3.2 Constructing the Colored Petri Net

A Petri Net is a bipartite graph that makes available mathematical analyses and decision processes to a wide range of applications by allowing the expression of system functionalities and architectures. Colored Petri Nets (CPNs) introduce additional functionality through distinguishing features among tokens traversing the network and complex processing by the transitions that dictate token paths. Their flexibility in modeling architectures and component processing, along with the availability of automated tools to simulate them, make them an appropriate representation for VFlows used to assess the risk of an adaptive plan.

The current representation of a VFlow models the major components important to the verification processes as places in the CPN. A generic VFlow is shown in Figure 13. Tokens traverse the places using transitions. In a CPN, transitions can perform complex processing based on embedded, immutable data structures.

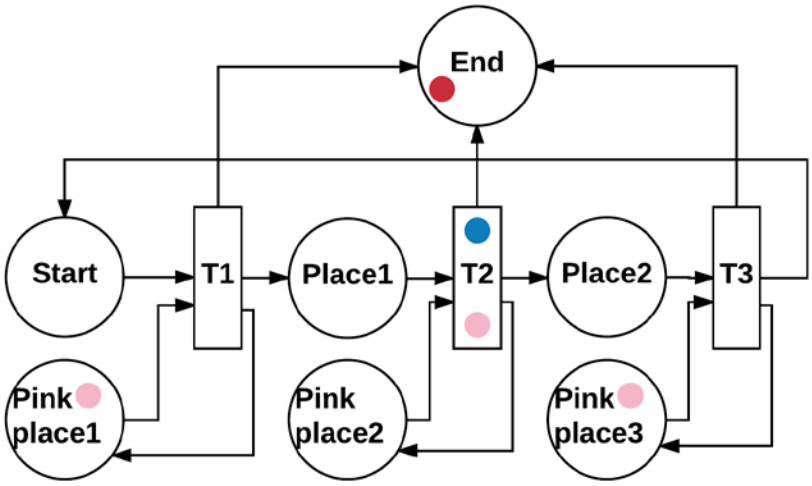



**Figure 13: Generic VFlow represented as a CPN**

There are three token colors used for the initial VFlow expression in a CPN. **Pink tokens** hold the adaptive plan's *change set*. The planning portion of the MAPE-K loop formulates and configures potential plans to be risk-assessed. The assumption is that the change sets embody information related to (1) what parameters are changed, (2) how, in general, they are changed, (3) what major components are affected, and (4) what the planner believes the impact of that change to be overall. The change impact can be determined by the planner based on accumulated knowledge through techniques such as reasoning over the success or failure history of changes, machine learning outcomes based on adaptive plans shared across related information systems, and partial plan simulations. Details on the calculation of the change impact will be discussed as part of the risk utility function. The structure of the pink token appears below where each element of the change set has a unique ID, the verification concern affected (VC), the type of effect to the verification concern (condition), and its planner-calculated change impact ($\hat{p}$).

$$\mathbf{t_{pink}} = ((\mathrm{ID}_1, \mathrm{VC}_1, \mathrm{condition}_1, \hat{p}_1), \ldots, (\mathrm{ID}_n, \mathrm{VC}_n, \mathrm{condition}_n, \hat{p}_n))$$

The **blue token** traverses the CPN looking for verification concerns in the verification process that may be affected by the change set. These are called *conflicts*. This token holds the outcomes of multiple affected verification concerns as well as change sets represented by pink tokens. Thus, checks against verification concerns can be performed at all places in the CPN, regardless of where the pink token designates the change will occur. As seen in the construct below, the blue token tracks the places traversed (visited) so that it ensures that every verification concern in the change set is examined at every place. The tracking also determines when the blue token's cycle is complete and it can be absorbed to terminate the CPN.

$\mathbf{t_{blue}}$ = (visited, vcMatches, vcConflicts, dependencies, conflictCount, tokenCount)

The set vcMatches in the blue token accumulates the conflicts found as a set of tuples of the form (IDconflict, vcInfo, conflictPlace, $\hat{p}$). IDconflict is a unique identifier of the conflict based

on the blue token's conflictCount. This identifier is used for the alert and may be repeated across alerts if more than one place is affected. vcInfo holds a record of the verification concern affected and the impact determined by the transition based on the change set's condition for that verification concern. Thus, vcInfo embodies the VFlow's impact indicator which may be different from the change set's impact indicator, $\hat{p}$. The conflictPlace is where the conflict was found. Each conflict in vcMatches will be at a unique place because a verification concern can only appear once at a place. If any change to a verification concern can strongly impact the risk of reusing the verification process for the requirement, it will be reflected in the impact indicator in vcInfo.

The set vcConflicts in the blue token holds the pink token's information for comparison with information at each place in the VFlow as the blue token traverses the CPN. Given that progress properties embody dependencies among state variables. These relationships are captured in the blue token's dependencies set, which allows the blue token to manage the dependencies by enforcing a check on the impact of a verification concern at a place before or after a conflict was already found with its dependent verification concern. The conflictCount generates the unique ID for each pink token information, while the tokenCount generates a unique ID per red token.

**Red tokens** are output by transitions to represent alerts. These alerts indicate potential conflicts between the adaptation's change set and the requirement's original verification process. The structure of a red token is as follows.

$$\mathbf{t_{red}} = (\text{ID, IDconflict, vcInfo, } \hat{p}\text{, conflictPlace, placeStatus})$$

The red token ID and IDconflict are assigned by the blue token prior to the red token being sent to the end state. The red token must hold all of the factors needed for the risk assessment of the adaptive plan for that VFlow. The set vcInfo contains tuples of the form (VC, vcImpact), so that the transitions' impact factor based on the pink token's change set condition is recorded. The pink token's pˆvalue is also maintained in the red token along with the place where the conflict occurred and the weight of that place's importance to the verification process.

An example transition appears as pseudo code below. This is one of 18 transition rules used in a VFlow based on our modeling of verification processes for safety and progress properties. Transitions always require a blue token and a pink token as input. In this example, the blue token, B, has not visited the input place to the transition. The transition, T, does not have a verification concern (VC) that conflicts with what the blue token has accumulated in B.vcConflicts. The transition does have a conflict with a VC in the pink token P's change set. We have successfully encoded these rules and tokens into the CPN and can now automate their generation along with the risk factors discussed next.

**Transition Conditions**:
T.place_name is not in B.visited
No VC in T.vcInfo appears in B.vcConflicts
A VC in P conflicts with a VC in T.vcInfo
**Transition Actions**:
**FORALL** VC in T.vcInfo that appear in P
Increase B.count and $B.ID_{conflict}$
Create a red token, R, with B.count as its ID, $B.ID_{conflict}$ as its $ID_{conflict}$, and other information held by P and T
Update B.vcMatches to include $ID_{conflict}$ and the appropriate information held by T for all matching VCs
Add the appropriate change information from P to B.vcConflicts using B.count for $ID_{conflict}$
**FORALL** VC in P that are not in T.vcInfo
Increase B.count and B. $ID_{conflict}$
Add the appropriate change information from P to B.vcConflicts using B.count for $ID_{conflict}$
Add T.placeName to B.visited
Send B to output place
Send P to its place of origin
Send all red tokens to end place

**Figure 14: Pseudo Code of an example transition**

#### 3.2.3.3 Initial Utility Function for Risk Assessment and Plan Comparison

For each requirement $r$ and adaptation plan $a$, the VFlow outputs a set of red tokens $T(r, a)$ representing alerts. Once the red tokens are generated, the system calculates a metric that can be used to compare the risks of adaptation plans based on the token information. To obtain a workable formula, we assume that each red token, independently of all other tokens, has the potential to represent an *actual violation* of verification process reuse. That is, the adaptive plan has altered something that was relied on by the original proof of the requirement. We also assume that verification process reuse is violated if and only if there is at least one red token representing an actual impact. We assume that a group of red tokens cannot represent the violation in combination, without doing so individually. For each red token $t$, let $S(t)$ be an indicator variable with value 0 if $t$ represents an actual violation and 1 otherwise. For each requirement $r$ and adaptation plan $a$, let $I(r, a)$ be an indicator with value 0 if $a$ violates the reuse of $r$'s proof and 1 otherwise. Although the values of $S(t)$ and $I(r, a)$ would typically be deterministic, we assume they are infeasible to

compute, and therefore model $S(t)$ and $I(r,a)$ as random variables. Our assumptions given above translate to the following statements. For each requirement $r$ and plan $a$,

$$I(r,a) = 0 \Leftrightarrow \big(\exists\, t \in T(r,a)\big)(S(t) = 0)$$

For each requirement $r$ and plan $a$, the random variables in the set are mutually independent.

$$\bigcup_{t \in T(r,a)} \{S(t)\}$$

From these statements, we deduce that the probability that plan $a$ does *not* violate the reuse of requirement $r$'s proof is

$$P(I(r,a) = 1) = \prod_{t \in T(r,a)} P(S(t) = 1).$$

To compare adaptation plans, we define the *requirement utility* of plan $a$ to be the weighted sum

$$U(a) = \sum_{r \in R} w(r) I(r,a)$$

where $R$ is the set of requirements and $w(r)$ is a stakeholder-supplied *utility weight* for the need to maintain system compliance with requirement $r$. The expected requirement utility is then as follows.

$$E[U(a)] = \sum_{r \in R} \left( w(r) \prod_{t \in T(r,a)} P(S(t) = 1) \right)$$

If the expected requirement utility can be computed, it can be used as a metric to distinguish riskier plans from less risky plans. However, this goal requires an estimate $p(t) \approx P(S(t) = 1)$ for each token $t$. Each red token contains such an estimate $\hat{p}(t)$, based on whatever knowledge the planner may have to compute it. Since $\hat{p}(t)$ is presumed to have been computed without consideration of the characteristics of the original verification process, we wish to adjust it based on proof-related information to get the final estimate $p(t)$.

Our approach to computing $p(t)$ is based on the idea that verification concerns and architectural places can have differing levels of prominence or impact in a verification process. We assume that red tokens generated from a high-impact place or verification concern are more likely to represent actual reuse violations than those coming from low-impact places or concerns. A red token $t$ contains impact multipliers $M_{PL}(t)$ and $M_{VC}(t)$, representing the impact of the

architecture place and verification concern (respectively) from which the token was generated. Lower multipliers represent higher impact/risk. We apply these multipliers to $\hat{p}(t)$ to get $p(t)$, resulting in an estimate that takes into account the proof characteristics.

We considered two possible ways to apply the multipliers: scaling the probability and scaling the odds. The latter allows for the possibility of multipliers greater than 1, which would indicate that the estimate given by the planner should be increased rather than decreased. One of the outcomes of this study is a comparison of the two approaches based on how well they estimate the relative risk of adaptations.

With probability scaling, we have

$$p(t) = M_{PL}(t)M_{VC}(t)\hat{p}(t).$$

With odds scaling, we have

$p(t) = \frac{o(t)}{1+o(t)}$, where $o(t) = \frac{M_{PL}(t)M_{VC}(t)\hat{p}(t)}{1-\hat{p}(t)}$.

(If $\hat{p}(t) = 1$, this formula is undefined, and we instead use $p(t) = 1$.)

#### 3.2.3.4 Risk Analysis using the MMTS Case Study

To evaluate our risk comparison methodology, we apply it the MMTS case study described in Section 3.2.1. The case study involves a system on which we impose multiple self-adaptive plans. As the system is quite simple, we can manually reason about and compare the risks of each adaptation. Our goal is to determine whether the more mechanistic utility comparison metric described in Section 3.2.4 can come to conclusions similar to those which we have derived manually, given the original verification processes.

The MMTS consists of a traveler that moves in a grid while attempting to avoid stationary enemies distributed randomly on the grid. At each step, the traveler attempts to choose a new position and move to it. Based on the direction of the move, the traveler's fuel level may increase, decrease, or stay the same when it reaches the next position. The traveler is given an upper and lower limit on its fuel value, and must keep the fuel within that threshold. More complex variants of the system employ mission planning and enemy avoidance.

The MMTS base code provides an update process that chooses and sets the new position and fuel value. We identified three high-level architectural components that comprise the update process. The first is getCurrentStatus (gCS), which reads and validates the state at the start of the update. The second is getNextPosition (gNP), which determines the set of valid moves and randomly chooses a move from that set. The third is setPosition (sP), which moves the traveler to the chosen position and updates its fuel level. These three components form the architectural places in the VFlow for the two of the MMTS requirements, R1 and R2, stated in LTL below.

**R1**: $\square$ (minFuel $\leq$ fuel $\leq$ maxFuel)

**R2**: □ ((canMove ∧ position = p) ⇒ ○notAt(p))

For R1, the fuel level must stay within the threshold at all times. For R2, if the traveler can move at a given time step, then it must move. $canMove$ is defined as validPositions ≠ ∅, where $validPositions$ is computed according to the current traveler state to exclude positions containing enemies and moves that would lead to a fuel threshold violation. A random move from the $validPositions$ set is chosen in each update. If the set is empty, the traveler stays in its current position. $notAt(p)$ is defined as $position \neq p$, given p represents the current position. In the prior report, this requirement is listed as R3, because R2 was to ensure that the traveler was not in an enemy position. Since the original R2 was a safety property, its risk factors were similar to R1 relating to the fuel threshold. But R3 (here R2) is a progress property, which increases the complexity of the verification process and the dependency expressions for the verification concerns. Hence, we temporarily removed the original R2 and replaced it with the progress property R3 for the experimentation with our utility function. In addition, for the MMTS case study, we assume the stakeholder-supplied utility weights of the requirements are $w(\text{R1}) = 0.75$ and $w(\text{R2}) = 1$.

We have verified that the MMTS code (without adaptation) satisfies these requirements and derived the verification concerns. As an outcome of the verification process, we extract impact multipliers for use in the risk comparison calculations. These multipliers are based on the prominence of different verification concerns and architectural places in requirements' proofs, as well as the conditions required by the proofs.

Table 4 shows the place impact multiplier for each of the 3 architectural places in MMTS for each requirement (R1-safety, R2-progress). In this case study, we manually assigned values of 0.2 (high impact), 0.5 (medium impact), or 0.9 (low impact) based on our perception of the importance of each place in each proof. These values and others described in this section are shown in the example VFlow for R1 in Figure 15.

**Table 4. Impact multiplier for the VFlow place if a change occurs**

| | **gCS** | **gNP** | **sP** |
|---|---|---|---|
| **R1** | 0.9 | 0.5 | 0.2 |
| **R2** | 0.5 | 0.5 | 0.2 |

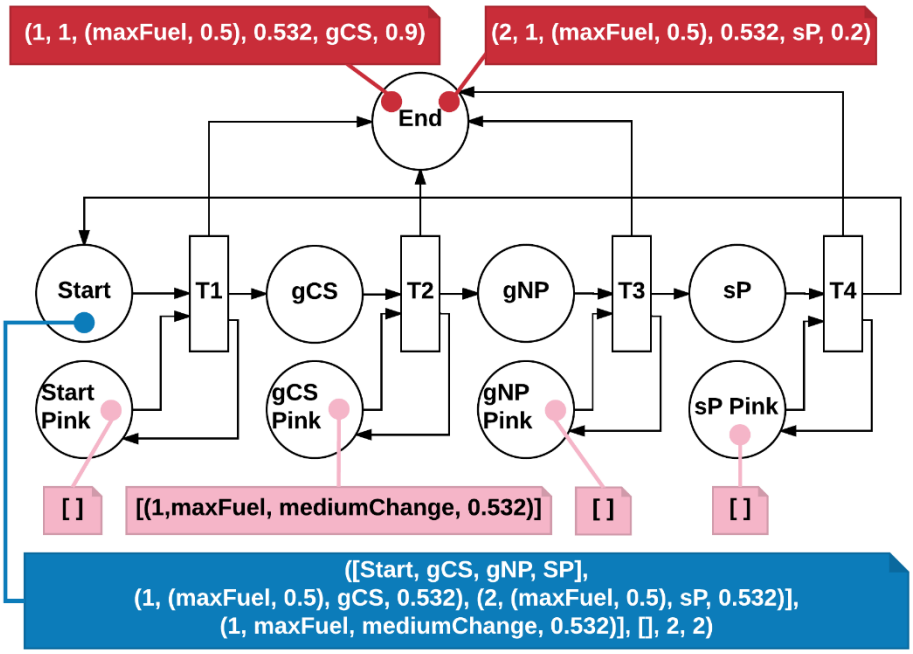



**Figure 15. VFlow for R1 with the change set from adaptation A1**

As discussed previously, the impact multiplier for a verification concern can depend on the type of change made. Verification concern impacts are categorized as *devastating*, *worrisome*, or *unconcerned*, and the corresponding impact multiplier values are 0.2, 0.5, and 0.9, respectively. Table 5 and Table 6 show the rules that we used for categorizing the relevant verification concerns' impacts for R1 and R2, respectively. The categorization is based on the type and/or magnitude of the change, which is supplied by the planner.

**Table 5. Verification concern impacts from the VFlow perspective for R1**

| R1 | devastating | worrisome | unconcerned |
|---|---|---|---|
| fuel | Change greater than or equal to $maxFuel - minFuel$, positive or negative. | Change greater than or equal to $\frac{maxFuel-minFuel}{2}$, positive or negative. | Change less than $\frac{maxFuel-minFuel}{2}$, positive or negative |
| minFuel | Change greater than or equal to $maxFuel - minFuel$, positive. | Change greater than or equal to $\frac{maxFuel-minFuel}{2}$, positive. | Negative change or change less than $\frac{maxFuel-minFuel}{2}$. |
| maxFuel | Change greater than or equal to $maxFuel - minFuel$, negative. | Change greater than or equal to $\frac{maxFuel-minFuel}{2}$, negative. | Positive change or change less than $\frac{maxFuel-minFuel}{2}$. |

**Table 6. Verification concern impacts from the VFlow perspective for R2**

| R2 | devastating | worrisome | unconcerned |
|---|---|---|---|
| fuel | Set to 0. | Large change, positive or negative. | Small change, positive or negative. |
| minFuel | Set to maxFuel. | Increased to a value less than maxFuel. | Set to 0. |
| maxFuel | Set to 0. | Decreased to a value greater than minFuel. | Increased. |
| validPosition | Changed to a nonempty set with no change in nextMove. | Changed to a nonempty set with a change in nextMove. | Set to ∅. |
| nextMove | Set to null with no change in validPosition. | Set to null with a change in validPosition. | Set to null with validPositionchanged to ∅. |
| position | Changed with nextMove set to null. | Maintained with nextMove set to null. | Maintained when validPositionis empty. |

### 3.2.3.5 Potential Adaptations

In our example scenario, the MMTS is initialized with the traveler at position (0,0) with a fuel value of 80, a lower fuel threshold minFuel = 0, and an upper fuel threshold maxFuel = 160. The system is simulated for 25 time steps, at which point an engine failure occurs, triggering the adaptation process. The planner configures possible adaptation plans and constructs their change sets to assess the risk of impacting proof reuse for requirements R1 and R2. To compare more plans and perform deeper risk analysis, we increased the number of potential adaptive plans from 2 to 4. The original and new adaptive plans are described below.

**A1**: maxFuel is divided by 2 within gCS. (original)
**A2**: maxFuel is reduced by 40, and minFuel is increased by 40 within gCS. (new)
**A3**: The next move for the traveler is not chosen for 5 time steps within gNP, even if validPositions is nonempty (to simulate a stop for repair). (new)
**A4**: validPositions is changed to the empty set for 5 time steps (to simulate a stop for repair) in gNP. (original)

### 3.2.3.6 Examining Adaptation Plan Risks

By manual analysis and simulation, we have identified the potential risks of each plan with respect to the requirements. A1 is risky for R1, as it leads to a violation if the current fuel value is greater than 80. When R1 fails, it also causes a failure in R2, because the traveler stops moving if it detects a violation of the fuel threshold. Therefore, A1 is also risky for R2. A2 can also pose a threat to R1 and R2 if the current fuel value is below 40 or above 120, but that is not possible given the function for calculating the fuel by the 25th time step when the adaptation occurs. Therefore, A2 has very little risk if performed early in the traveler movement.

A3 and A4 both disallow movement for 5 moves, which poses no threat to R1. A3 is very risky for R2, and in fact will always cause a violation if the set of valid moves is nonempty at the time

of the adaptation. A4 is superficially similar to A3, but it actually is not risky for R2, because R2 only requires movement when the set $validPositions$ is nonempty.

Based on our analysis, A3 should be considered risky for R2, while A1 should be considered risky for R1 and R2. A2 might be considered marginally risky for R1 and R2, because it could fail under some circumstances, although those circumstances are not possible in our simulation. A4 poses no risk for either of the requirements.

It remains to be shown how the planner generates the initial success probability estimate $\hat{p}$ for each plan's pink token(s). In this case study, we assume the planner knows that reducing the set of valid moves is less risky, so it uses $\hat{p} = 0.99$ for all of A4's pink tokens. We assume the planner has been able to determine that changing the logic in getNextPosition (gNP) has some risk to both requirements, so it uses $\hat{p} = 0.5$ for A3's pink tokens.

For A1 and A2, we assumed that the planner might run predictive simulations involving perturbations of minFuel or maxFuel, to estimate the sensitivity of the system to such changes. We performed 200-step simulations in which either minFuel or maxFuel was perturbed at the 100th step, with 1000 runs of the simulation for each perturbation size and requirement. The simulations produced a success rate of 0.532 for R1 when maxFuel was reduced by 80, meaning 53.2% of the 1000 simulations found no violation of R1. The corresponding success rate for R2 was 0.505. Therefore, we set $\hat{p} = 0.532$ for A1's pink token in R1's VFlow, and $\hat{p} = 0.505$ for the pink token in R2's VFlow.

In all simulations where minFuel was increased by 40 or maxFuel was decreased by 40, no proof violations were detected. Assuming the planner would be cautious enough not to indicate a guaranteed success based on a simulation, we set $\hat{p} = 0.99$ for all of A2's pink tokens.

#### 3.2.3.7 Computed Results

Table 7 and Table 8 show the success probabilities computed from the set of red tokens for each requirement/plan pair, along with the expected utility based on the probabilities and the requirements' utility weights. Table 7 provides the results for the probability scaling approach. Table 8 shows the odds scaling results.

**Table 7. Results from probability scaling approach**

| *Results Using Probability Scaling* | | | |
|---|---|---|---|
| | **R1 Success Prob.** | **R2 Success Prob.** | **Expected Utility** |
| **A1** | 0.0127 | 0.00708 | 0.0166 |
| **A2** | 0.0204 | 0.00630 | 0.0216 |
| **A3** | 1 | 0.0500 | 0.800 |
| **A4** | 1 | 0.446 | 1.20 |

**Table 8. Results from odds scaling approach**

| *Results Using Odds Scaling* | | | |
|---|---|---|---|
| | **R1 Success Probability** | **R2 Success Probability** | **Expected Utility** |
| **A1** | 0.0345 | 0.0226 | 0.0485 |
| **A2** | 0.875 | 0.858 | 1.51 |
| **A3** | 1 | 0.0909 | 0.841 |
| **A4** | 1 | 0.978 | 1.73 |

The risk values calculated using odds scaling match fairly well with what we would expect based on our manual reasoning. For R1, A3 and A4 were found to have no risk, A2 was found to have low risk (high success probability), and A1 was found to have high risk. For R2, A1 and A3 were found to have high risk, while A2 and A4 had low risk. However, there are some discrepancies from what we would expect. For example, the success probability computed for R2 was higher for A3 than for A1, even though A3 nearly always causes a failure for R2 while A1 causes failures less frequently. The same issue occurs when probability scaling is used. An additional problem occurs with probability scaling in that A2 is found to have the lowest success probability for R2, even though it is one of the safer adaptations for that requirement.

### 3.2.4 Employing KIV and Working toward Security Certification Awareness

In this section, we detail efforts (1) investigating the formal expression and classification of security controls used for certifying federal information systems, (2) using the SIMS case study to show the impact of adaptation on security certification and (3) develop a theoretical methodology and heuristic guidelines for the use of the KIV theorem prover (Ernst, 2015) to identify verification concerns, the impact of their change on the risk of inhibiting the verification process, and the modeling of the verification workflow for adaptation plan comparison and risk assessment.

To scope our examination of the impact of self-adaptive plans on security certification, we start with three main assumptions.

- The planning process as part of the MAPE-K (Monitor, Analyze, Plan, Execute with Knowledge) can produce adaptive plans that may not have been evaluated prior to system deployment,
- A static analysis tool can be available that disallows any configured plans that do not follow predefined rules, such as naming conventions, and
- If an adaptive plan interferes with or alters a verification concern extracted from the proof process, there is an increased risk that the same proof strategy to show security control compliance cannot be reused, providing the basis for the risk assessment of the plan.

The first assumption allows the planner to go beyond prefabricated plans or plans that must satisfy predefined adaptive constraints. The planner can use any knowledge of predefined plans, attempted but not deployed plans, and meta-data from successfully deployed plans. The second assumption allows for constraints to be applied to the plan configuration. With this assumption, we can introduce constraints on deleting processes that explicitly serve as security control mechanisms as part of an adaptive plan configuration.

The third assumption forms the foundation of the approach. Because each requirement verification or certification process is documented, the pattern of control flow and how states are examined provide meta-data about the process: verification concerns (VCs) and verification workflow (VFlows). VCs and VFlows become part of the planning process within the MAPE-K control loop as verification and certification awareness. If an adaptive plan alters a VC in a risky manner at a risky place in the workflow, the original verification/certification process may be voided and not be reusable. The greater the risk, the higher the probability that a new proof process would be needed, thus increasing the probability of a requirement violation by the adaptive plan.

#### 3.2.4.1 Transforming Security Controls

Because security controls express functional and non-functional requirements, certification processes involve determining the existence of certain functionality in the system, as well as the correctness of the functionality. In this report, we focus on security control AU-12(1), which appears in Figure 16, as taken directly from the NIST SP 800-53 (NIST, 2013). AU-12 is designated all baselines (i.e. low, moderate, and high impact systems). This means that all information systems adhering to the security controls must consider AU-12. Enhancement 1 is part of the baseline of controls for high impact systems. A high impact information system means that there is a high degree of concern, such as monetary, reputation, or life-threatening, should one of confidentiality, integrity, or availability be violated with respect to the data stored and in transit.

Figure 16 shows that AU-12 references additional audit controls, namely AU-2a, AU-2d, and AU-3. These control statements appear in Figure 17 as taken directly from the NIST SP800-53 (NIST, 2013). Security certification examines the requirements of AU-12, followed by the requirements for its first enhancement and assessing these against the system. The NIST SP800-53a (NIST, 2014) guidelines state that examination can be done on the “procedures addressing audit record generation” because of the reference to AU-2a and “list of auditable events” because of the reference to AU-2d and AU-3. When enhancement (1) is included, the examination guidelines extend to “system-wide audit trail”.

**AU-12 AUDIT GENERATION**

Control: The information system:

a. Provides audit record generation capability for the auditable events defined in AU-2 a. at [*Assignment: organization-defined information system components*];

b. Allows [*Assignment: organization-defined personnel or roles*] to select which auditable events are to be audited by specific components of the information system; and

c. Generates audit records for the events defined in AU-2 d. with the content defined in AU-3.

Control Enhancements:

**(1)** *AUDIT GENERATION | SYSTEM-WIDE / TIME-CORRELATED AUDIT TRAIL*

**The information system compiles audit records from [*Assignment: organization-defined information system components*] into a system-wide (logical or physical) audit trail that is time-correlated to within [*Assignment: organization-defined level of tolerance for the relationship between time stamps of individual records in the audit trail*].**

**Figure 16: AU-12 Control Statement with Enhancement (1)**

**AU-2 AUDIT EVENTS**

Control: The organization:

a. Determines that the information system is capable of auditing the following events: [*Assignment: organization-defined auditable events*];

…

d. Determines that the following events are to be audited within the information system: [*Assignment: organization-defined audited events (the subset of the auditable events defined in AU-2 a.) along with the frequency of (or situation requiring) auditing for each identified event*].

**AU-3 CONTENT OF AUDIT RECORDS**

Control: The information system generates audit records containing information that establishes what type of event occurred, when the event occurred, where the event occurred, the source of the event, the outcome of the event, and the identity of any individuals or subjects associated with the event.

**Figure 17: Control Statements for AU-2a, AU-2d, and AU-3**

Formally expressing the security control statements manifests their separation of non-functional from functional requirements. Below we restate them as R1-R5 using formal notation and the ontological relationships expressed in (Hale, 2017). In LTL, the square is “invariant” and the diamond is “eventually”.

**R1**: (non-functional) For each component (C) in the information system (IS) identified to have auditable events, there exists a function to perform auditing for predefined auditable events (AU-12a, AU-2a). Below, we call the function *generateAuditRecord*.

$$(\forall C \in IS: \exists C.\, generateAuditRecord)$$

**R2**: (functional) For each predefined auditable event, an audit record satisfying the minimal contents is generated when an auditable event occurs (AU-12c, AU-2d, AU-3)

$$\Box(auditableEvents(e) \Longrightarrow \Diamond(recordGenerated(e)))$$

where $auditableEvents$ contains tuples of the form $(v, t, w, c, o, id)$, with $v$ the event type; $t$ the event time; $w$ where event occurred; $c$ the event source; $o$ the event outcome, and $id$ the event identity.

**R3**: (non-functional) There exists a system-wide, virtual or physical component, which we will call $Audit$, that collects auditable events (AU-12(1)) from designated components.

$$(\exists C \in IS : Audit \in C)$$

**R4**: (functional) All audit records are sent to the audit trail (AU-12(1))

$$\Box(recordGenerated(e) \Rightarrow \Diamond(e \in auditTrail))$$

**R5**: (functional) The audit trail time-correlates the audit records (AU-12(1))

$$\Box(\forall\, i, j \in [0, \|auditTrail\|) \cap \mathbb{Z} :$$
$$i < j \Rightarrow auditTrail[i].t \leq auditTrail[j].t)$$

**R6**: (functional) Time correlation is checked against a defined level of tolerance (AU-12(1))

$$\Box(\forall\, i \in [0, \|auditTrail\| - 1) \cap \mathbb{Z} :$$
$$auditTrail[i+1].t - auditTrail[i].t > \varepsilon \Rightarrow$$
$$flag(auditTrail[i].t, auditTrail[i+1].t))$$

R1 requires that the mechanism exists in identified components to capture predefined auditable events. R3 requires the existence of an audit trail. The requirements R2 and R4-R6 must be certified by testing or formal verification.

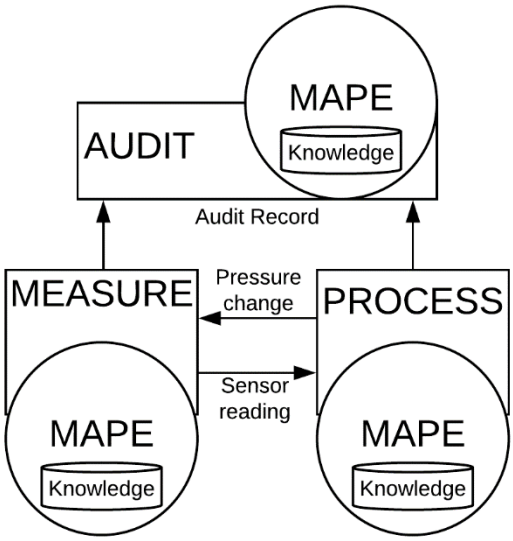



**Figure 18: SIMS Architecture**

Figure 18 shows a different SIMS architecture than originally designed in Figure 9, because the connectivity process was unnecessary when the simulation of the system was implemented as multiple threads in Java. SIMS tracks inventory's stock condition using a pressure sensor and adjusts pressure the threshold depending on the inventory movement. Thus, for this experiment, SIMS has three components with local MAPE-K control loops: Measure reads the pressure sensor, Process adjusts the threshold, and Audit houses audit records from Measure and Process for security certification.

#### 3.2.4.2 Examining the Audit Security Controls

Within Figure 16 and Figure 17 there is text of the form [*Assignment*: …]. This text provides the organization with the flexibility to tailor the security controls with more explicit values or expressions so that they are relevant to the system of interest. Once tailoring is performed, the resulting values cannot be changed without re-certification, and thus, form safety properties.

Table 9 shows the tailored values chosen to satisfy AU-12(1), AU-2a, AU-2b, and AU-3 for the SIMS case study. Once tailoring is completed, certification can begin. For the targeted audit security controls in Section 3.2.4.1, the first part of the process is to determine if the required mechanisms exist to satisfy R1 and R3. For R1, Table 9 indicates that Measure and Process are responsible for record generation (row 1) and produce audit records from the auditable events of signal updates in Measure and threshold value changes in Process (row 5).

To ensure the existence requirements (R1 and R3) are maintained in an adaptive plan, a mapping of the security control to the component process is provided to the planning process in the MAPE-K control loop. If an adaptive plan is configured that deletes the process, then the violation of the requirement is clear and can be provided as an alert to the human analyst that recertification will need to be performed for that mechanism.

We focus the requirements proof and self-adaptation assessment on the Audit component. Its process control flow is depicted in Figure 19. The main update loop consists of checkCongestion (logs the amount of buffer capacity being used), dequeueAuditRecord (removes an audit record from the queue), findInsertionPoint (uses a binary search to determine where the audit record's timestamp fits), checkTimeTolerance (logs when the time difference is not within the prescribed tolerance), and storeAuditRecord (stores the audit record at the determined insertion point of the audit trail).

**Table 9: Tailored Values for the Targeted Security Controls**

| Targeted control | Organization-defined | Assignment |
|---|---|---|
| AU-12a | information system components (for audit record generation capability) | Measure, Process |
| AU-12b | personnel or roles | Human analyst |
| AU-12(1) | information system components (from which audit records will be compiled) | Measure, Process |
| AU-12(1) | level of tolerance for the relationship between time stamps of individual records in the audit trail | 10 seconds |
| AU-2a | auditable events | signal updates, threshold value changes |
| AU-2d | audited events with frequency of (or situation requiring) auditing for each defined event | at every occurrence |

### 3.2.4.3 Potential Adaptations

During deployment of the SIMS, the planning process configures the following potential plans for risk assessment, none of which violate the existence criteria for R1 and R3.

- **A1:** Allow Audit to periodically drop messages.
- **A2:** Add dequeued records to the end of the audit trail (instead of performing binary search) until a predefined number of records have been added.
- **A3:** Increase the performance of the sorting technique by providing a rolling lower timestamp and sorting only records less than it.

We will return to these adaptations and their impact on the verification of the audit security controls in Section 3.2.4.7.

To compare adaptations configured by the planner, we assess their risk of inhibiting the reuse of the original verification process of the security control requirements. In this section, we outline the methodology of extracting the verification concerns (VCs) and the verification workflows (VFlows) using the KIV theorem prover. We focus on proving the safety property R5 and the progress property R4 from Section 3.2.4.1. Figure 20 outlines the process to identify the VCs and

VFlow associated with a proof. With the Java code restated in KIV's language, KIV can prove LTL expressions. We define a set of lemmas that provide guidance to the KIV proof so that the proof process is made more explicit and the meta-information needed to identify VCs and VFlows per requirement is available.

**Table 10: Targeted Functions for Audit Control Certification of Mechanism Existence**

| Control | Requirement Mapping | Mechanism | Certification Process | Targeted Function(s) |
|---|---|---|---|---|
| AU-12a | R1 | Provide audit record generation capability | Examine Measure for capability | Measure.generateAuditRecord |
| | | | Examine Process for capability | Process.generateAuditRecord |
| AU-12(1) | R3 | Provide audit record compilation (physical) | Examine Audit for capability | Audit.findInsertionPoint<br>Audit.checkTimeTolerance<br>Audit.storeAuditRecord |
| AU-2a | R1 | Provide auditable event recognition | Examine Measure for capability specific to updating signal | Measure.updateSignal |
| | | | Examine Process for capability specific to computing threshold | Process.updateThresholds |

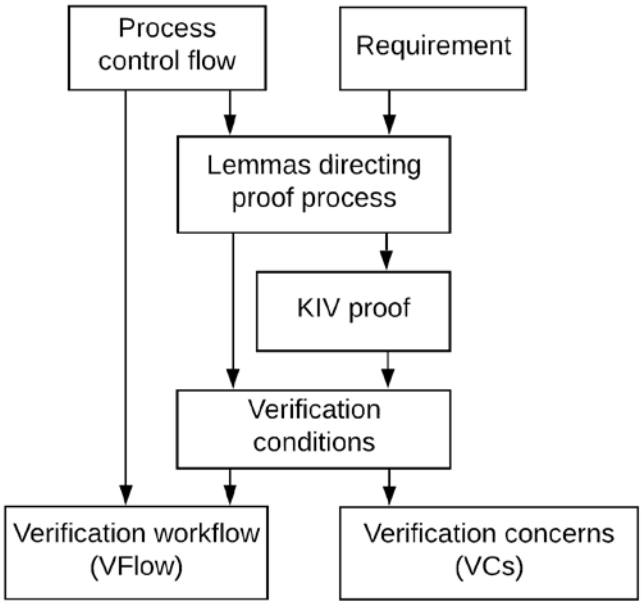



**Figure 19: Process Control Flow for Audit**

### 3.2.4.4 Working with KIV

Each LTL state transition has two phases in KIV: the program modifies the state and the environment modifies the state. Let $X$ be a state variable. $X'$ denotes the value after program modification. $X''$ denotes the value after program and environment modification, which is the value in the next state. It must be explicit about what the environment cannot change, otherwise KIV assumes the environment can change any state variable. For Audit, we assume the environment does not modify any state variables other than the input message queue.

Hoare triples (Hoare, 1985) are an intuitive way to prove program properties and extract VCs. However, they make no guarantees about the states in between their pre- and postconditions. Thus, they are not suitable for proofs where intermediate states must satisfy a property. To allow KIV to use Hoare triples when proving a property for a single component, we augment them with intermediate state and termination guarantees. Our *temporal contract proposition* (TCP) is similar to KIV's *rely-guarantee* statements for threads but can be used to decompose the different procedure calls in an individual program thread.

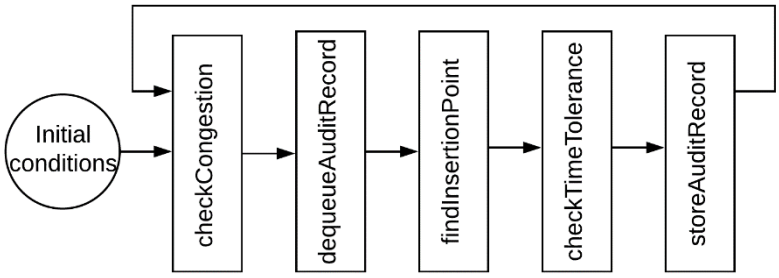



**Figure 20: VC and VFlow Identification Process**

A TCP is stated as $tcp(Pre, Code, Mid, Post)$, which means that when Code is executed under condition Pre, it eventually terminates with condition Post, and all intermediate states satisfy condition Mid. In KIV, we state a TCP with the template:

Pre, [: $V_{in}$, $V_{inout}$| Code($V_{in}$; $V_{inout}$); [PL RestProg] ]
⊢
Mid **until** (Post ∧ RestProg)

where $V_{in}$, $V_{inout}$ are any input/output program parameters. [*PL RestProg*] represents the program (if any) that executes after Code. *Mid* **until** (*Post* ∧ *RestProg*) means (*Post* ∧ *RestProg*) holds in the present or some future state, and *Mid* holds in every state before that, starting at the present state.

#### 3.2.4.4.1 Verification of Safety Property R5

In KIV, invariants to be proven are often represented as

$$\neg(N'' = N - 1 \textbf{ until } \neg p)$$

to mean "there does not exist a natural number variable *N* that decreases until *p* is false." Typically, for KIV to prove an invariant it symbolically executes a full iteration of the main loop, considering all program branches and proving *p* at every step. Then it applies induction to complete the proof.

Instead of focusing on symbolic execution directly, our methodology uses TCPs when possible to skip over parts of the code until a main loop iteration has been completed. Symbolic execution is used directly to prove the TCPs, but not to apply them. In accordance with the KIV TCP template described in Section 3.2.4.4, applying $tcp(Pre, Code, Mid, Post)$ to a proof goal where *Pre* and [*: $V_{in}$, $V_{inout}$| Code($V_{in}$; $V_{inout}$);* [*PL RestProg*] ] are known to be true allows KIV to deduce *Mid* **until** (*Post* ∧ *RestProg*). Note that we must instantiate *RestProg* with a specific program formula when applying a TCP, but not when proving one. To actually skip over *Code* we need to derive a new proof goal where the program formula is *RestProg*. We apply the following lemma, called *lemma-invariant*, after applying the TCP:

*N* = *n*,
□(*After*$_I$ ∧ *InvProp* ∧ $N \leq n$ ⇒ ¬(*N* = *N''* + 1 **until** ¬*InvProp*)),
□(Mid$_I$ ⇒ InvProp),
Mid$_I$ **until** (InvProp ∧ After$_I$)
⊢
¬(*N* = *N''* + 1 **until** ¬*InvProp*)

We apply *lemma-invariant* using the substitution After$_I$ = *Post* ∧ *RestProg* and Mid$_I$ = Mid. InvProp is substituted with the invariant property to be proven (i.e., to prove □ p, we use InvProp

= p). Given that Post contains InvProp, the fact *Mid* **until** (*Post* ∧ *RestProg*) derived from the TCP implies $Mid_I$ **until** (*InvProp* ∧ $After_I$). N = n can always be established for some n.

Since KIV knows that $Mid_I$ **until** (*InvProp* ∧ $After_I$) and N = n hold when *lemma-invariant* is applied, two new proof goals result, matching the second and third formulas in *lemma-invariant*. Using KIV's *execute always* rule, we can remove the *always* from these proof goals. Therefore, the new proof goals that we reach are:

$After_I \wedge InvProp \wedge N \leq n \Rightarrow \neg$ (*N = N''* + 1 **until** ¬ *InvProp*),
$Mid_I \Rightarrow$ InvProp

Figure 21 shows a fragment of a KIV proof tree in which a TCP has been applied, followed by an application of *lemma-invariant*. In this case, KIV was able to automatically close the proof goal $Mid_I \Rightarrow InvProp$, resulting in only one new proof goal.

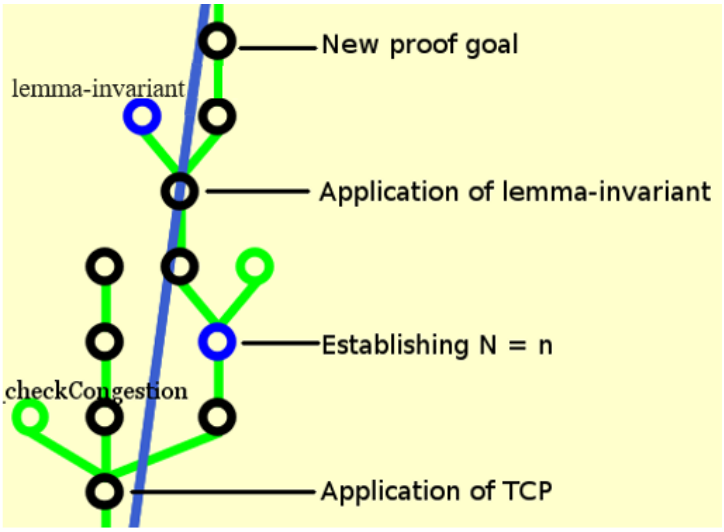



**Figure 21: KIV Tree Showing Guided Invariant Proof Process**

To roughly match our Java simulation's architecture for the SIMS case study, our KIV code assigns four (compound) state variables to each component: internal state, external state visible to the environment, and input and output message queues. Since the Audit component uses only one input queue and does not send messages, we abstract the KIV Audit state in this discussion to three variables: Int for the internal state, Ext for the external state, and InQ representing Audit's single input message queue. For brevity in our discussion, we define some abbreviations:

$e \equiv \Box$ (Int'' = Int' ∧ Ext'' = Ext' ∧∃ Msgs :InQ'' = InQ' + Msgs)
$s \equiv$ sorted(Ext.auditTrail, compareRecords)
$t \equiv$ Int.auditRecord = nullOpt ∨ Int.insertionPoint < 0 ∨
(Int.insertionPoint ≤ # Ext.auditTrail ∧

sorted(Ext.auditTrail.insert (Int.insertionPoint, Int.record.getOpt),
compareRecords))

where *e* is an environmental assumption saying that the only possible environmental state modification is to add a list of new messages to the back of InQ. *s* arises from our modeling of Java comparators, which are used in the Java code. compareRecords is a comparator constant that compares audit records based on time stamp. Sorted is a predicate indicating that a list is sorted according to a comparator.

In our proof of R5, we first show that $\Box s$ implies R5, making $\Box s$ an intermediate invariant to be proven. *t* is an intermediate condition needed in some of our TCPs. Intuitively, *t* means that there is no audit record being passed to the process in the control flow (see Figure 19) *or* no insertion point has been selected *or* inserting the audit record at the insertion point preserves the sorting. The use of nullOpt and getOpt in this formula arise from our modeling of *optional* sorts, which can have null values.

The resulting TCPs constructed to prove $\Box s$ are as follows:

*tcp*($e \wedge s$, checkCongestion(; Int, Ext, InQ), *s*, *s*)
*tcp*($e \wedge s$ , dequeueAuditRecord(; Int, Ext, InQ), *s*, *s*)
*tcp*($e \wedge s$ , findInsertionPoint(; Int, Ext, InQ), *s*, $s \wedge t$)
*tcp*($e \wedge s \wedge t$, checkTimeTolerance(; Int, Ext, InQ), *s*, $s \wedge t$)
*tcp*($e \wedge s \wedge t$, storeAuditRecord(; Int, Ext, InQ), *s*, *s*)

After applying these TCPs in order and following each TCP application with *lemma-invariant*, we reach a proof goal where the program formula is:

[: Int, Ext, InQ | **while** true **do** update(; Int, Ext, InQ) ]

This is the same program formula we had when we started the induction, because we have completed an iteration of the main loop. KIV also knows that N has decreased, because it decreased once in the evaluation of the loop condition and we preserved that fact by the inclusion of $N \leq n$ in the new proof goals (using *lemma-invariant*). Since N has decreased without a violation of *s*, and we have reached the original program formula, we can have KIV apply the induction to complete the proof of $\Box s$.

### 3.2.4.5 Identifying the Verification Concerns and Verification Workflow

Verification concerns (VCs) are state variables with values that are relevant to the proof. VCs are identified by verification conditions (see Figure 20) that are examined within the instantiated *lemma-invariants* associated with the TCPs used to guide the KIV proof. For example, the conditions

auditRecord = nullOpt
insertionPoint ≤ #auditTrail

are a subset of conditions needed to prove R5 within the processes findInsertionPoint, checkTimeTolerance, and storeAuditRecord. The conditions rely on auditRecord, insertionPoint, and auditTrail, identifying them a VCs for R5, as well as the processes where the conditions are checked, which will be used in the construction of the Colored Petri Net (CPN) that describe the associated verification workflow.

Once the VCs are identified, the conditions used for the proof indicate the range of potential changes to the variables that may increase risk. While it is plausible that finding the VCs could be automated given the resulting KIV proof, the flexibility with which the VCs can be changed by self-adaptation is still a manual procedure. Table 11 shows how a human analyst might classify the impact of certain changes to a VC given the conditions used in the proof of R5. We separate them into fuzzy sets of Devastating, Worrisome, and Unconcerned.

**Table 11: R5 VC Conditions for Change Impact**

| R5 | Devastating | Worrisome | Unconcerned |
|---|---|---|---|
| auditRecord | | Alter time stamp<br>Set to null | Alter contents |
| insertionPoint | Continuous set to #auditTrail<br>Set to > #auditTrail<br>Eliminate sorting | | Alter sorting performance |
| auditTrail | Remove records | Reorder records | |

Removing auditRecord from auditTrail causes record loss, making both changes potentially devastating to maintaining R5. Setting insertionPoint to null and eliminating sorting is problematic, but decreasing the performance of the sorting algorithm may not cause problems. A change in the condition table that is part of an adaptation will be reflected in the risk impact factor, $M_{VC}$, for the targeted VC.

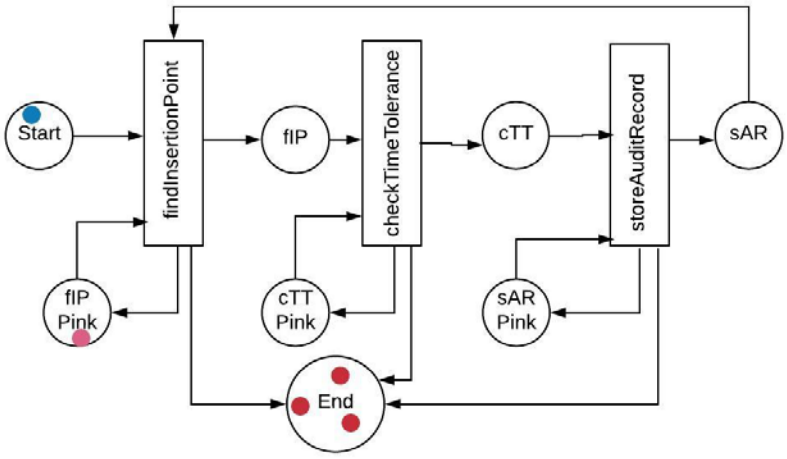



**Figure 22: VFlow for Safety Property R5 and Adaptation A2**

In Figure 20, the process control flow and the verification conditions that result from the KIV proof contribute to the definition of the verification workflow (VFlow) per requirement. We rely on prior work for the construction of a CPN as the VFlow representation as reported in the previous quarter. The objective of the VFlow is to inform the planning process of the comparative risk values associated with potential adaptation on the proof reuse for that requirement.

To calculate the overall utility of a plan, the CPN representing a VFlow mimics the process control flow with the processes as transitions. Three tokens are used: blue (for traversal through the VFlow), pink (representing the change) and red (representing the risk impact factors computed by the transitions).

Figure 22 represents the VFlow crafted by the meta-data taken from the proof of R5. Three processes, findInsertionPoint, checkTimeTolerance, and storeAuditRecord, are represented as transitions. Because the processes are equally involved in the proof, a change to them has equal impact on proof reuse. This will be reflected in the process impact multiplier, $M_{PL}$. For adaptation A2, the pink token indicates that the change will occur within findInsertionPoint. The planning process assigns values representing its assessment of the plan quality, which is the risk impact multiplier $\hat{p}$. The blue token carries the information presented by the pink token to each transition for assessment.

Red tokens are produced when a VC has an impact value above 0. They carry the accumulated impact values, $M_{VC}$, $M_{PL}$, and $\hat{p}$ to the end state. Multiple red tokens can be produced if multiple VCs are affected and if multiple processes contribute to the potential risk. For R5, adaptation A2 causes three red tokens to be generated, one from each transition, meaning that a VC at the transition was impacted. This VC is insertionPoint for each red token because A2 invokes the condition "Continuous set to #auditTrail" which is devastating. We will discuss how the impact values are used in Section 3.2.4.7.

#### 3.2.4.5.1 Verification of the Progress Property R4

For Audit, we decompose R4 into two progress properties.

**R4.1**: $\Box(recordGenerated(v) \Rightarrow \Diamond(v \in InQ))$
**R4.2**: $\Box((v \in InQ) \Rightarrow \Diamond(v \in \text{Ext}.auditTrail))$

In this section, we outline the methodology used to guide KIV to prove R4.2. We use *e* and *s* as defined in Section 3.2.4.4.1, along with the following abbreviations:

$e_1 \equiv \Box$(messageVar $\in$ InQ $\wedge$ InQ′=InQ $\Rightarrow$ messageVar $\in$ InQ′)
$s_1 \equiv$ messageVar $\in$ InQ $\Leftrightarrow$ messageVar $\in$ InQ′
$s_2 \equiv$ Int.auditRecord $\neq$ nullOpt $\wedge$
Int.auditRecord.getOpt = messageVar.auditRecord

where $e_1$ states that when the program does not modify InQ, a message in the queue will be preserved in the queue in the next state; this is easily proven from *e*.

We rely on the proof of R5 for $\Box s$ since our code relies on the fact that the audit trail is sorted to avoid generating an out-of-bounds insertion index, which would cause the program to crash. We make an additional assumption that we are only interested in messages containing audit records. This condition is stated as isRecordMessage(messageVar).

In KIV, the progress property goal for R4.2 is stated as

$\Box(q \Rightarrow \Diamond p)$, where
$q \equiv$ messageVar $\in$ InQ $\wedge$ isRecordMessage(messageVar)
$p \equiv$ messageVar.auditRecord $\in$ Ext.auditTrail

We refer to $q$ as the progress precondition under which some form of progress is required to eventually occur. We refer to $p$ as the progress postcondition that ensures the progress required by $q$ happens. When proving a progress property, it is helpful to decompose the proof across the process control flow so that, similar to the invariant proof, we can prove lemmas on smaller code blocks. Since the requirement has the form $\Box(q \Rightarrow \Diamond p)$, which includes a temporal formula inside the always, we require lemmas that are somewhat more complex than *lemma-invariant*.

Let the term *q-preserving* describe parts of the program in which the progress precondition cannot change from true to false. A key observation driving our methodology is that, due to the nature of the LTL *eventually* operator, we can use TCPs to "skip" over *q*-preserving code, only proving $q \Rightarrow \Diamond p$ at the end of each skip and when non-*q*-preserving code is encountered.

As in the invariant example, we can prove and apply a TCP $tcp(Pre, Code, Mid, Post)$ to allow KIV to deduce *Mid* **until** (*Post* $\wedge$ *RestProg*), then use that fact in an additional lemma. The lemma used to skip past *q*-preserving code is *lemma-progress:*

N = n,
$\Box$(After$_P$ $\wedge$ N $\leq$ n $\Rightarrow$
    $\neg$(N = N" + 1 until (PreProgress $\wedge \neg\Diamond$PostProgress))),
$\Box$(After$_P$ $\wedge$ PreProgress $\Rightarrow \Diamond$ PostProgress),
$\Box$(Mid$_P$ $\wedge$ PreProgress $\Rightarrow \bullet$ PreProgress),
Mid$_P$ **until** After$_P$
$\vdash$
$\neg$(N = N" + 1 **until** (PreProgress $\wedge \neg\Diamond$PostProgress))

where the $\bullet$ operator is a *weak next* indicating that PreProgress holds in the next state if there is a next state. This lemma is applied using the substitution After$_P$ = *Post* $\wedge$ *RestProg*, Mid$_P$ = Mid, PreProgress = *q*, PostProgress = *p*. It is used in a similar manner to *lemma-invariant*, but the proof goals generated are different. Again, using KIV's *execute always* rule, applying this lemma after a TCP results in the following new proof goals:

Post ∧ RestProg ∧ N ≤ n ⇒ ¬ (N = N" + 1 until (q ∧ ¬ ◇ p))

Post ∧ RestProg ∧ q ⇒ ◇ p

Mid ∧ q ⇒● q

Code that is not *q*-preserving creates a disjunction in the proof goal, because it may either (1) not actually change *q* from true to false, or (2) establish or preserve some other condition, $r$, that ensures ◇$p$. To represent this case, we define a *split temporal contract proposition* (STCP) as

$$stcp(Pre, Code, Mid, Post_1, Post_2)$$

meaning when Code is executed under condition Pre, either (1) it eventually terminates with condition $Post_1$, and all intermediate states satisfy condition Mid, or (2) it eventually terminates with condition $Post_2$, with intermediate states not necessarily satisfying Mid. Typically, Code represents a non-*q*-preserving program part, and $Post_2$ represents the intermediate condition $r$. We represent a STCP in KIV using the following template.

Pre, [: $V_{in}$, $V_{inout}$| Code($V_{in}$; $V_{inout}$); [PL RestProg] ]

⊢

(Mid **until** (Post1 ∧ RestProg)) ∨ ◇(Post2 ∧ RestProg)

The first case of the disjunction can be handled using *lemma-progress*. The second case requires a new temporal logic lemma, called *lemma-progress-split*:

N = n,

□(After$_{PS}$ ∧ N ≤ n ⇒

¬(N = N" + 1 until (PreProgress ∧ ¬◇PostProgress))),

□(After$_{PS}$ ⇒ ◇PostProgress),

◇After$_{PS}$

⊢

¬(N = N" + 1 **until** (PreProgress ∧ ¬◇PostProgress))

After applying a STCP, we apply *lemma-progress-split* with the substitution After$_{PS}$ = *Post*$_2$ ∧ *RestProg*, PreProgress = *q*, PostProgress = *p*. We get two new proof goals:

$Post_2$ ∧ RestProg ∧ N ≤ n ⇒ ¬(N = N" + 1 until (*q* ∧ ¬◇*p*))

$Post_2 \wedge RestProg \Rightarrow \Diamond p$

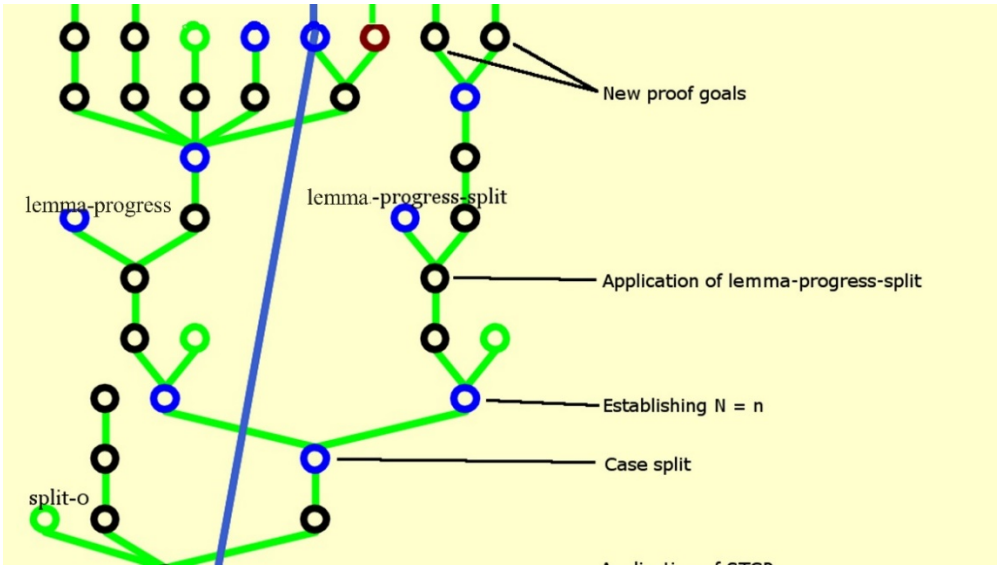



**Figure 23: KIV Tree with Progress Property Proof Process**

Figure 23 shows a fragment of a KIV proof tree (Ernst, 2015) where a STCP has been applied followed by an application of *lemma-progress* in one branch and *lemma-progress-split* in the other branch.

By applying TCPs and STCPs along with *lemma-progress* and *lemma-progress-split*, we eventually skip to the end of the program's main loop and can apply induction as with an invariant proof. At that point, the remaining proof goals have the form u ⇒ ◇p for some formula u. These goals may also use TCPs in their proofs, but as they do not require induction, *lemma-invariant*, *lemma-progress*, and *lemma-progress-split* are typically not required. For brevity, we list here only the TCPs and STCPs used with *lemma-progress* and *lemma-progress-split* to reach the end of the main loop, though the other TCPs do produce VCs:

*tcp*($e \wedge e_1 \wedge \Box s$, checkCongestion(; Int, Ext, InQ), $s_1$, *true*)

stcp($e \wedge e_1 \wedge \Box s$, dequeueAuditRecord(; Int, Ext, InQ), $s_1$, *true*, $s_2$)

*tcp*($e \wedge e_1 \wedge \Box s$, findInsertionPoint(; Int, Ext, InQ);
checkTimeTolerance(; Int, Ext, InQ);
storeAuditRecord(; Int, Ext, InQ),$s_1$, *true*)

### 3.2.4.6 Identifying the Verification Concerns and Verification Workflow

The precondition of *R4.2* identifies that the program is in a state in which messageVar, containing a non-null auditRecord, appears in InQ, identifying these three state variables immediately as VCs. They are also included as postconditions, since messageVar must eventually be removed from InQ, while auditRecord must eventually be placed in auditTrail. Thus, auditTrail is a VC. In addition, the proof relied on the invariant established conditions making insertionPoint also a VC. It is also noted that all processes have conditions in the proof that contain one or more of the identified VCs.

**Table 12** denotes the potential impacts of certain changes to the identified VCs from Section 3.2.4.5.1 for R4.2. It can be seen that the same changes in Table 11 are viewed as impacting this requirement differently. For example, sorting changes are of less concern because the requirement focuses on ensuring the audit records are stored in the audit trail.

**Table 12: R4.2 VC Conditions for Change Impact**

| R4.2 | Devastating | Worrisome | Unconcerned |
|---|---|---|---|
| messageVar | Modify auditRecord | | |
| InQ | Alter queuing of auditRecord | Reorder queue | |
| auditRecord | Alter contents;<br>Set to null | | Alter time stamp |
| insertionPoint | Set to > #auditTrail | | Eliminate sorting;<br>Alter sorting performance;<br>Continuous set to #auditTrail |
| auditTrail | Remove records | | Reorder records |

While the process control flow is the same for the CPN created for R4.2's VFlow, the internal computation performed at the transitions to determine which VCs match and the impact estimate of that match can introduce different risk values.

The VFlow representing the meta-data from the proof process for R4.2 involves all five processes in the SIMS case study, which will be equally impacted by changes affecting their VCs. Figure 24 shows the CPN representing R4.2's VFlow. Notice that the pink token is associated with findInsertionPoint as it was in Figure 22. The same VC, insertionPoint, is affected, but the affect is less (unconcerned) than for R5.

### 3.2.4.7 Comparing Adaptation Risk

We use a previously derived utility function to compare adaptation plans A1-A3 according to the assessed risk they pose to inhibiting the reuse of the verification processes for R4 and R5. The success probability of an adaptation plan $a$ with respect to a requirement $r$ is estimated as $\prod_{t \in T(r,a)} p(t)$, where $T(r,a)$ is the set of red tokens generated from $r$'s VFlow for adaptation plan a and $p(t)$ is an estimate of the probability that token $t$ does *not* represent an actual violation of a verification process. We consider probability scaling, where

$$p(t) = M_{PL}(t) M_{VC}(t) \hat{p}(t)$$

with $M_{PL}(t)$ a risk factor associated with changing a process, $M_{VC}(t)$ a risk factor associated with the VC and a verification condition, and $\hat{p}(t)$ the quality that the planning process associates with the adaptation plan. These values all appear in each red token $t$. The expected utility of an adaptation plan is

$$E[U(a)] = \sum_{r \in R} \left( w(r) \prod_{t \in T(r,a)} p(t) \right)$$

where $w(r)$ is the impact weight applied to a requirement $r$.

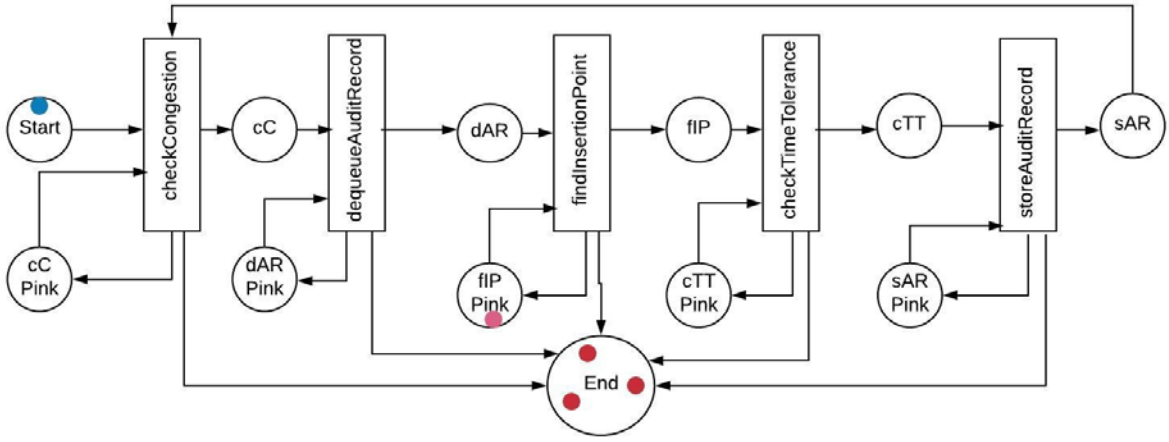



**Figure 24: VFlow for R4.2 and Adaptation A2**

Comparing the potential risk of adaptation on the reuse of the proof process means providing values for the impact factors for $p(t)$. Starting with $M_{VC}$, we assign VC conditions that match the plan and are devastating (0.2 impact factor), worrisome (0.5 impact factor), and unconcerned (0.9 impact factor), where a higher impact factor has less impact. The lowest impact factor is chosen if more than one condition matches. If there is no match, the $M_{VC} = 1$.

For adaptation A1, auditRecord's devastating condition "Set to null" is matched for R4.2 (**Table 12**, $M_{VC} = 0.2$), but is only worrisome for R5 (Table 11, $M_{VC} = 0.5$), since time-correlating the auditTrail is unaffected, but dropping is problematic overall. For adaptation A2, insertionPoint's devastating condition "Continuous set to #auditTrail" is matched for R5 with $M_{VC} = 0.2$. For R4.2, this same condition is rated at unconcerned so its $M_{VC} = 0.9$. For Adaptation A3, the match is with insertionPoint's unconcerned condition "Alter sorting performance", making $M_{VC} = 0.9$ for both

R4.2 and R5. Since the impact on the processes is the same, $M_{PL} = 0.5$. The AU-12 requirements are representative of the baseline security control and valued similarly. Therefore, we set w(r) = 1 for all requirements r. The planning process rates the quality of the plan. For A1: $\hat{p} = 0.25$, A2: $\hat{p} = 0.6$, A3: $\hat{p} = 0.75$.

Using the VFlows associated with requirements R4.2 and R5, the expected utility of A1-A3 is calculated as follows from the resulting red tokens:

$$E[U(A1)] = (0.25 * 0.2 * 0.5)^3 + (0.25 * 0.2 * 0.5)^3 = 2.60 \times 10^{-4}$$
$$E[U(A2)] = (0.6 * 0.9 * 0.5)^3 + (0.6 * 0.9 * 0.5)^3 = 1.99 \times 10^{-2}$$
$$E[U(A3)] = (0.75 * 0.9 * 0.5)^3 + (0.75 * 0.9 * 0.5)^3 = 7.69 \times 10^{-2}$$

Thus, adaptation plan A3 has the highest utility and is the least risky because of its plan to increase the performance of the sorting technique by providing a rolling lower timestamp and sorting only records less than the timestamp. As expected, A1's plan to allow Audit to periodically drop messages is the riskiest.

### 3.2.5 Deploying the Framework

#### 3.2.5.1 The Wearable Security Testbed

In order to embed formal verification on a physical device, an existing self-adaptive device is needed. We used an in-house wearable security testbed that relies on Raspberry Pi 3 Model Bs to simulate both the wearables and base stations. The testbed communicates between devices using Bluetooth, a common communication protocol that has been shown to be vulnerable to attacks (Walter, 2018b). The testbed allows developers and researchers to program directly on the device, with direct control of the Bluetooth communication. We input the Raspberry Pis with a MAPE-K (monitor, analyze, plan, execute, knowledge) loop for monitoring the data communication, analyzing the packets being sent, planning an adaptation method, and executing the optimal adaptation option. Each adaptation is designed to prevent an attacker from performing an attack. We start with providing to the planner 2 potential adaptations should a vulnerability be detected by wearable or base station.

- *Send empty data packets*, preventing an attacker from eavesdropping on data communication or beginning a Man-in-the-Middle attack. Because the base station remains connected, it is possible for the wearable to override this adaptation, allowing an attacker to gain data through eavesdropping.
- *Disconnect and await reconnection*, preventing an attacker from eavesdropping but potentially allowing a Man-in-the-Middle attack. Because the devices are disconnected, there is a risk of a loss of data if the base station and wearable do not reconnect.

The testbed uses an in-house developed self-awareness process, called *fostering*, to handle data communication that was also pushed to the control of the MAPE-K loop. Fostering allows a

wearable that has become disconnected from its base station to learn of potentially insecure environments by temporarily (for only a few packets) connecting to other devices in the area also running the self-awareness application. These devices exchange no more than 2 data packets containing the current security state of the area before disconnecting. The existence of fostering gives rise to 2 additional adaptation options, for a total of 4 potential adaptive plans that are accessible by the planner.

**A1.** Send empty data packets and disallow fostering
**A2.** Disconnect and disallow fostering
**A3.** Send empty data packets and allow fostering
**A4.** Disconnect and allow fostering

#### 3.2.5.2 Embedding Verification on Wearables

To prototype the algorithm for the CPN that expresses our verification contract with each requirement in the wearable devices, we use the wearable testbed to simulate three self-adaptive wearables.

- Heart Rate Variability Monitor (HRVM), which uses captured heart rate data over a known period of time to determine the amount of stress a user is under
- Hearables, wearables which are designed to be worn in the ear, streaming audio data from and accelerometer data to a base station
- Insulin Pump, a medical grade wearable that tracks a user's blood glucose level and receives instructions about when to administer insulin from a base station

##### 3.2.5.2.1 Heart Rate Variability Monitor

To formally define the HRVM, we created pseudocode describing the general operation of the wearable (Figure 25). From this pseudocode, we extract critical functional requirements of the wearable.

HRVM1. The buffer does not overflow, leading to a loss of heart rate data
HRVM2. Data is not lost when the wearable has determined the wearer is in a stressed state

To show these requirements are never violated, we formally prove them using the pseudocode. Figure 26 shows the proof outline.

```
i ← 0
N ← length(buffer)
L ← length(buffer)
while (true)
do
   i++                                                    //S1
   hr_rate_info ← read(port1)
   buffer[i] ← hr_rate_info
   connected_device_sync_request ← read(port2)
   stressed ← computeStressLevel(buffer)
   if (connected_device in list_of_valid_devices)
      then
         if (connected_device_sync_request) or (i mod N == 0 and connected) //S2 and S3
            then
               send(buffer)
               i ← 0
               buffer_change ← false
               N ← L
         else
            if (connected and streaming) //S4
               then
                  send(hr_rate_info)
                  i ← 0
                  buffer_change ← false
            else
               buffer_change ← true //S5
         if (stressed and i == N)  //S6
            then
               increaseBufferSize()
               buffer_change ← true
```

**Figure 25: Pseudocode for the heart rate variability monitor**

**Proof Sketch for HRVM1**

S1: i increases only if i < N (Read)

S2-S4: connected, i is set to 0 after send and buffer length is reset (Reset)

S5: not connected, i is set to 0 (Rewrite)

S6: not connected, N is increased (Increase)

Therefore, i is always at most N.

**Proof Sketch for HRVM2**

Assume R1: **Inv** ($i \leq N$)

S1: next hr_rate_info is always added to the buffer and is not lost (Read)

S2 – S4: connected **leads-to** buffer send (so data in buffer is not lost) and reset to add next hr_info into available spot (Reset)

S5: data is lost, but stressed is not determined (Rewrite)

S6: not connected and stressed leads-to buffer length increase to add next hr_info (so data in buffer is not lost) (Increase)

Therefore, when a read is performed in a stressed state, the read is eventually stored in the buffer.

**Figure 26: Proof outline for heart rate variability monitor**

From the proofs we extract VCs and formulate the VConds, as in Tables Table 13 and Table 14 for each requirement. For requirement HRVM1, the VCs are the state variables *i* (representing the place in the buffer which data is read in to), *N* (the size of the buffer), *connected_device* (the device that the HRVM is connected to), and *send* (the ability of the HRVM to send data to its base station). Requirement HRVM2 has 6 VCs. The first 4 are the same VCs as requirement 1. The final 2 are *read* (the ability of the HRVM to read data from the base station and from its own sensors) and *stressed* (the value representing the stressed state of the user.

The VConds for Devastating, Worrisome, and Unconcerned are in Table 13 for requirement HRVM1 and Table 14 for requirement HRVM2. Recall from the last report that changes which are in the Unconcerned set have a very small chance of affecting the original proof reuse, while changes in the Devastating set have a very large chance of affecting the proof reuse.

**Table 13: Verification concern condition table for Requirement HRVM1**

| $\mathbf{M_{vc}}$ | | | |
|---|---|---|---|
| **HVRM1** | **Devastating (0.2)** | **Worrisome (0.5)** | **Unconcerned (0.9)** |
| *I* | Remove $i \leftarrow 0$, set $i > N$, increase *i* by more than 1 per state change, perform $i \leftarrow 0$ in a different location | Set $i < 0$, decrease *i* by more than 1 per state change | |
| *N* | Reduce *N*, reset *N* where is it not currently performed | Increase *N* above *L* | Increase *N* where is it not currently performed |
| *connected* | Inhibit connection | Disconnect without altering *N* | |
| *Send* | Inhibit *send* while connected, Expect *send* while disconnected | | Sending null data (empty packets) |

**Table 14: Verification concern condition table for Requirement HRVM2**

| $M_{vc}$ | | | |
|---|---|---|---|
| **HVRM2** | **Devastating (0.2)** | **Worrisome (0.5)** | **Unconcerned (0.9)** |
| *I* | Any change to *i* | | |
| *N* | Reducing *N* when disconnected | Increase *N* above *L* | Increase *N* where is it not currently performed |
| *connected* | Inhibit connection | Disconnect without altering *N* | |
| *Read* | Inhibiting read | Adding more reads without buffer and send accommodations | |
| *stressed* | Adding or removing stress from a guard | | |
| *Send* | Inhibit *send* while connected, Expect *send* while disconnected | | Sending null data (empty packets) |

From the expected utility function shown in Equation 1, w(r) to be the weight of the requirement. These weights are assigned by the developer based on the flexibility of the requirement to the overall function of the wearable. Lower values mean the requirement is rigid in its acceptance of change. Thus, the lower the value the more likely inhibiting proof reuse occurs, which is represented by any alteration to the verification contract, affecting the risk of the adaptation. For the HRVM, we assign a higher weight to requirement HRVM1. Requirement HRVM1, not allowing the buffer to overflow, allows for some flexibility in how all stress data is stored. In the case of a full buffer, overwriting old data will still allow stress data to be collected and stored, thus fulfilling the primary purpose of the HRVM, though it does mean that individual heart rate data will be lost, leading to a more rigid requirement HRVM2. The weights are listed below.

- w(HVRM1) = 0.75
- w(HVRM2) = 0.35

The pseudocode in Figure 25 provides a description of the process flow diagramed in Figure 27, which is translated into the VFlow used by the Colored Petri Net (CPN) algorithm and implemented within the simulated HRVM wearable on the testbed. We assign impact factors to each process in the flow, to represent the risk of a change to that process as it affects proof reuse. For the HRVM, we assign a value of $M_{PL}$=0.5 to all processes, indicating a moderate impact for both requirements.

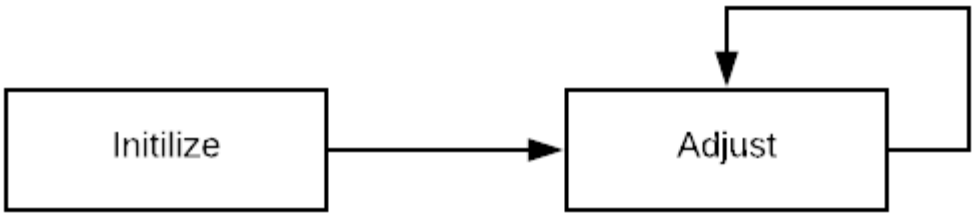



**Figure 27: Process flow for the HRVM**

Because the plans are defined the same for each wearable and are statically placed in the planner, their impact value, ($\hat{p}$), is provided based on the security and social situational awareness each plan provides. These weights are:

- $\hat{p}(A1) = 0.55$
- $\hat{p}(A2) = 0.6$
- $\hat{p}(A3) = 0.65$
- $\hat{p}(A4) = 0.75$

With all these values determined, the planner is capable of calculating the expected utility of the adaptations. Equation 1 shows the expected utility function and Table 15 shows the results of calculating the expected utility of the adaptations. For the HRVM, the least risky adaptation is A1, *sending empty data packets without allowing fostering*. This status is because it changes the fewest VCs, only affecting the *send* by sending empty packets in both requirements HRVM1 and HRVM2. Adaptation A3, *send empty packets with fostering*, is not chosen because, when fostered, the send must be prohibited. This status is because connecting to another device while fostering requires the HRVM to not send data even though it is connected, resulting in a devastating value for $M_{VC}$.

$$\boldsymbol{E}[\boldsymbol{U}(\boldsymbol{a})] = \sum_{\boldsymbol{r}\in\boldsymbol{R}}\left(\boldsymbol{w}(\boldsymbol{r})\prod_{\boldsymbol{t}\in\boldsymbol{T}(\boldsymbol{r},\boldsymbol{a})}\boldsymbol{P}(\boldsymbol{S}(\boldsymbol{t})=\boldsymbol{1})\right) \quad (1)$$

**Equation 1: Equation used to determine the expected utility of the adaptation**

**Table 15: Calculation results for HRVM**

| a | r | Mvc | Mpl | p̂ | P(S(t)=1) | w(r) | w(r) * P(S(t=1)) | E[U(a)] |
|---|---|---|---|---|---|---|---|---|
| A1 | R1 | 0.9 | 0.5 | 0.55 | 0.2475 | 0.75 | 0.185625 | 0.27225 |
| | R2 | 0.9 | 0.5 | 0.55 | 0.2475 | 0.35 | 0.086625 | |
| A2 | R1 | 0.5 | 0.5 | 0.6 | 0.15 | 0.75 | 0.1125 | 0.165 |
| | R2 | 0.5 | 0.5 | 0.6 | 0.15 | 0.35 | 0.0525 | |
| A3 | R1 | 0.9 | 0.5 | 0.65 | 0.2925 | 0.75 | 0.219375 | 0.242125 |
| | R2 | 0.2 | 0.5 | 0.65 | 0.065 | 0.35 | 0.02275 | |
| A4 | R1 | 0.5 | 0.5 | 0.75 | 0.1875 | 0.75 | 0.140625 | 0.166875 |
| | R2 | 0.2 | 0.5 | 0.75 | 0.075 | 0.35 | 0.02625 | |

An example of the VFlow as represented by the CPN, which runs directly on the HRVM, can be seen in Figure 28. Note that the CPN is the same for each of the 4 adaptations. The pink tokens contain information about the VCs and VConds, the blue tokens contain information about the adaptation examination through the VFlow, and the red tokens are the alert values used by the planner to determine the expected utility of the adaptation.

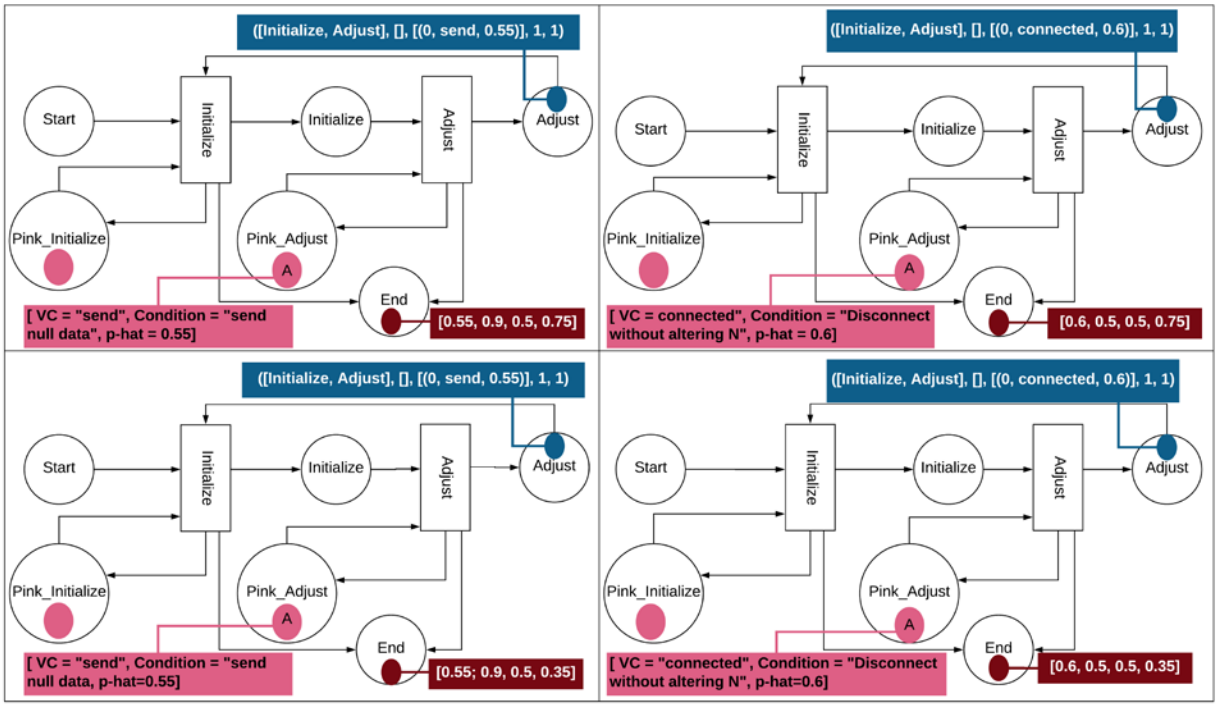



**Figure 28: The Colored Petri Net for the HRVM running on the wearable**

#### 3.2.5.2.2 Hearables

To formally define the hearables, we created pseudocode describing the general operation of the wearable (Figure 29). From this pseudocode, we extract critical functional requirements of the wearable.

Hear1. Music may be streamed from any connection
Hear2. The buffer may only be sent on an authorized connection
Hear3. Accelerometer data is always collected and temporarily stored

To show these requirements are never violated, we formally prove them using the pseudocode. Figure 30 shows the proof outline.

```
i ← 0
N ← length(buffer)
buffer_change ← false
while (true)
do
   i++
   accelerometer_info ← read(port1)
   buffer[i] ← accelerometer_info
   if (any_connection)
      then
      playMusic(read(port2))
   else
      if (auth_connection and i > 0 and buffer_change)
      then
         send(buffer[1..i])
         i ← 0
         buffer_change ← true
      else
         if ( i  == N and buffer_change)
            then
            i ← 0
            buffer_change ← true
         else
            buffer_change ← false
```

**Figure 29: Pseudocode for the hearable**

**Proof Sketch for Hear1**
S1: not affected (Read)
S2: Music is read and played if there is any connection (Play)
S3 & S4 not affected (Adjust)
Therefore, when there is a connection, music is played

**Proof Sketch for Hear 2**
S1: accelerometer is read and added to the buffer
S2: not affected
S3: Buffer is sent, up to i, only when connected to an authorized connection
S4: If no connection and buffer is full, overwrite the buffer
Therefore, the buffer is only sent when connected to an authorized connection

**Proof Sketch for Hear 3**
S1: accelerometer data is read and input into the buffer
S2: not affected
S3: Buffer is sent to base station, i is set to 0 to reset the buffer, and buffer_change is set to true
S4: Buffer is full and cannot be sent, i is set to 0 to reset the buffer, and buffer_change is set to true
Therefore, accelerometer data will always be collected and stored

**Figure 30: Proof outline of hearable requirements**

From the proofs, we extract VCs and craft VConds as shown in Table 16-Table 18 for each requirement. For requirement Hear1, the verification concerns are *any_connection* (used to determine if there is a connection of any type), *playMusic* (used to play music to the wearer), and *read* (used to read the music from the base station or the accelerometer data). Requirement Hear2's VCs are *i* (representing the place in the buffer which data is read into), *send* (the function to send accelerometer data to the base station), *read* (used to read music from the base station or the accelerometer data), and *auth_connection* (used to determine if the connection is an authorized connection). For requirement Hear3, the VCs are *i*, *buffer_change* (used to recognize that the buffer has been changed in some way), *send*, and *read*.

**Table 16: Verification concern condition table for hearables requirement Hear1**

| $M_{vc}$ | | | |
|---|---|---|---|
| **Hear1** | **Devastating (0.2)** | **Worrisome (0.5)** | **Unconcerned (0.9)** |
| *Any_connection* | | Set to 0 when connected | Set to 1 when not connected |
| *playMusic* | Inhibiting *playMusic* | | |
| *Read* | Inhibiting *read* | *Read* blocks due to no incoming data | |

**Table 17: Verification concern condition table for hearables requirement Hear2**

| $M_{vc}$ | | | |
|---|---|---|---|
| **Hear2** | **Devastating (0.2)** | **Worrisome (0.5)** | **Unconcerned (0.9)** |
| *i* | *i* > *N* | *i* not reset when needed | *i*=0 |
| *Send* | Inhibit *send* while connected to authorized device | Expect *send* while disconnected | Send null data |
| *Read* | Inhibit *read* | | |
| *Auth_connection* | Set to 1 when not connected to authorized device | Set to 0 when connected to authorized device | Set to 0 when not connected to authorized device |

**Table 18: Verification concern condition table for hearables requirement Hear3**

| $M_{vc}$ | | | |
|---|---|---|---|
| **Hear3** | **Devastating (0.2)** | **Worrisome (0.5)** | **Unconcerned (0.9)** |
| *i* | *i*>*N* | *i* not reset when needed | *i*=0 |
| *Buffer_change* | Set to false when change has been made | Set to true when no change made | |
| *Send* | Inhibit *send* while connected to authorized device | Expect *send* while disconnected | Send null data |
| *Read* | Inhibit *read* | | |

For w(r), we assign a lower weight (more rigidity) to requirement Hear1, since playing music is the primary purpose of the hearables. Requirement Hear3 has more flexibility for change but is more rigid than requirement Hear2 as accelerometer data always being collected is important for hands-free control of the hearables and the music playing features. It is not a problem if the buffer is not sent on an authorized connection, as overflowing the buffer does not cause a problem for the hearables. The weights are listed below.

- w(Hear1) = 0.35
- w(Hear2) = 0.75
- w(Hear3) = 0.5

For the hearables, the process flow is very similar to the HRVM flow shown in Figure 27. However, there is the addition of a 'Play' process after the 'Adjust' process to account for the need to play music. We assign a value of $M_{PL}$=0.5 for 'Initialize' and 'Adjust', indicating a moderate impact for the requirements, and a value of $M_{PL}$=0.3 for 'Play,' indicating a major restriction on changes to that process.

We use the same values for $\hat{p}$ as in the HRVM for the static adaptations. These retained values force the wearable to use its verification awareness directly on determining the least risky plan, illustrating the differences that verification awareness makes. Table 19 shows the results of calculating the expected utility of the adaptations. For the hearables, the least risky adaptation is A4, *disconnecting and fostering*. This status is because, for the sending empty packets adaptations, remaining connected requires the inhibiting of the read for Hear2 and Hear3, as any connection

with the hearables base station will result in the reading of music data from the base station to be inhibited to maintain the empty packet solution.

**Table 19: Calculation results for the hearables**

| a | r | Mvc | Mpl | p̂ | P(S(t)=1) | w(r) | w(r) * P(S(t=1)) | E[U(a)] |
|---|---|---|---|---|---|---|---|---|
| A1 | R1 | 0.5 | 0.5 | 0.55 | 0.1375 | 0.35 | 0.048125 | 0.116875 |
| | R2 | 0.2 | 0.5 | 0.55 | 0.055 | 0.75 | 0.04125 | |
| | R3 | 0.2 | 0.5 | 0.55 | 0.055 | 0.5 | 0.0275 | |
| A2 | R1 | 1 | 0.5 | 0.6 | 0.3 | 0.35 | 0.105 | 0.4425 |
| | R2 | 0.9 | 0.5 | 0.6 | 0.27 | 0.75 | 0.2025 | |
| | R3 | 0.9 | 0.5 | 0.6 | 0.27 | 0.5 | 0.135 | |
| A3 | R1 | 0.5 | 0.5 | 0.65 | 0.1625 | 0.35 | 0.056875 | 0.138125 |
| | R2 | 0.2 | 0.5 | 0.65 | 0.065 | 0.75 | 0.04875 | |
| | R3 | 0.2 | 0.5 | 0.65 | 0.065 | 0.5 | 0.0325 | |
| A4 | R1 | 0.9 | 0.5 | 0.75 | 0.3375 | 0.35 | 0.118125 | 0.54 |
| | R2 | 0.9 | 0.5 | 0.75 | 0.3375 | 0.75 | 0.253125 | |
| | R3 | 0.9 | 0.5 | 0.75 | 0.3375 | 0.5 | 0.16875 | |

#### 3.2.5.2.3 Insulin Pump

To formally define the insulin pump wearable, we created pseudocode describing the general operation of the wearable (Figure 31). From this pseudocode, we extract critical functional requirements of the wearable.

IP1. The buffer may never be full when blood sugar level is high
IP2. The insulin pump only connects to authorized base stations
IP3. Insulin is administered when blood sugar levels are too high

To show these requirements are never violated, we formally prove them using the pseudocode. Figure 32 shows the proof outline.

```
i ← 0
N ← length(buffer)
L ← N
D ← max_blood_sugar_allowed
while (true)
do
   i++
   blood_sugar_info ← read(port1)
   administer_insulin ← read(port2)
   buffer[i] ← computeBloodSugar(blood_sugar_info)
   if ( connected and auth_connection and i < N )
      then
         send(buffer[1..i])
         i ← 0
         N ← L
   else
      if ( i == N and bloodSugarLevel(buffer) ≤ D)
         then
            i ← 0
            N ← L
      else
         if ( i == N)
            then
               N ← increase(N)
   if ( bloodSugarLevel(buffer) ≥ D or administer_insulin )
      then
         administerInsulin()
```

**Figure 31: Pseudocode for the insulin pump**

**Proof sketch for IP1:**
S1: i increases only if i < N (read)
S2: i is set to zero after send and buffer length is reset (reset)
S3: i is set to zero and buffer length is reset (rewrite)
S4: not connected, N is increased (increase)
S5: no change to buffer (Administer)
Therefore, i is always at most N

**Proof sketch for IP 2:**
S1: no change
S2: send is only called if connected to an authorized device
S3-S5: not applicable for authorized connections
Therefore, i will only connect to authorized base stations

**Proof sketch for IP 3:**
S1: Reads if it needs to administer insulin from the base station
S2-S4: no change to insulin
S5: Insulin is administered if base station is connected the pump to administer or if blood sugar level is too high
Therefore, insulin is administered when blood sugar level is too high

**Figure 32: Proof outline of the insulin pump requirements**

From these proofs, we extract the VCs and craft the VConds associated with each requirement and shown in Table 20-Table 22. For requirement IP1, the VCs are *i* (representing the place in the buffer which data is read into), *N* (the size of the buffer), *connected* (representing if the insulin pump is connected to a base station), and *send* (the function to send information to the base station). Requirement IP2's verification concerns are *auth_connection* (used to determine if a connection is an authorized connection) and *connected*. For requirement IP3, the verification concerns are *read* (used to read both the current blood sugar levels and data streamed from the base station), *administer_insulin* (used to administer insulin directly to the wearer), *bloodSugarLevel* (a calculation which determines the blood sugar level of the wearer), and *D* (the maximum safe blood sugar level for the wearer).

**Table 20: Verification concern condition table for IP1**

| $M_{vc}$ | | | |
|---|---|---|---|
| **IP1** | **Devastating (0.2)** | **Worrisome (0.5)** | **Unconcerned (0.9)** |
| *i* | Remove $i := 0$, set $i > N$, increase *i* by more than 1 per state change, set *i* to 0 where it is not currently performed | Set $i < 0$, decrease *i* by more than 1 per state change | |
| *N* | Reduce N, reset N to L where is it not currently performed | Increase *N* above *L* | Increase *N* where is it not currently performed |
| *connected* | Inhibit connection | Disconnect without altering *N* | |
| *send* | Inhibit *send* while connected, Expect *send* while disconnected | | Sending null data (empty packets) |

**Table 21: Verification concern condition table for IP2**

| $M_{vc}$ | | | |
|---|---|---|---|
| **IP2** | **Devastating (0.2)** | **Worrisome (0.5)** | **Unconcerned (0.9)** |
| *Connected* | Set to true when connected to non-authorized device | Set to false when connected to an authorized device | |
| *Auth_connection* | Set to true when connected to non-authorized device | Set to false when connected to authorized device | |

**Table 22: Verification concern condition table for IP3**

| $M_{vc}$ | | | |
|---|---|---|---|
| **IP3** | **Devastating (0.2)** | **Worrisome (0.5)** | **Unconcerned (0.9)** |
| *Read* | Inhibit *read* | Read null data | |
| *Administer_insulin* | Set to false when instructed to administer | Set to true when not told to administer | Unable to set *administer_insulin* |
| *bloodSugarLevel* | Prevent processing | | |
| *D* | Value of *D* raised | Value of *D* lowered | |

For w(r) we assign a lower weight to requirement IP3, since administering insulin is the primary purpose of the insulin pump. Requirements IP1 and IP2 are given the same weights. Both the buffer never being full in a high blood sugar state and the sending of the buffer to only authorized devices are equally important for the insulin pump. The weights are listed below.

- w(IP1) = 0.75
- w(IP2) = 0.75
- w(IP3) = 0.5

The pseudocode for the insulin pump is very similar to the flow for the HRVM shown in Figure 27. However, there is an addition of a ‘Administer’ process after the ‘Adjust’ process to account for the additional need to provide insulin if the wearer is in a dangerous state. We assign impact

factors to each process as follows: $M_{PL}$=0.5 for 'Initialize' and 'Adjust', indicating a moderate impact for the requirements, and $M_{PL}$=0.1 for 'Administer,' indicating a major impact for the requirement.

We use the same values for $\hat{p}$ as in the HRVM, as the adaptations remain static from the perspective of the planner. Table 23 shows the results of calculating the expected utility of the adaptations. For the insulin pump, the least risky adaptation is A2, disconnecting without fostering. The insulin pump is capable of providing insulin in an emergency without input from its base station. Neither adaptation A3 nor adaptation A4 are chosen because fostering will cause *connected* to become true when connected to an unauthorized device to foster. In this case, both adaptation A1 and A2 are equally risky, so A2 is chosen based primarily on its higher $\hat{p}$ value.

**Table 23: Calculation results for the insulin pump**

| a | r | Mvc | Mpl | $\hat{p}$ | P(S(t)=1) | w(r) | w(r) * P(S(t=1)) | E[U(a)] |
|---|---|---|---|---|---|---|---|---|
| A1 | R1 | 0.9 | 0.5 | 0.55 | 0.2475 | 0.75 | 0.185625 | 0.460625 |
| | R2 | 1 | 0.5 | 0.55 | 0.275 | 0.75 | 0.20625 | |
| | R3 | 0.5 | 0.5 | 0.55 | 0.1375 | 0.5 | 0.06875 | |
| A2 | R1 | 0.5 | 0.5 | 0.6 | 0.15 | 0.75 | 0.1125 | 0.4725 |
| | R2 | 1 | 0.5 | 0.6 | 0.3 | 0.75 | 0.225 | |
| | R3 | 0.9 | 0.5 | 0.6 | 0.27 | 0.5 | 0.135 | |
| A3 | R1 | 0.9 | 0.5 | 0.65 | 0.2925 | 0.75 | 0.219375 | 0.414375 |
| | R2 | 0.2 | 0.5 | 0.65 | 0.065 | 0.75 | 0.04875 | |
| | R3 | 0.9 | 0.5 | 0.65 | 0.2925 | 0.5 | 0.14625 | |
| A4 | R1 | 0.5 | 0.5 | 0.75 | 0.1875 | 0.75 | 0.140625 | 0.365625 |
| | R2 | 0.2 | 0.5 | 0.75 | 0.075 | 0.75 | 0.05625 | |
| | R3 | 0.9 | 0.5 | 0.75 | 0.3375 | 0.5 | 0.16875 | |

The implementation exercise reported this quarter allowed us to assess both efficacy and performance of the algorithm. We knew that none of the CPN tools available would provide what we needed because of their lack of APIs. However, we needed the algorithm to respond rapidly with the risk assessment results, which was achieved. The use of static plans was a first step in making the direct change to the code. Additional infrastructure will need to be embedded into the component to accommodate dynamic changes at runtime from the coding standpoint. We believe the algorithm for risk assessment will remain viable.

### 3.2.6 Designing and Evolving Security Assurance Cases within the Framework

#### 3.2.6.1 Representing Security Controls as Assurance Cases

The NIST SP800-53(NIST, 2013) has become the de facto standard for security compliance best practices to protect information confidentiality, integrity, and availability. The document details 18 security control families that house security requirements with which companies working with the US government must demonstrate compliance. A self-adaptive information system complying with designated security controls means (1) compliance must be guaranteed at an expected confidence level, (2) mechanisms deployed to enable security controls should not be deleted as part of an adaptation, and (3) the system should have awareness of its security controls, mechanisms enabling their effectiveness, and dependencies between controls to reduce conflict and change propagation effects.

Figure 33 shows the SC-8 Transmission Confidentiality and Integrity security control. The "SC" stands for the control family "System and Communications Protection". Figure 33 includes, as an example, one of the four control enhancements to SC-8. In SC-8, the information system must have a functional mechanism to perform the selected protections (e.g. confidentiality and/or integrity). Omitted from Figure 33 are the supplemental guidance statements that provide an overview of the control and enhancement.

Other aspects of the security control statement that are relevant for system security awareness are the Selection/Assignment blocks and Related controls. The appearance of [*Selection/Assignment:* …] in a control statement provides the organization with the opportunity to explicitly tailor the controls across the entire organization, certain segments of the organization, or specific information systems. Related controls list external controls with which there exists an interdependency, though it may depend on the instantiated values imposed by tailoring.

The challenge to providing security awareness to a self-adaptive system is three-fold: (1) the security control must be expressed in a way that separates, yet captures the statement information, (2) how compliance is guaranteed in the system should be made explicit but at different abstraction levels, and (3) change must be introduced into the representation in such a way that its effect is understood and assessable. To address the security awareness challenge, we rely on assurance case concepts. Because the NIST SP800-53 controls generally follow the statement structure in Figure 33, we define a security assurance case template, shown in Figure 34, using Goal Structuring Notation (GSN) that expresses the pattern of the security control while allowing for compliance guarantees to be represented as claim or goal arguments. A security assurance case can instantiate the template by providing the values of the parameters indicated by curly braces.

**SC-8: TRANSMISSION CONFIDENTIALITY AND INTEGRITY**

Control: The information system protects the [*Selection (one or more): confidentiality; integrity*] of transmitted information.

Related controls: AC-17, PE-4.

Control Enhancements:

(1) TRANSMISSION CONFIDENTIALITY AND INTEGRITY | CRYPTOGRAPHIC OR ALTERNATE PHYSICAL PROTECTION

The information system implements cryptographic mechanisms to [*Selection (one or more): prevent unauthorized disclosure of information; detect changes to information*] during transmission unless otherwise protected by [*Assignment: organization-defined alternative physical safeguards*].

**Figure 33: SC-8 Security Control Statement with Enhancement SC-8(1)**

In Figure 34, Goals are represented as rectangles. The security control is the main assurance case Goal, which can have 0 or more enhancements that depend on it shown by the SupportedBy link (filled arrow). It is dependent on 0 or more related controls. Context nodes (ovals) attached to goals through the InContextOf link (hollow arrow) provide environment and state information. The main Goal can have 0 or more tailoring Context nodes to express selection and/or assignment. In GSN, the argument that supports the claim extends from an assessment Strategy (parallelogram) that may contain subgoals on which the argument depends. These subgoals may be defined in lower level Modules (folder shape), such as process-specific operational goals related to an assessment method. For security controls, assessment methods can include examination, model checking, requirements verification, and testing. The assessments provide the evidence needed for the Solution (circle). We introduce an additional Context node as a numeric measure [0..1] of how flexible the satisfaction of the goal is to the full certification effort, which may be based on its singular importance or its interdependencies that could negatively impact certification through the propagation of compliance violation. A triangle associated with a GSN node means the node is abstract or uninstantiated. The joined triangles mean the node is both undeveloped and uninstantiated. The impact baseline allocation is provided. The “provides” attribute holds the *provision set* of state variables and conditions that are part of the mechanisms needed for compliance with the security control. This set flows through a SupportedBy link that is augmented with a diamond to indicate the security control source for the provision set. In Figure 34, provision sets flow to the main control from related controls and enhancements. The achievement weight, $a_w$, is assigned to all goals. It holds the current value calculated at the goal for assessing the satisficing level of the main goal as discussed in Section 3.2.7.5.

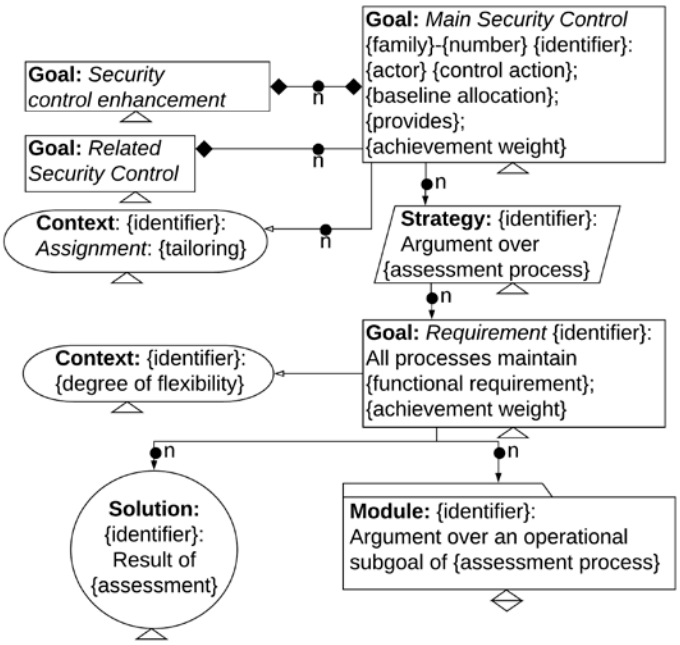



**Figure 34: GSN Template for a Security Assurance Case**

### 3.2.6.2 Case Study using Smart Inventory Management System (SIMS)

To demonstrate our security assurance case adaptation approach, we use our Smart Inventory Management System (SIMS) discussed in Section 3.2.2. We impose the SC-8 security control from Section 3.2.6.1 in which an information protection mechanism has been implemented at Measure and Process for secure transmission.

Suppose that during execution, the SIMS Audit MAPE-K loop monitor detects that an incoming transmission channel is malfunctioning. Analysis determines a correction is needed, triggering the planner to generate potential adaptations, such as:

A1: *Disable confidentiality service protecting channel.* There may be a conflict between the integrity and confidentiality protection services requiring correction. The organization places a higher priority on integrity.

A2: *Store data locally and transmit data in a batch when channel is restored.* Shutting down the channel temporarily would not compromise protection.

A3: *Activate and perform transmission through another channel.* Activate redundant channels with their own protection services according to priority use, to allow deeper analysis of the original channel malfunction.

Figure 35 instantiates the security assurance case template (Figure 34) for SC-8 given the SIMS behavior expectations. The enhancement SC-8(1) is a GSN Away Goal dependent on SC-8's compliance guarantee. SC-8 depends on related controls AC-17 and PE-4, also notated as Away Goals. The Selection tailoring appears in Context nodes C1 and C2, showing that both confidentiality and integrity should be protected in transmission. SC-8 has a moderate flexibility value (0.60 in Context node C3). Since it appears in the related controls of many other controls, a

change could impact the overall security certification. The argument Strategy (S1) to satisfy SC-8 relies on a module (M1) for the argumentation of the transmission process.

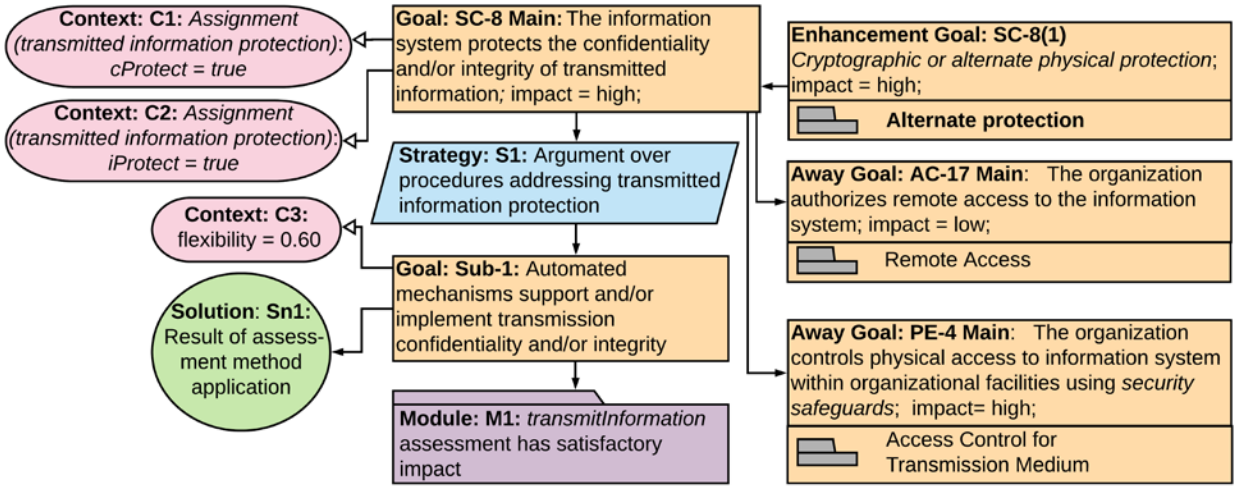



**Figure 35: Security Assurance Case for SC-8**

Figure 36 shows the expanded module M1's contents for the transmitInformation process. The top-level goal of the module is that transmitInformation has a satisfactory impact on the assessment process, requiring guarantees with three operational goals that check the protection services (OpGoal G-1), check the channel used (OpGoal G-2), and ensure that the transmission is performed without interruption (OpGoal G-3).

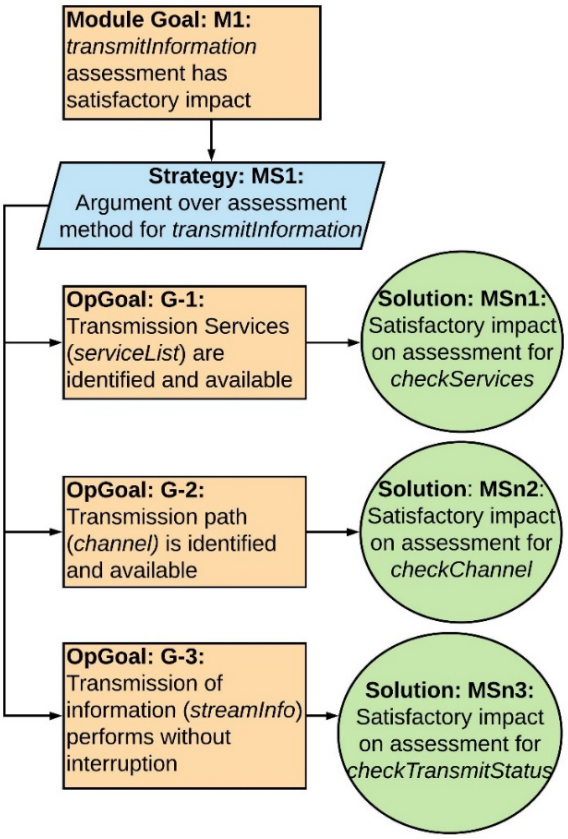



**Figure 36: Expanded GSN Module for transmitInformation**

### 3.2.6.3 Adapting Assurance Cases

Given that functional adaptations can affect security control compliance, adaptations configured by the MAPE-K loop should be reflected in related security assurance cases so that confidence and risk levels of the adaptations can be assessed. We assume that the planner can represent the parameters of the adaptation as a set of tuples, ChangeSet, formulated around state variables affected by the change and understood by an Adaptation Operator Manager (AOM) within the planner that oversees the security assurance cases. The AOM currently considers two high-level types of adaptations: (1) those that make a direct change to the state variables within the code, and (2) those that indirectly alter behavior of the state variables by modifying the code. For type (2), it considers two subcategories: (i) those that introduce new functionality that may include new state variables in support of the system's goals, and (ii) those that replace functionality with new functionality relying only on existing state variables. To accommodate the AOM, the ChangeSet includes state (contextual) and process changes, evidence for ensuring functional accuracy, and a rationale for the adaptation, as follows:

$$\begin{aligned} &\textit{ChangeSet} = \\ &\{(\textit{stateVar}, \textit{newState}, \textit{changeCond}, \textit{evidence}, \textit{rationale})_1, \ldots, \\ &\ (\textit{stateVar}, \textit{newState}, \textit{changeCond}, \textit{evidence}, \textit{rationale})_N\} \end{aligned}$$

where *stateVar* is an existing variable to be changed to *newState*, *changeCond* constrains *stateVar* in the adapted system, evidence is the available argument support for the change, and rationale describes the anomaly detected.

The planner's expression of the internal constraints of *changeCond* may introduce a newly created state variable, *newVar*, that impacts the existing *stateVar*, or a new function, *newFunc*, that is part of the adaptation. Both *newVar* and *newFunc* can be null, but *newVar* cannot be introduced without *newFunc*. Using *changeCond*, the AOM assigns one of three operators, *ChangeVal*, *Support*, or *Substitute*. *ChangeVal* and *Substitute* both apply only to existing state variables. *ChangeVal* is directed toward a tailored state variable found in Context nodes, while *Substitute* is directed toward state variables that are not part of tailoring but are targeted by the new functionality designated in the *ChangeSet*. *Support* works with a new state variable that is part of the new functionality introduced by *ChangeSet*. The AOM crafts the operators to adapt affected security assurance cases using the following rules.

Rule 1:
**IF** *newVar = null* & *newFunc = null*
**THEN** *ChangeVal*(*stateVar*, *newState*, *evidence*, *rationale*)

Rule 2:
**IF** *newVar* ≠ *null* & *newFunc* ≠ *null*
**THEN** *Support*(*stateVar*, *newVar*, *newFunc*, *evidence*, *rationale*)

Rule 3:
**IF** *newVar = null* & *newFunc* ≠ *null*
**THEN** *Substitute*(*stateVar*, *newState*, *newFunc*, *evidence*, *rationale*)

Given Rule 1, when both *newVar* and *newFunc* are null in *changeCond*, it indicates a direct change to a state variable. Thus, the targeted *stateVar* will be forced to change state to *newState* as part of the adaptation, yielding

*ChangeVal*(*stateVar*, *newState*, *evidence*, *rationale*)

*ChangeVal* is applied to a Context node within any security assurance case that holds the assignment of the affected state variable. The operation does not cause a change to the structure of the security assurance case as it only impacts an existing context node. The Context node change could affect the satisfaction of the goal and any security control that depends on that goal satisfaction. The adaptation A1 in Section 3.2.6.2 involves disabling confidentiality protection for data transmission on the main channel. For A1, the planner configures $ChangeSet_{A1}$ as

$ChangeSet_{A1}$ = {(*cProtect*, *false*, (*null*, *null*), *null*, "channel malfunction")}

Since *newVar = null* and *newFunc = null* within the *changeCond* of $ChangeSet_{A1}$, the AOM triggers the following *ChangeVal* operator using Rule 1

$ChangeVal_{A1}$(*cProtect*, *false*, *null*, "channel malfunction")

The operation modifies the context node in the security assurance case for SC-8 (and possibly others), because the assurance case houses the assigned value for *cProtect*. The change is shown in C1 of Figure 37.

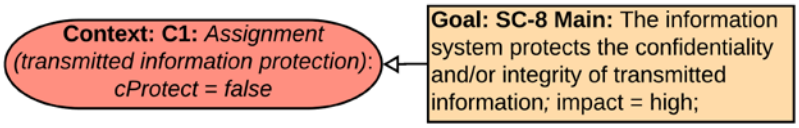



**Figure 37: Adaptation change in context node by ChangeVal Operation**

An adaptation can maintain a state variable but introduce new functionality to use it in a different way. This scenario is reflected within *changeCond* in the *ChangeSet* by introducing new variables and functions that work with an existing state variable. We restrict *changeCond* to have at most one new state variable introduced as part of the new functionality. In this adaptation scenario, the AOM activates the *Support* operator using Rule 2.

*Support*(*stateVar*, *newVar*, *newFunc*, *evidence*, *rationale*)

The *Support* operation changes the security assurance case structure by incorporating a new argument subtree containing a goal node that describes the new supporting functionality as introduced by the adaptation. The adaptation A2 in Section 3.2.6.2 introduces functionality to locally store and enable batch transmission of data, while maintaining the state of *streamInfo* to satisfy the OpGoal: G-3 subgoal of Module: M1 within the SC-8 security assurance case (see Figure 38). The planner configures $ChangeSet_{A2}$ to express the changes

$ChangeSet_{A2}$ =
{(*streamInfo*, *null*, (*localStorage*, *store*(*streamInfo*, *localStorage*)),
*storageLog*, "channel malfunction"),
(*streamInfo*, *null*, (*batchTransmission*, *enable*(*batchTransmission*)),
*checkBatchTransmission* , "channel malfunction")}

Within $ChangeSet_{A2}$ there are two tuples that express a change that needs state variable *streamInfo* without directly changing its state. Given the *changeCond* information, the AOM identifies the need for two *Support* operations that introduce two new state variables (*localStorage* and *batchTransmission*) and their new functions (*store* and *enable*). The operations are:

$Support_{A2.1}$(*streamInfo*, *localStorage*, *store*(*streamInfo*, *localStorage*),
*storageLog*, "channel malfunction")
$Support_{A2.2}$(*streamInfo*, *batchTransmission*, *enable*(*batchTransmission*),
*checkBatchTransmission*, "channel malfunction")

The change to the security assurance case for SC-8 is shown in Figure 38. OpGoal G-3 now has two new subtrees to form the new arguments representing the adaptive functionality. Goal G-3(Sub1) represents the functionality for storing the data locally ($Support_{A2.1}$) and Goal G-3(Sub2) represents the functionality for transmitting the batch data ($Support_{A2.2}$). These two subgoals are supported by Solution evidence provided by the planner (*storageLog* and *checkBatchTransmission*). Given their satisfaction, the OpGoal G-3 is still considered to perform transmission without interruption. The overall security certification would be affected if the new subgoals caused a failure in satisfying a higher-level goal.

Recall that a mechanism that is deployed to guarantee the effectiveness of a security control cannot be removed entirely without violating that guarantee. The current assumption is that the planner configures adaptations that retain these mechanisms. Thus, when functionality must be replaced, the planner can only change lower level processes to minimize security assurance case failure. For these cases, the AOM uses the *Substitute* operator as seen in Rule 3 to produce the following.

*Substitute*(*stateVar, newState, newFunc, evidence, rationale*)

The operation replaces the goal associated with *stateVar* with an alternative goal describing the new functionality. Adaptation A3 assumes that each channel has a priority field. It introduces the *priorityReplace*(*channel, newChannel*) function that embeds a priority field comparison and replaces the existing malfunctioning channel with another channel that is available in the system. The planner configures the following $ChangeSet_{A3}$:

$ChangeSet_{A3}$ = {(*channel, newChannel*, (*null*, *priorityReplace*(*channel, newChannel*)), *checkChannel*, “channel malfunction”)}

Using $ChangeSet_{A3}$ the AOM identifies the change as a *Substitute* operation because it does not introduce a new state variable but only introduces a new function to replace the current channel with a new channel at the correct priority level, which occurs only within a Goal node. The *Substitute* operation is configured as

$Substitute_{A3}$(*channel, newChannel, priorityReplace*(*channel, newChannel*), *checkChannel*, “channel malfunction”)

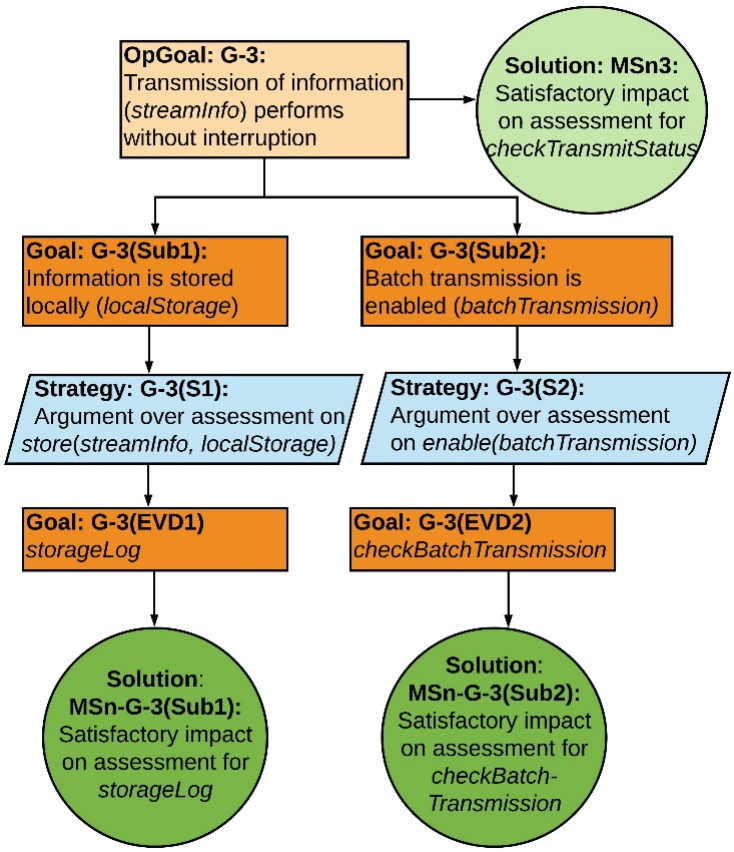



**Figure 38: Adaptation change in transmitInformation by Support Operation**

As shown in Figure 39, the adaptation updates the OpGoal: G-2 in Module: M1 of SC-8 to reflect that the current channel now relies on a priority. The Solution node remains the same.

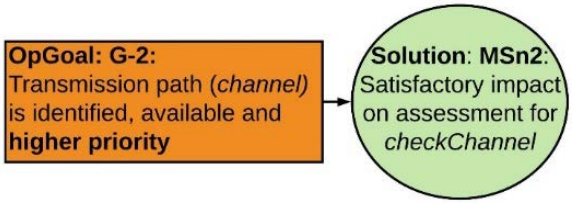



**Figure 39: Adaptation change in checkChannel by Substitute Operation**

One challenge in adapting the security assurance cases is that security controls have interdependencies, as shown in the presented template. Thus, we will be investigating how to create operators that propagate the adaptation to all dependent controls. To automate the instantiation and adaptation, we will examine how best to implement the security assurance cases, such as using XML, and implement the AOM into the MAPE-K control loop to directly adapt the assurance case as the code is adapted.

### 3.2.7 Evaluating Security Assurance Case Adaptations

#### 3.2.7.1 Returning to Security Assurance Cases

For the security assurance cases, we continue to use the NIST SP800-53 security controls. We return to the use of audit controls, AU-4, AU-5, and AU-5(1) as shown in Figure 40. A security control associates a title with each identifier. AU-4 refers to the 4th control within the Audit family of controls. The actor is either the information system or the organization. The control statement follows the actor designation. It may be a single statement, like AU-4, or separated into distinct parts, like AU-5(a) and AU-5(b). The statement can contain a mix of functional and non-functional requirements. Tailoring, a major part of security control certification, is performed when the organization instantiates what is required by the [Assignment: …] for the information system under consideration.

The related controls infer a dependency relationship among the controls. For AU-5, they are AU-4 and SI-12. There are other controls that tag AU-5 as a related control, such as AU-4, with different dependencies. The relationships may be tightly coupled, where AU-5 relies on the audit storage capacity determined in AU-4, or loosely coupled, where AU-4 provides AU-5 with a parameter it obtains from its related control AU-11. These inter-dependencies can be used to assess the impact of a self-adaptation on not just a single security control, but on the network of security controls. AU-5(1) is a control enhancement, which provides additional specification decisions and constraints. The related controls can be inherited from the main control or the enhancement can have its own related controls that are not shared with the main control. Controls are assigned to a baseline set related to the impact on the confidentiality, integrity, or availability of the system if a breach occurs. For example, AU-5 appears in the baseline set for moderate impact, while AU-5(1) appears in the baseline set for high impact systems.

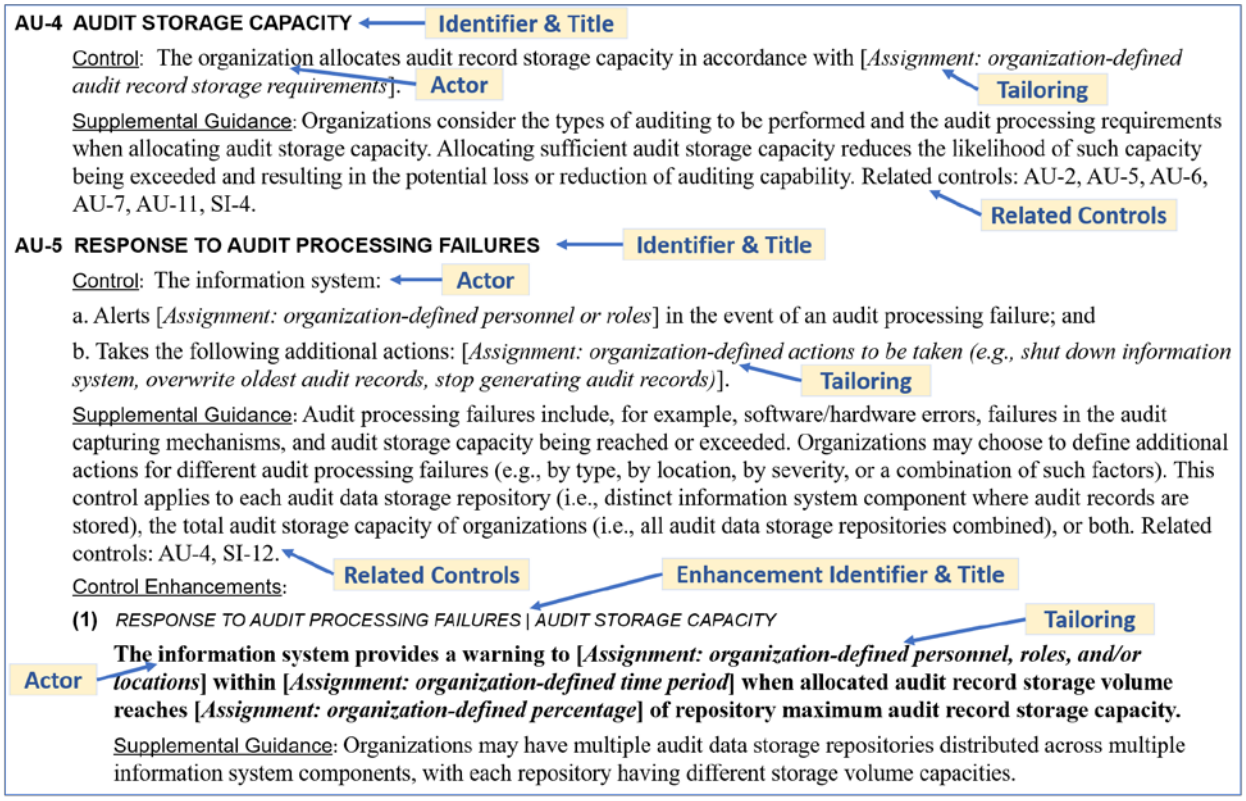



**Figure 40: Security Controls AU-4, AU-5, and AU-5(1)**

The NIST SP800-53A (NIST, 2014) companion to the 800-53 (NIST, 2013), provides assessment guidelines for each security control. Figure 41 shows the guidelines for AU-5(1). Notice that it dissects the security control statement into evaluative portions, providing distinct labels for each portion. We use both the security control and its guidelines to create and instantiate a security assurance case for a specific information system using GSN.

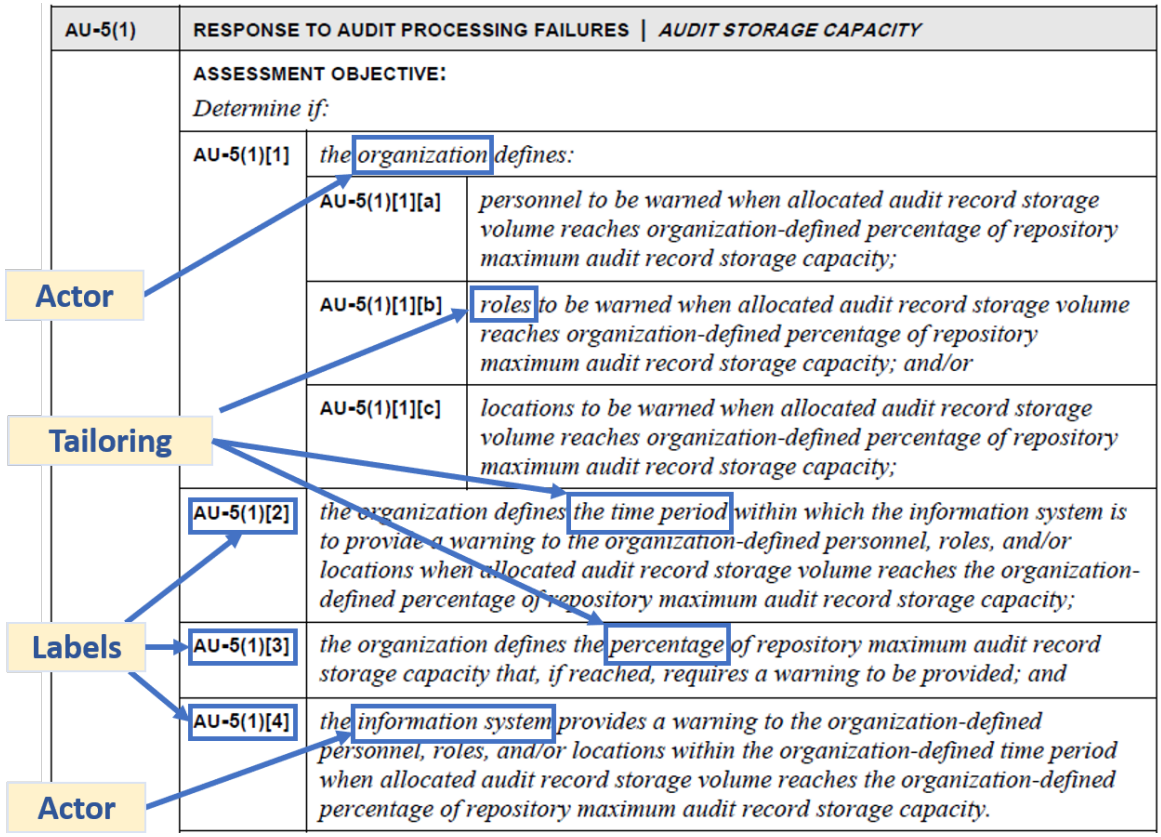



**Figure 41: AU-5(1) Assessment Guidelines**

### 3.2.7.2 Reusing the Smart Inventory Management System (SIMS) Case Study

We demonstrate security assurance case expression, evolution, and satisficing evaluation on a sample Smart Inventory Management System (SIMS) mentioned in Section 3.2.2. The process flow for SIMS appears in Figure 42. Process flow understanding is needed because it is possible to formally express the low-level functionality that is part of a security control and directly prove the implementation complies with it as we have shown in a prior report. A formal proof can be part of the argument needed within a security assurance case as described in the next section.

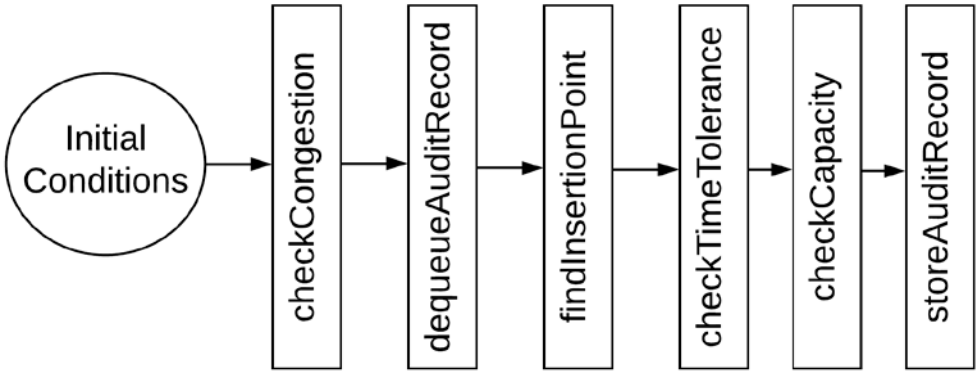



**Figure 42: SIMS Audit Component Processes**

The MAPE-K loop in each SIMS's component monitors for anomalies in the system and activates the planner to generate an adaptation. The *checkCongestion* process in Figure 42 provides the monitor with information about the input queue. Imagine that the monitor has received a certain pattern of information from *checkCongestion* that causes it to invoke the analyze phase. Here it is determined that the input queue is filling too rapidly for Audit, but that Measure and Process are not the problems. The planner configures three potential adaptations.

**A1**: Increase the capacity ratio limit, delaying the generation of an audit trail capacity alert.
**A2**: Introduce a new storage buffer and alter Audit to offload older records in its audit trail to the new buffer.
**A3**: Change Audit to overwrite old records and disable capacity alert within the same audit trail.

We return to these adaptations after introducing the assurance cases for the security controls.

#### 3.2.7.3 Creating Security Assurance Case for AU-5(1) in the New Template

Given that 800-53 security controls have a similar structure as in Figure 40, we extend a GSN template for security assurance cases to allow for dependency and achievement weight expressions as shown in Figure 34. The 800-53A directs the expansion of the assurance case into subgoals, context elements, and strategies for each control. Evidence can be formulated by multiple means, such as testing, model checking, and proof. We express the template in XML, based on CertWare (CertWare, 2007) but without the use of its display facilities to allow for more coding flexibility.

Figure 43 instantiates the security assurance case template for AU-5(1) using 800-53A labels. The subgoal Req1 is a functional requirement represented by an invariant expressed in Linear Temporal Logic, as "it is always the case that the audit trail size is less than the capacity ratio limit associated with the record storage capacity or an alert occurs." The context nodes in the instantiation have the tailoring for *capRatioLimit* and the various alert parameters segregated in Figure 41. AU-5 holds the capacity value in its provision set for AU-5(1) that it acquires from its dependency on AU-4. AU-5(1) assigns the value of *capAlert* which it provides to AU-5. The modules M1-M6 are the operational goals related to the process flow for SIMS in Figure 42.

#### 3.2.7.4 Adapting Assurance Cases

To illustrate performing and evaluating an adaptation on a security assurance case, we expand Module M5 in Figure 43 to show the argument of maintaining a satisfactory impact on the *checkCapacity* process. Figure 44 shows the expanded module for M5, which has the argument over the proof process of our system. The proof process is modeled as operational goals to maintain the invariant subgoal from Figure 44.

We assume the MAPE-K loop planner can describe the needed changes to the XML that represents the security assurance case and construct the adapted assurance cases for **A1** through **A3** as described in Section 3.2.7.2.

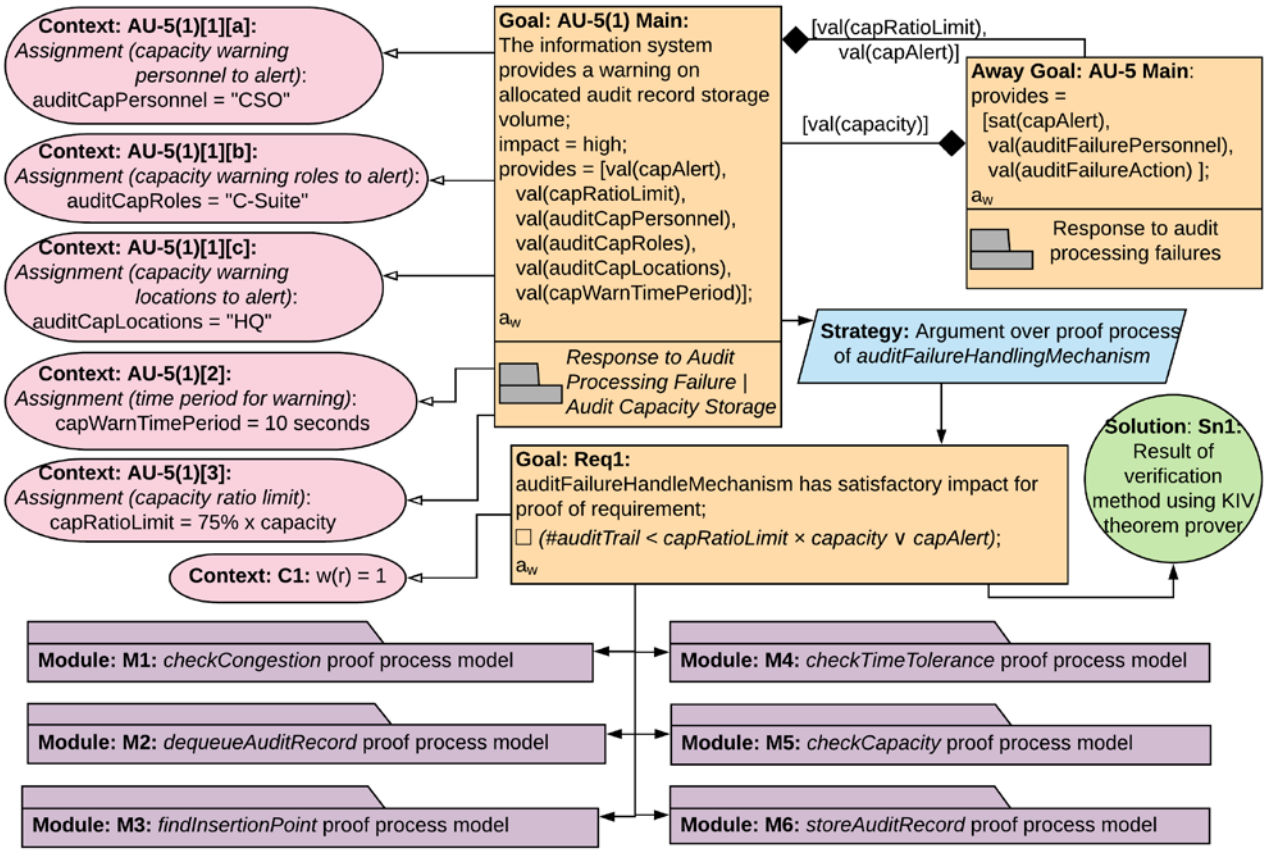



**Figure 43: Security Assurance Case for AU-5(1)**

Adaptation **A1** directly affects the assurance case for AU-5(1) by changing the *capRatioLimit* tailored value in the context node Context: AU-5(1)[3] of Figure 43. Figure 45 reflects the change to the adapted Context: AU-5(1)[3] node, where the tailored value increases from 75% to 90%. It also includes the XML for that context node where the adaptation increases *capRatioLimit* as shown on line 31. The impact to the achievement weight is shown on line 34.

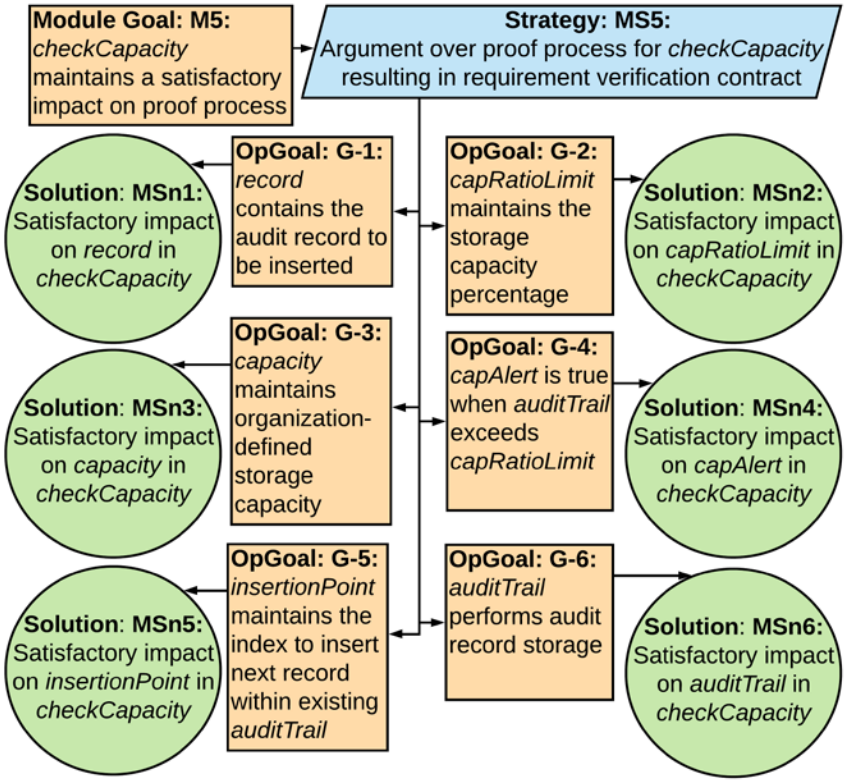
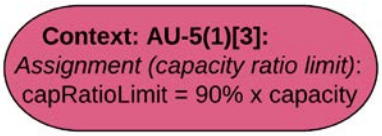



**Figure 44: Expanded checkCapacity Module**

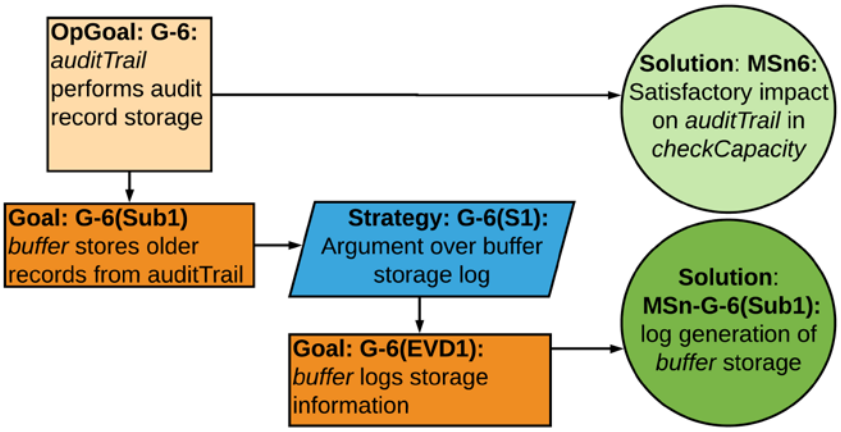



```
31 <Context assignmentTailoring="capRatioLimit = 90%"
32          identifier="AU-5(1)[3]"/>
33 <Goal identifier="OpGoal-G2"
34       achievementWeights="0.2"
35       state="capRatioLimit = 90%"
36       stateVar="capRatioLimit"/>
```

**Figure 45: AU-5(1) with Adaptation A1**

Adaptation **A2** introduces a new buffer into the Audit component, but AU-5(1)'s assurance case has no solution node to satisfy the new subgoal. Because there exist security controls that refer to offloading audit records to alternate storage, we assume the planner can reuse the evidence that such logging is sufficient to comply with operation goal G-6.

Figure 46 reflects the adapted operational goal G-6 from Figure 44 for adaptation **A2.** This adaptation introduces a new branch for G-6 to be satisfied with an argument using an external buffer to store older records in the audit trail through G-6(Sub1), G-6(S1), G-6(EVD1). The XML produced by the planner reflects the argument additions. Line 31 shows a reduced achievement weight to 0.5, reflecting the potential for a negative impact on the goal. The goal for the new supporting argument is added at line 34.



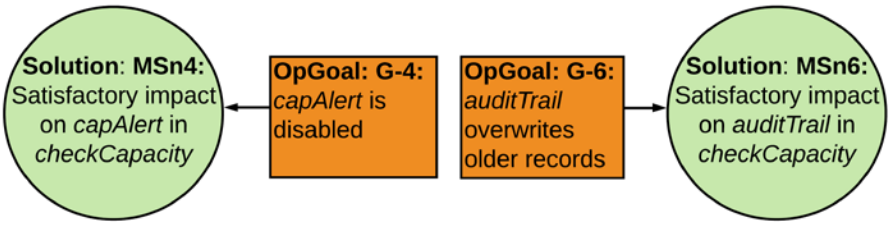

```
12 <Goal identifier="G-AU-5(1)"
13       achievementWeights="0.7777777777777778"
14       family="AU"  number="5"
15       actor="IS" baselineAllocation="high"
16       controlAction="a warning on allocated
17                      audit record storage volume"
18       provides="val(capRatioLimit);
19                 val(auditCapPersonnel);
20                 val(auditCapRoles);
21                 val(auditCapLocation);
22                 val(capWarntimePeriod)"
23       requires="val(capRatioLimit);
24                 val(capacity)"/>
25 <Goal identifier="M5"
26       achievementWeights="0.8333333333333334"
27       assessmentProcess="checkCapacity maintains
28                          satisfactory impact on
29                          proof process"/>
30 <Goal identifier="OpGoal-G6"
31       achievementWeights="0.5"
32       state="auditTrail offload older record"
33       stateVar="auditTrail"/>
34 <Goal identifier="OpGoal-G6(sub1)"
35       achievementWeights="0.5"
36       state="store older record"
37       stateVar="buffer"/>
```

**Figure 46: AU-5(1) with Adaptation A2**

Figure 47 shows the affected operational goals G-4 and G-6 from Figure 44 due to adaptation **A3**. The adaptation affects G-6 and G-4 by substituting their functions with overwriting older records and disabling the capacity alert, respectively, to satisfy module M5's goal. The XML lines 31and 35 indicate the reduced achievement weights to 0.2 that impact the goal specified in line 12.

### 3.2.7.5 Goal Satisficing Level Determination using Achievement Weights

Maintaining the security control in the self-adaptive system is a non-functional requirement. We represent each main security control as a softgoal and use the subgoals and operational goals from its security assurance case to create direct edges that form a Softgoal Interdependency Graph (SIG) (Mylopoulos, 1992). The SIG results in a tree with only AND relationships. We adapt the Soft Goal using Weight (SGW) approach (Kobayashi, 2016), to determine the satisficing level of the assurance case. A modified vulnerability metric calculation (Wei, 2018) provides the achievement weight of each softgoal. Satisficing calculations can indicate the impact of an adaptation on the security assurance case, including propagation of required state values from other security controls. The remainder of the section defines the formulas used and their adaptations. We show how the achievement weights and satisficing levels are calculated for adaptations **A1-A3** and the level of satisficing that results from each.




```
12 <Goal identifier="G-AU-5(1)"
13       achievementWeights="0.7111111111111111"
14       family="AU" number="5"
15       actor="IS" baselineAllocation="high"
16       controlAction="a warning on allocated
17                      audit record storage volume"
18       provides="val(capRatioLimit);
19                 val(auditCapPersonnel);
20                 val(auditCapRoles);
21                 val(auditCapLocation);
22                 val(capWarntimePeriod)"
23       requires="val(capRatioLimit);
24                 val(capacity)"/>
25 <Goal identifier="M5"
26       achievementWeights="0.7333333333333334"
27       assessmentProcess="checkCapacity maintains
28                          a satisfactory impact on
29                          proof process"/>
30 <Goal identifier="OpGoal-G4"
31       achievementWeights="0.2"
32       state="capAlert = disable"
33       stateVar="capAlert"/>
34 <Goal identifier="OpGoal-G6"
35       achievementWeights="0.2"
36       state="auditTrail overwrite older record"
37       stateVar="auditTrail"/>
```

**Figure 47: AU-5(1) with Adaptation A3**

Using the SGW approach, we define a softgoal interdependency graph, $SIG_A$, for the security assurance case, A, as a tree of goals with the main security goal, $m_A$, as the root. $SIG_A = (G_A, D_A)$ where

- $G_A = \{m_A\} \cup O_A$
- $O_A$ = set of subgoals and operational goals for A that support the main security control, $m_A$ (root)
- For all goals $g \in G_A$, $a_w(g)$ is the achievement weight calculated for that goal.
- $D_A$ = the set of edges (p, c), representing dependencies among the parent (p) and child (c) goals in $SIG_A$.

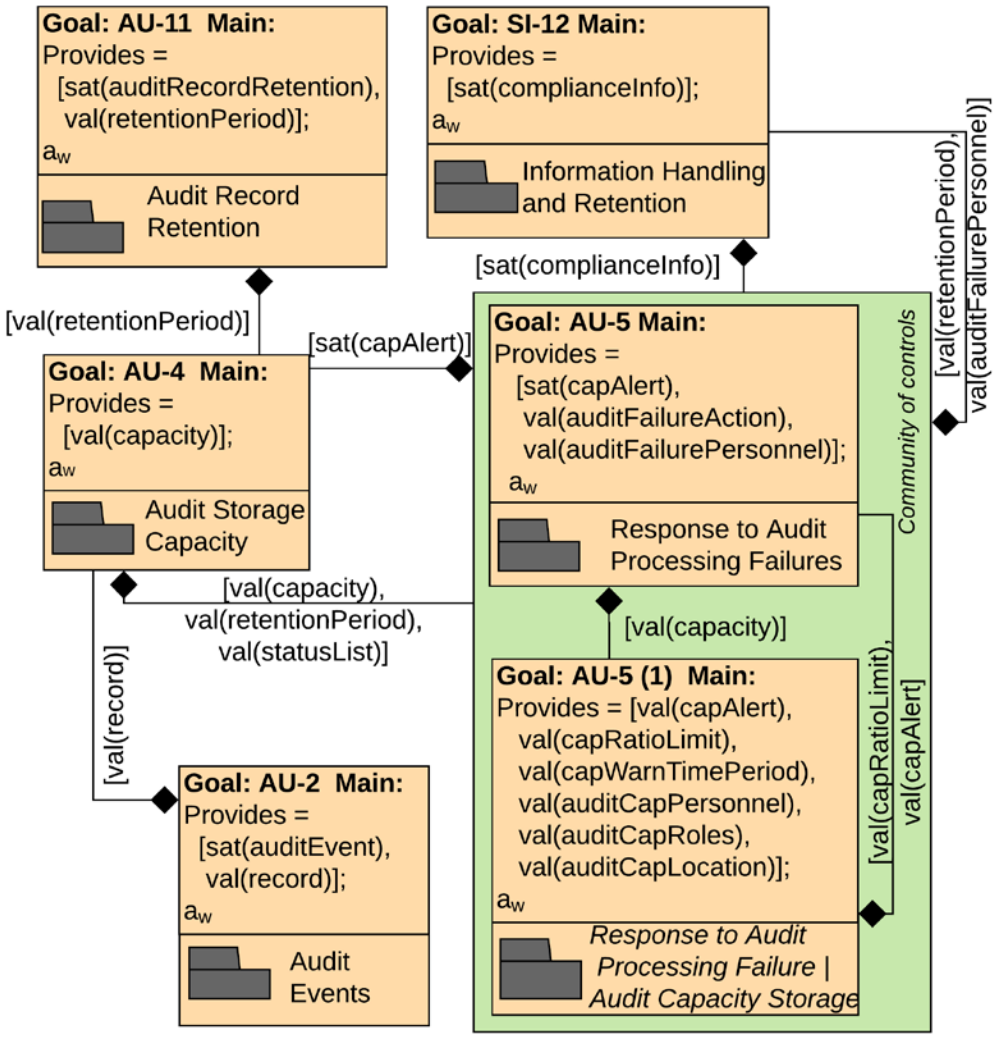



**Figure 48: Sample Security Control Network**

The related security controls introduce inter-dependencies that form a network of security controls. As assurance cases, they only have knowledge of the controls on which they depend. However, from the MAPE-K loop perspective, the inter-dependencies can be traversed as an adaptation is evaluated. A partial dependency graph appears in Figure 48. The links specify the provision sets passed from source (diamond) to target control. This expression facilitates the propagation impact evaluation of an adaptation.

The security control network (SCN) = (M, $D_M$), where

- M = $\{\bigcup_{SIG} m\}$, the set of all SIG root goals
- $D_M$ = set of weighted, directed edges with provision sets representing dependencies among security controls (Figure 48).

The community structure advocated by vulnerability metric calculation provides for a higher degree of influence across the edges. In our representation, a security control and its enhancements form a natural community, as represented by the green box surrounding AU-5 and AU-5(1) in Figure 48. To calculate $a_w(g)$ for $g \in G_A$, we measure the vulnerabilities of the community structure in the SCN. The achievement weight is inversely related to a community's vulnerability.

Achievement weight is then defined for a $SIG_A$ as

$$a_w(g) = I(g), \text{ for leaf nodes, } g \in O_A$$
$$= average\big(a_w(c)\big), \text{ for all } c \in children(g) \text{ for non-leaf nodes, } g \in G_A$$

where $I(g)$ is the impact factor defined on the state variables supporting the operational goals at the SIG leaves. Currently, $I(g)$ must be determined by the certifiers prior to deployment given potential changes to state variables and the organization's risk policy.

Table 24 provides sample values for $I(g)$ related to the state variables affects by adaptations **A1-A3**. A lower value has more negative impact on achievement weights. In a community, the control enhancements (e.g. AU-5(1)) propagate their achievement weights to their community parent (e.g. AU-5) as one of its edges.

**Table 24: Sample Impact Table**

| $I(g)$ | *capRatioLimit* | *capacity* | *auditTrail* | *insertionPoint* |
|---|---|---|---|---|
| *1* | = 75 % | = 100 | Store record | ≤ *#records* |
| *0.9* | | > 100 | | |
| 0.5 | < 75% | < 100 | Offload older record | > *#records* |
| 0.2 | > 75% | | Overwrite older record | |
| 0 | ≤ 0% or ≥ 100% | ≤ 0 | Drop record | < 0 |

Determining the satisficing level of a main control softgoal, such as AU-5, relies on the SCN. A partial SCN is shown in Figure 48. The satisficing level, $SL(m)$, of main goal $m$ is the average of achievement weights that include $a_w(m)$ and the $neighbors(m)$ as defined by the direction

that the provision sets are passed. For example, $neighbors$(AU-4) = {AU-2, AU-5, AU-6, AU-7, AU-11, SI-4} from Figure 40, with a subset shown in Figure 48. Thus,

$$SL(m) = average(a_w(m) + \sum_{g \in neighbors(m)} a_w(g))$$

A control enhancement, $e$, that has a neighbor outside of its community can potentially have $SL(e) \neq a_w(e)$. In this case, $SL(e)$ has priority. When security controls are mutually related with the same provision, the algorithm cannot double count the impact. To resolve this issue, our satisficing algorithm preserves the last calculated achievement weight, $a_{prev}(g)$, and uses that achievement weight as the neighbor's achievement weight to stabilize the network-based calculation. We assume that when deployed, the SIMS security controls have an achievement weight of 1. We show how adaptations **A1-A3** directly lower certain achievement weights and propagate the impact through the SCN.

### 3.2.7.6 Adaptation Results

Table 25 shows the achievement weight changes for AU-5's partial community after applying adaptations **A1-A3** to the security assurance case for AU-5(1). Though we focused on Module M5, other modules are also affected by the adaptations and are reflected in Table 25.

**Table 25: $a_w(g)$ in AU-5 Community**

| Goal | Base | A1 | A2 | A3 |
|---|---|---|---|---|
| Opp-G1 | 1 | 0.2 | 1 | 1 |
| Opp-G2 | 1 | 1 | 1 | 1 |
| Opp-G3 | 1 | 1 | 1 | 1 |
| Opp-G4 | 1 | 1 | 1 | 0.2 |
| Opp-G5 | 1 | 1 | 0.5 | 1 |
| Opp-G6 | 1 | 1 | 0.5 | 0.2 |
| M1 | 1 | 1 | 1 | 1 |
| M2 | 1 | 1 | 0.5 | 0.2 |
| M3 | 1 | 1 | 0.5 | 0.6 |
| M4 | 1 | 1 | 1 | 1 |
| M5 | 1 | 0.867 | 0.833 | 0.733 |
| M6 | 1 | 0.867 | 0.833 | 0.733 |
| G1 | 1 | 0.956 | 0.778 | 0.711 |
| AU-5(1) | 1 | 0.956 | 0.778 | 0.711 |
| AU-5 | 1 | 0.956 | 0.778 | 0.711 |

Table 26 shows the satisficing level computed for each main security control at the base (deployed) level and after applying adaptations **A1-A3**. Note that $SL$(AU-5(1)) = $a_w$(AU-5(1)) because the **A1-A3 a**re internal to that security control. AU-5 is affected by **A1-A3** because of its relationship with AU-5(1). The effects of **A1** and **A2** only propagate to AU-5 since the adapted provisions remain in the community. Adaptation **A3** impacts AU-5 and AU-4 because *capAlert* is in the propagated provision set (Figure 48).

**Table 26: Satisficing Levels**

| | Base | A1 | A2 | A3 |
|---|---|---|---|---|
| AU-2 | 1 | 1 | 1 | 1 |
| AU-4 | 1 | 1 | 1 | 0.928 |
| AU-5 | 1 | 0.985 | 0.926 | 0.903 |
| AU-5(1) | 1 | 0.956 | 0.778 | 0.711 |
| AU-11 | 1 | 1 | 1 | 1 |
| SI-12 | 1 | 1 | 1 | 1 |

### 3.2.7.7 Adaptation Evaluation

To evaluate the alignment of the adaptive system behavior with the satisficing level determination in Section 3.2.7.6, we deploy **A1-A3** in the SIMS application. We embed checkpoints as probes in the *checkCapacity* module (M5 in Figure 43 and Figure 44) and log the effects on the audit trail. Figure 49 shows how the checkpoints are placed to determine if (*i*) a *record* is generated (CK1), (*ii*) the *capRatioLimit* is maintained (CK2), (*iii*) the audit trail *capacity* is maintained with capability to store a record within the *auditTrail* (CK3), (*iv*) the alert is properly performed by *capAlert* (CK4), (*v*) the proper *insertionPoint* can be found to store the next record while maintaining the existing *auditTrail* contents (CK5), and (*vi*) the record is stored in *auditTrail* (CK6).

We ran tests with sufficient audit trail capacity and insufficient audit trail capacity. With sufficient capacity, adaptation **A1** performs better than **A2** and **A3**. Allowing more records to flow into the audit trail is a local change that impacts only a single state variable and does not propagate outside the community. Thus, **A1** is not heavily relied on by the assurance case argument or proof for all audit functionality. **A2** and **A3** impact several operational goals that are needed for the overall argument or proof. Table 27 shows the results with insufficient capacity in which the audit trail can hold only 50 records. Column 1 represents the base deployment (B), followed by the adaptations when the number of records needed is 75 and 100. **A1** does poorly with insufficient records. **A2** performs the best but requires addition buffer storage. **A3** performs worse than **A1** overall. **A3** fails at CK4 by disabling *capAlert* and fails at CK5 when overwrite functionality violates the requirement that the insertion point maintains the records in the audit trail.

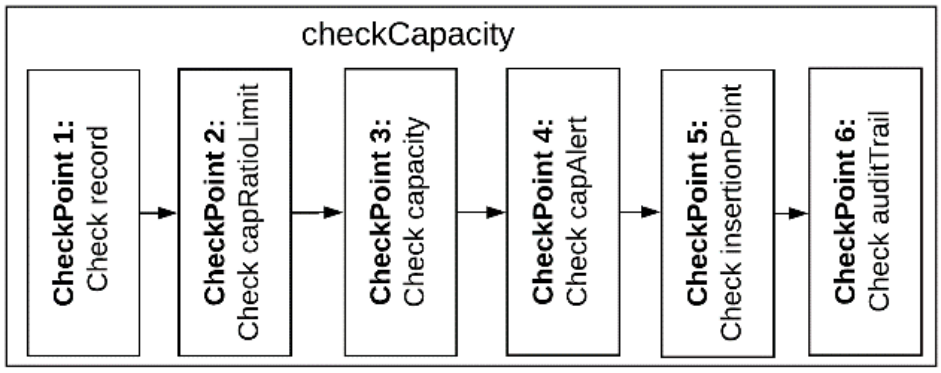



**Figure 49: checkCapacity Checkpoints**

**Table 27: Performance Evaluation Results**

| | #Rec | CK1 | CK2 | CK3 | CK4 | CK5 | CK6 |
|---|---|---|---|---|---|---|---|
| **B** | 75 | 75 | 37 | 50 | 75 | 50 | 50 |
| **B** | 100 | 100 | 37 | 50 | 100 | 50 | 50 |
| **A1** | 75 | 75 | 43 | 50 | 75 | 50 | 50 |
| **A1** | 100 | 100 | 43 | 50 | 100 | 50 | 50 |
| **A2** | 75 | 75 | 74 | 50 | 75 | 75 | 75 |
| **A2** | 100 | 100 | 99 | 50 | 100 | 100 | 100 |
| **A3** | 75 | 75 | 74 | 50 | 37 | 37 | 75 |
| **A3** | 100 | 100 | 99 | 50 | 37 | 37 | 100 |

#### 3.2.7.8 Discussion

We focused on the AU-5 control, which is the same control that we formally specified and proved correct using the KIV theorem prover in Section 3.2.4. We also represented this control within our verification process model and provided a risk assessment calculation for comparative adaptive plans. The use of security assurance cases for the same control provides a complementary specification that introduces the calculation of a satisficing level of a security control for a potential self-adaptation based on its internal changes and from propagated satisficing levels in the network. In addition, we implement the security assurance cases using XML to perform the adaptations and measurements at runtime, as demonstrated using a sample application with three adaptations and embedded checkpoints. We show the alignment of the adaptation failure rates with the calculated satisficing levels. Using system domain knowledge, experts can introduce satisficing level thresholds to identify acceptable adaptations.

Scalability is a potential limitation of both the formal and the assurance case approaches given the size of the security control network of related controls for a large-scale system. The embedded verification process model along with the XML representation can streamline the automated assessment process when an adaptation is considered. Though codifying the security assurance cases in XML is potentially burdensome during design, once codified, achievement weight and satisficing level determination could be optimized. Formal verification offline and during design time can provide the evidence needed for the adaptive security assurance case.

### 3.2.8 Examining the Framework in an Alternate Testbed with Different Formalisms

#### 3.2.8.1 Adaptive Coordination to Complete Mission Goals

Coordinating distributed systems, such as autonomous systems, is a complex problem in which each system develops an individual, local plan that is refined while synchronizing with other systems to make use cooperative opportunities for improved completion of mission objectives and avoid potential conflicts. The coordination objective is to achieve the global mission goal while using resources effectively even as the environment changes. To model the specifications of the

local and global missions, we use the Partial-Order, Causal-Link (POCL) plans for multi-agent systems (Cox, 2005). POCL plans describe each individual agent's plan to reach its local goal. Moreover, the union of local POCL plans represents the overall multi-agent system, or unit, plan. Any detected violations of the multi-agent POCL plan would be used as adaptation triggers such that agents can self-integrate into each other's local goals to accomplish a global mission.

In order to ensure adaptations to the local goal do not conflict with the global goal, there needs to be a method in place to both detect when a change is required and validate that mission constraints can be maintained by any potential changes. One way to do this is with assurance cases (Rushby, 2015). Assurance cases organize the necessary evidence and arguments to show that a system complies with a critical requirement. Assurance cases provide the descriptive medium that can regulate adaptive changes to requirements for an agent to maintain resilience in achieving its goals (Jahan, 2018) will discuss in Section 3.2.8.3.

#### 3.2.8.2 Case Study using Cozmo testbed

To examine self-adaptation for local integration that ensures global goal completion, we create a platform for multiple physical agents to coordinate to complete a global mission with minimal intercommunication. Our platform uses the Anki Cozmo robot as an agent (Cozmo, 2018). Access to Cozmo's SDK allows for custom programs to be created. The architecture of our testbed is depicted in Figure 50. We use Raspberry Pis as base stations running our code, using Bluetooth to communicate with each other. Bluetooth has good throughput to communicate current mission goals, such as which cubes have been collected, consistently transmits messages for devices that are near each other, and does not require internet connectivity for testing. Each Raspberry Pi is tethered to an Android device running the Cozmo app. This tethering is the only way to access the Cozmo SDK. The Android device connects to Cozmo through wi-fi and performs the required image processing for Cozmo to recognize objects and to know its world location. Each Cozmo hosts its own wi-fi server, allowing only a single connection from a device running the Cozmo app. Each Raspberry Pi contains the mission for its connected Cozmo to complete. For our case study, we use the existing Cozmo cubes as objects to collect as the unit's mission to complete.

Each Cozmo contains a camera, IR sensor, and knowledge of its location and distances to objects it can recognize. It can recognize faces, its charging station, and the cubes that come with the Cozmo. We can create simple local tasks for it to complete, including specifying the distinct cubes it is assigned to collect. Cozmo first searches for its assigned cubes to collect. After finding an assigned cube, it will attempt to collect it, followed by all the cubes it can as part of the unit's global goals.

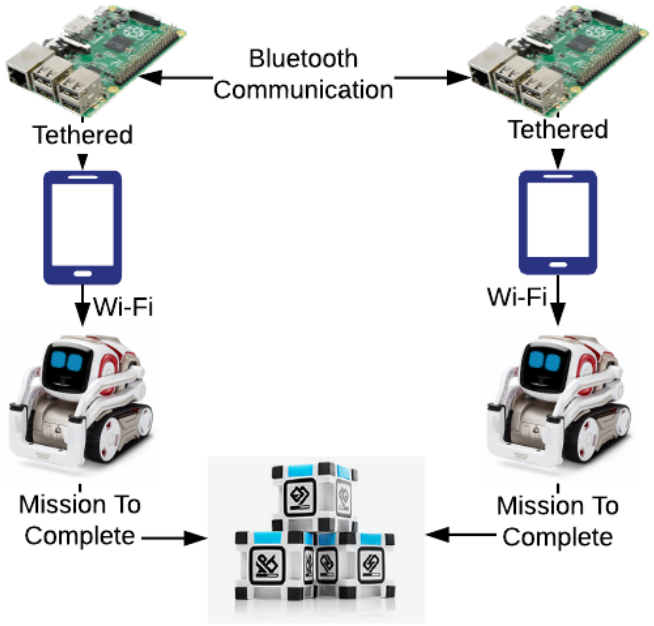



**Figure 50: Cozmo testbed architecture**

For the case study, we have two Cozmos assigned to collect all cubes as quickly as possible. The cube assignment could be segregated across the two Cozmos or overlap. Each Cozmo is told to complete the task independently in as little time as possible. They are in contact with each other Cozmo to ensure all cubes are collected. In the naive approach, each Cozmo will search for and collect the cubes they are assigned. However, to improve this method, each Cozmo will be capable of adapting, through their connected Raspberry Pis, their own cube-collection order and the cubes they collect based on their position. Each Cozmo first examines its surroundings to detect cubes in its vicinity and reports to the other Cozmo the cubes it can see. If it is able to see the cubes it is assigned to collect, and these cubes are close to it, it will adjust the order of cubes it can collect by collecting the closest cube, returning it to its "home base," and then collecting the next cube in the order. Cozmo can reorder its collection method based on distance.

However, if Cozmo can only see cubes it is not assigned to collect, it can use the knowledge it obtained and stored through communicating with the other Cozmo to update its list of cubes to including newly assigned ones. Thus, a Cozmo can self-integrate into the second Cozmo's mission to ensure a rapid completion with minimal resources. For our case study, we will embed the code for each Cozmo to examine, assess, and perform one of two adaptations to its policy:

**A1**. Broadcast current task allocation to other Cozmos before deciding to pick up cubes

**A2**. Broadcast information about completed and current goals after detecting or picking up a cube

### 3.2.8.3 Multi-agent Coordination using Self-Integration

The multi-agent system we target enables multiple agents to work together to achieve a global mission goal by accomplishing their individual goals independently. However, these agents must coordinate their local goals to avoid potential conflicts with another agent's local goals and take advantage of cooperative opportunities to complete the global goal. Each agent has its own plan to accomplish its local goal and the union of agents' plans represents the multi-agent (global) plan. By combining agents' plans in one global plan, the agents can detect any conflict between their plans and coordinate to avoid such conflicts. Conflicts may include redundant goals or missing requirements for specific goals. In order to detect any violation or flaw of the plan specifications, agents are supposed to exchange the important information about their current and intended goals. By detecting flaws, they can locally decide to adapt by self-integrating into each other's local plan (i.e. assign themselves to another agent's local goal or to a step of its plan) to achieve the overall mission goal. They must still satisfy their local goal, but it may be altered to allow for resiliency.

We model each agent's plan to accomplish its goal using POCL. As stated in (Cox, 2005), a multi-agent POCL plan is a tuple $P = (A, O, S, \prec_T, \prec_C, \#, =, X)$ where:

- A is the set of agents
- O is a set of plan operators (such as *explore*, *detect*, *pick-up* etc.)
- S is a set of plan steps (instantiated operator enhanced with temporal and spatial details).
- $\prec_T$ is the temporal partial order on S, where e $\in \prec_T$ is a tuple $\langle s_i, s_j \rangle$ with $s_i, s_j \in$ S (ordering constraints).
- $\prec_C$ is the causal partial order where e $\in \prec_C$ is a tuple $\langle s_i, s_j, c \rangle$ with $s_i, s_j \in$ S and $c \in \sum$ where $\sum$is a predefined set of temporal and spatial conditions.
- (O,S,$\prec_T$,$\prec_C$) represents a POCL plan for an individual agent.
- X is a set of tuples of form $\langle s, a \rangle$, representing that the agent a∈A is assigned to executing step s.
- = is the symmetric concurrency relation over the steps in S.
- # is a symmetric non-concurrency relation (ordering) over the steps in S.

The relation $\langle s_i, s_j \rangle \in \#$ can be defined using the ordering constraints: $\langle s_i, s_j \rangle \in \prec_T$ *or* $\langle s_j, s_i \rangle \in \prec_T$ that indicates that $s_i$ and $s_j$ should be done in a specific temporal order. The relation $\langle s_i, s_j \rangle \in =$ means that $s_i$ and $s_j$ are required to be executed at the same time. For example, if two Cozmos are required to carry the same cube together then they both need to be timely synchronized in their cube picking steps.

Given the POCL definition of a multi-agent plan, we specify the temporal and causal partial order flaws of the agent plan and use them as adaptation triggers. For the agent's local plan, a plan flaw is either a *causal link threat flaw* or an o*pen condition flaw* as defined in (Cox, 2005).

A **causal-link threat flaw** in a single agent POCL plan exists when there is some step $s_k$ and some causal link e $\in \prec_C$ of the form

$$\langle s_i, s_j, c \rangle, s.t. not(c) \in post(s_k), \langle s_k, s_i \rangle \notin \prec_T \ and \ \langle s_j, s_k \rangle \notin \prec_T.$$

**Example 1**: Using the Cozmo Testbed Architecture, assume Cozmo A is the agent that has the following causal link in its plan:

$\langle s_i, s_j, c\rangle$ = ⟨findCube(type1), pickup(type1), cube1_at_p⟩

However, Cozmo A needs to announce to all other Cozmos that it has detected the cube before deciding to pick it up. This requirement would be specified as follows:

$\langle s_i, s_k, c\rangle$ = ⟨findCube(type1), communicate(Cozmo), cube1_at_p⟩

$\langle s_k, s_j, c\rangle$ = ⟨communicate(Cozmo), accomplishTask(type1),
sufficient_knowledge⟩

Here, Cozmo A has to communicate with other Cozmos to accumulate sufficient knowledge before making the decision either to pick the cube or ignore it (represented by accomplishTask). Assume another agent Cozmo B (hereafter we use Cozmo instead of agent) announced that it is closer to position *p* than Cozmo A and it is going to pick up cube 1. The current local goal for Cozmo A conflicts with Cozmo B. This conflict would trigger adaptation opportunity for Cozmo A to reassign itself to another goal by finding another cube.

An **open precondition flaw** exists when there is some step $s_j$ with precondition *c* but there is no causal link $\langle s_i, s_j, c\rangle \in \prec_C$.

**Example 2**: Building on the previous example, assume Cozmo A receives a message from Cozmo B with the location of cube 2 in position *q*. It then reassigns itself to pick up this cube via step $s_j$ which represents "pick up cube from position *q*". However, the causal link constraint for picking up the cube is (moveTocube, pickup, distance < collecting_range). To achieve the goal of picking up cube2, according to the given specification, Cozmo A needs to introduce a new step $s_i$ to move itself close enough to the cube to grab it. This process is a self-integration of Cozmo A to accomplish the global goal of collecting as many cubes as possible by assigning itself to a new local goal.

$\langle s_i, s_j, c\rangle$ = ⟨moveTocube (type2), pickup(type2), distance (A,q) < collecting_range ⟩

***A parallel step threat flaw*** exists in a multi-agent plan when there are steps belonging to different agents $s_j$ and $s_i$ where post($s_i$) is inconsistent with post($s_j$), $\langle s_i, s_j\rangle \notin \prec_T$, $\langle s_j, s_i\rangle \notin \prec_T$, and $\langle s_i, s_j\rangle \notin \#$.

**Example 3**: This flaw occurs when two Cozmos see the same cube and go to pick it up. Here, the postcondition of both Cozmos would be "holding the cube", which is not physically possible for two Cozmos to do simultaneously. The adaptation here would be to change the goal for one of the Cozmos according to certain rules. The rules would be used to set priorities for Cozmos, including distance to the cube, battery life, and the ability to pick up other types of cubes. For instance, one Cozmo might be only allowed to pick up one cube, the detected one, but the other Cozmo is more flexible and can collect all cubes.

### 3.2.9 Assurance Case for Control System

The multi-agent plan coordination problem can benefit from self-integration that coordinates the goals of the local control system with those of the global control system increasing the confidence level of mission success. We use GSN to define assurance cases that specify a global mission goal for the group of agents and a local mission goal for each agent. The assurance case argument is used to validate the temporal and spatial constraints of the mission, which could be violated at any point of time due to the adaptation of the agent. We apply resilience-based operations to the assurance case to allow the integration of the local goals of the agents to make the global goal consistent. In this way, both the temporal and spatial constraints of the mission can be continued to be validated. In this section, we illustrate the use of assurance cases to provide confidence that the local control system complies with its mission requirements.

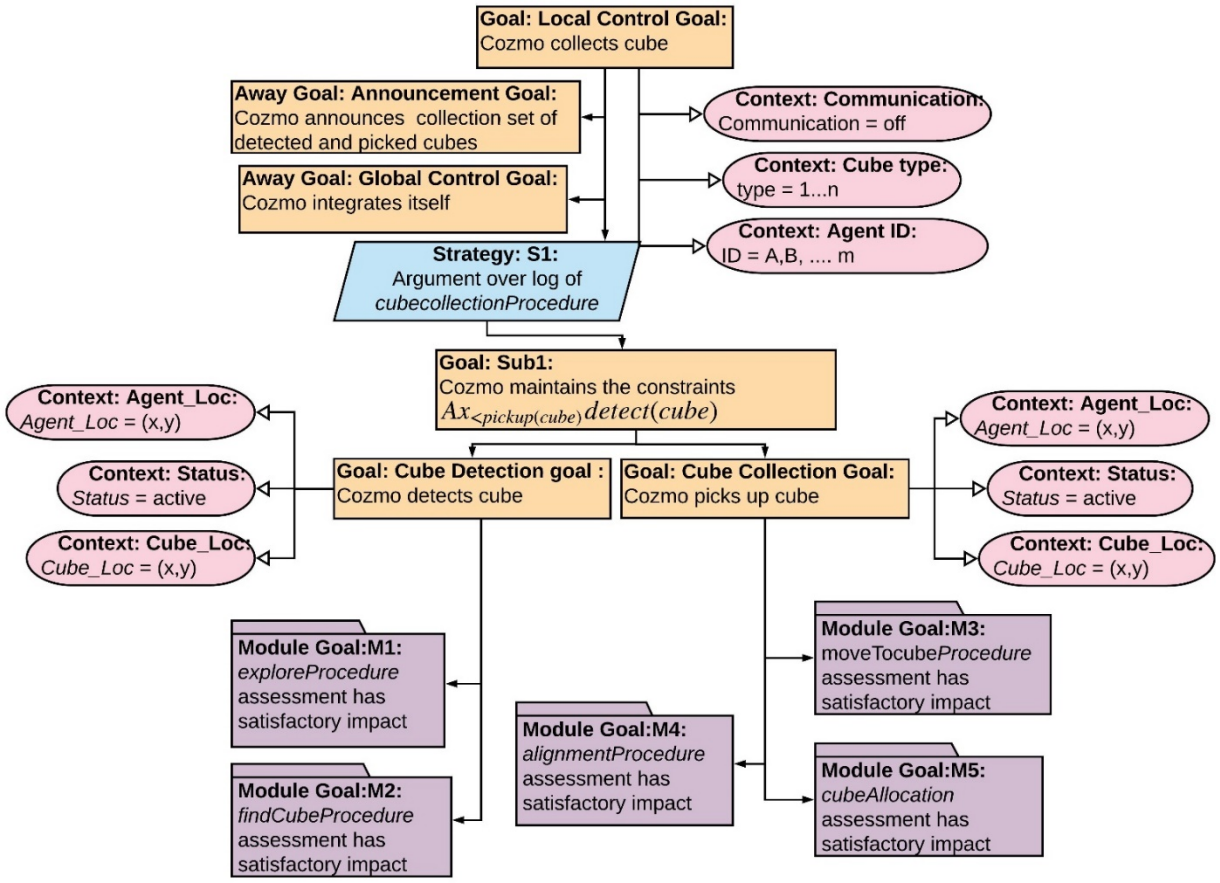



**Figure 51: Instance of a local mission assurance case**

We configure an instance of a local mission assurance case for the individual Cozmo agents using GSN notation as shown in Figure 51. The main goal is collecting cubes, which requires the satisfaction of two external goals (called “away goals” in GSN): Announcement and Global Control. The Announcement goal dictates which set of cubes are assigned for detection and collection. The Global Control goal provides the rationale regarding the local control system’s action and choice of cube.

For assurance cases, the argument for a goal involves a strategy that includes sub-goals and solutions for collecting evidence to satisfy the argument. The argument of a local goal uses the

POCL plan for that local mission. Because the POCL plan must maintain temporal and causal partial order constraints to avoid the conflicts for global mission, we reflect that as a subgoal in the assurance case in Figure 51. We have identified that the base approach of performing the local mission is to detect a cube and pick that cube. So, the causal link for base plan of local mission is

$\langle s_i, s_j, c \rangle$ = ⟨findCube(type1), pickup (type1), cube1_at_p⟩.

We use Fuzzy Branching Temporal Logic (FBTL) to express the causal link constraints to facilitate uncertainty handling and improve the tolerance level of the system while removing inconsistencies among the plans for the global mission. We identify the flexibility point for the constraints and represent them using FBTL semantics:

$Ax_{<pickup(cube)}$ detect(cube)

The context nodes connected through the GSN InContextOf link (hollow arrow in Figure 51) with the goal node holds the assigned context value for the goal. In Figure 51, the context node attached to the main goal assigns the Cozmo (agent) a location and status and cube location information for which the POCL plan will execute. The POCL plan involves an operation to accomplish the goal and argumentation over how the satisfaction of that operation is represented through the modules.

### 3.2.9.1 Adapting the Assurance Case

When an adaptation occurs and changes the system functionality, we need to incorporate the changed functionality into the assurance case to maintain consistency with the expectation that the constraints are satisfied. This incorporation demands relaxing the original ordering constraint (Ax) based on domain knowledge, which improves the tolerance level of POCL flaws.

In Section 3.2.6.3, we discuss three adaptation operators for assurance cases: (i) ChangeVal that changing the value in a Context node, (ii) Support that adds new functionality for existing state variables, causing new arguments to be added to existing subgoals, and (iii) Substitute that alters existing functionality and can introduce new state variables.

To support dynamically relax some constraints of the agent for coordinating with other agents, we introduce a *RelaxConstraints* operator by involving the RELAX process from (Whittle, 2010). This operator affects the constraint maintenance goals in the assurance case in Figure 51. During the RELAX process, the operator monitors the POCL plan and environment to define the relationship between the environment attributes and the plan's constraints. To complement the new *RelaxConstraints* operators and avoid the inconsistency of the causal order constraint in the POCL plan, we introduce an *Augmentation* operator to adjust the system's functionality by adding intermediate steps and/or local state variables to accomplish them, while retaining the critical functionality from a global perspective. In the remainder of the section, we discuss Adaptations A1 and A2 with respect to the evolution needed by the assurance case using the *RelaxConstraints* and *Augmentation* operators.

If Adaptation A1 occurs, it activates communication among the Cosmos about current task allocation, which nullifies the cube detection process. This changes the status of the Cozmo agent to *inactive* for the detection step, as reflected in context node of cube detection goal shown in Figure 52. The POCL flaws in Example 1 in Section 3.2.8.3, trigger this adaptation. The causal

link constraints for the POCL plan needs to relax the base causal link ordering. By monitoring the environment and POCL plan, RelaxConstraints determines the flexible point based on domain knowledge to relax the constraint that states "Cosmo shall communicate before proceeding to either detect or pick up for as many cubes as possible."

$$\mathrm{Ax}_{<(\mathrm{AGF}(\Delta(\mathrm{pickedUpCubes})\in S))}\ \mathrm{communicate}(\mathrm{Cozmo})$$

The *"as many cubes as possible"* clause is introduced for flexibility while handling the uncertainty and the number of cubes picked up, which is expressed as *"Δ(pickedUpCubes)*" to define the relaxed, uncertainty factors as an element of fuzzy set *S*. The *AGF* quantifier expresses a temporal "eventually true" expression with the uncertainty factor. Since the rationale for applying A1 is to improve resource utilization, a new justification node is connected to the altered goal. This adaptation includes new steps in causal links as shown in Example 1. The dependency on new steps introduces a new subgoal "Communication Activation goal″ that is placed within the assurance case through the use of the Augmentation operator (Figure 52).

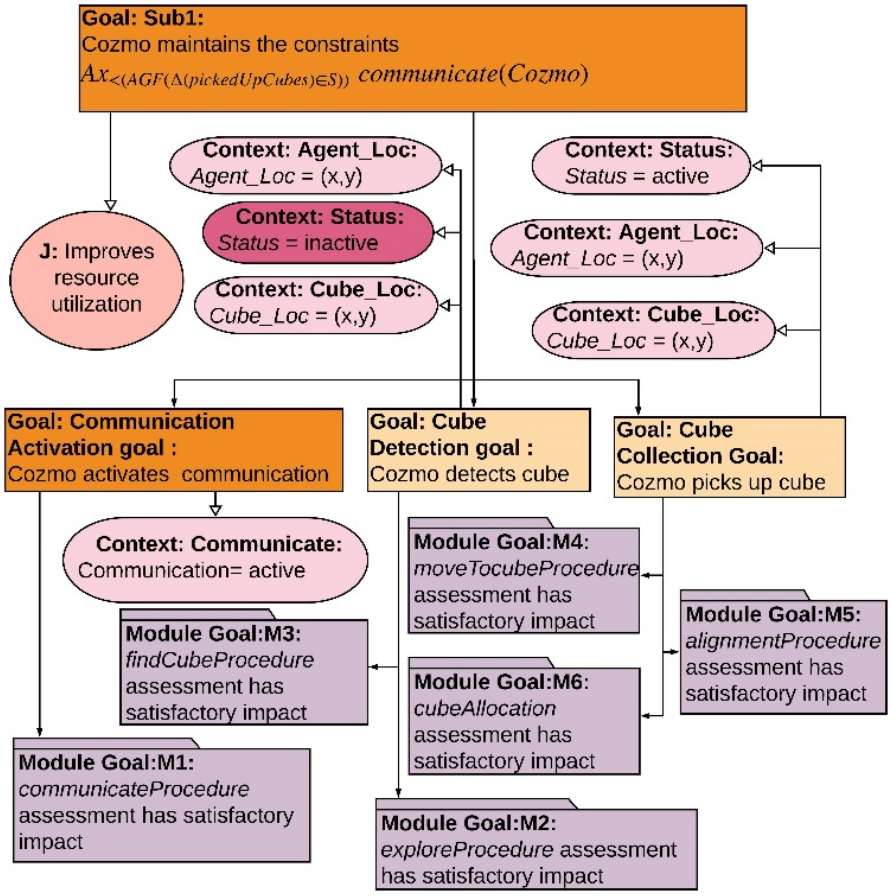



**Figure 52: Applying adaptation A1 to the assurance case**

Adaptation A2 also involves new intermediate broadcasting steps for causal link order in the POCL plan. The POCL plan flaws in Examples 2 and 3 in Section 3.2.8.3 trigger this adaptation. The adaptation introduces a new operational goal for both detecting and picking up the cube. The new goal reflects the additional functionality needed for Cozmo to broadcast its accomplishment of detecting or picking up the cube as shown in Figure 53. From this broadcasting, other agents

can self-integrate to aid in the task completion. The rationale behind this adaptation is same as adaptation A1 that is, improving resource utilization. The *RelaxConstraints* operator is activated because the POCL plan needs to relax the base causal link ordering by including intermediate steps.

$$AX_{>(AGF(\Delta(detectedCubes)\in S)\ \vee\ AGF(\Delta(pickedUpCubes)\in S))}\ broadcast(taskAccomplishment)$$

The relaxed constraints state that "Cozmo shall broadcast its accomplishment after detecting or picking up for as many cubes as possible." The adaptation demands the use of the *Augmentation* operator for detection and pick up functionality by including a broadcast function.

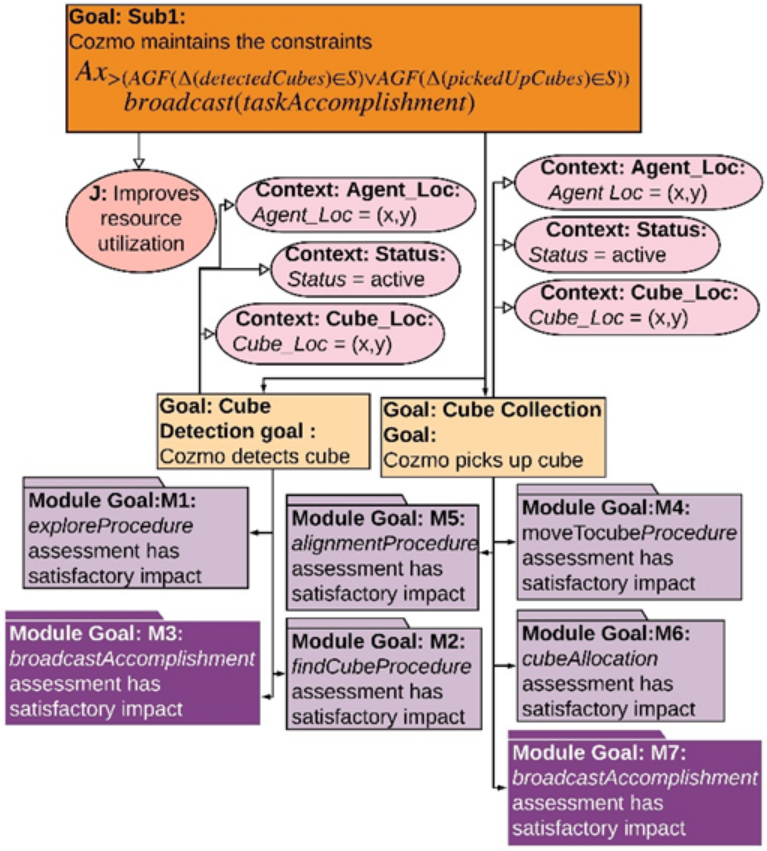



**Figure 53: Adaptation A2 applied to the original assurance case**

## 3.2.9.2 Evaluation

We evaluate our method first by looking at how long it takes for Cozmo to collect all three cubes by itself, which yields a baseline for the maximum time expected to complete the mission. We assume Cozmo goes after the cubes in distance order with the closest cube first. Once collected, Cozmo returns to its original starting point, turns around, and deposits the cube behind it. Once all cubes are deposited, Cozmo returns to its starting location and completes its mission. Using POCL, we represent this plan as follows:

P = ⟨findCube(type1), pick_up(type1), cube1_at_p⟩,
⟨findCube(type2), pick_up(type2), cube2_at_p⟩,
⟨ findCube(type3), pick_up(type3), cube3 _at_p⟩

Notice that this plan does not ensure Cozmo collects cubes based on distance. Instead, Cozmo decides the ordering at runtime based on the distance from Cozmo to each cube. We look only at the time it takes to collect all cubes. The examination setup appears in Figure 54.

With the baseline established (column 2 in Table 28), we introduce a second Cozmo with the same collecting mission. We examine two different methods of coordination in this case. The first, the naïve method, allows both Cozmos to collect all cubes. Thus, their local goal is the same as the global goal, but the requirements for collection must be maintained. Each Cozmo uses the Bluetooth communication between the Raspberry Pis to send the cubes they have collected and deposited at their starting position. As a naïve method, it is possible that two Cozmo may attempt to collect the same cube, wasting resources. Each Cozmo uses the same POCL mission as the initial single-Cozmo approach.

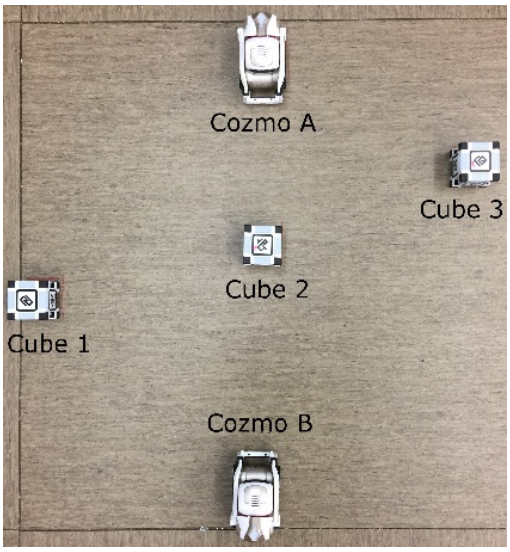



**Figure 54: Cozmo testing setup**

To implement our coordination method using self-integration, with the global goal of collecting all cubes and a local goal of collecting specific cubes. The Cozmos use adaptation A2 to broadcast their completed missions to each other. We only use A2 for our experimentation because this adaptation is most likely to cause self-integration between Cozmos, as Cozmos must adapt to a changing mission based on what the other Cozmo has done. We assign each Cozmo to collect cube 2 of Figure 54. Cozmo A is assigned cube 1's collection and Cozmo B is assigned cube 3's collection. In this setup, the cubes they are assigned to collect are far away from their starting position. Each Cozmo communicates the distances from their current positions to their cubes, and, if the Cozmo cannot find its cube from its initial search, informs the other Cozmo that it cannot find all of its assigned cubes. For self-integration, if one Cozmo cannot see an assigned cube and the other can, the Cozmo that can see the cube will adjust its mission to allow the collection of that cube. If a cube it is not assigned to collect is significantly closer than a cube it is assigned to collect, Cozmo will assign itself to collect the nearby cube. Each Cozmo assumes that, while it is collecting the nearest cube, the other Cozmo will also adapt to collect any cube it is missing to complete the global goal. The POCL representation for each Cozmo plan is listed below:

**Both Cozmos:**

P = ⟨findCube(type1), communicate(Cozmo), cube1_at_p⟩,
⟨findCube(type2), communicate(Cozmo), cube2_at_p⟩,
⟨findCube(type3), communicate(Cozmo), cube3_at_p⟩

After communicating, Cozmo A will recognize that it is closest to cube2 and cube3 while Cozmo B will recognize that it is closer to cube1 than Cozmo A. They adapt their final plans, such that Cozmo B will choose to pick up cube1, while Cozmo A chooses to pick up cubes 2 and 3. Their new POCL plans are shown below:

**Cozmo A:**

P = ⟨communicate(CozmoB), accomplishTask(type2), sufficient_knowledge⟩,
⟨pick_up(type2), cube2_at_p⟩,
⟨communicate(CozmoB), accomplishTask(type3), sufficient_knowledge⟩,
⟨pick_up(type3), cube3 _at_p⟩

**Cozmo B:**

P = ⟨communicate(CozmoA), accomplishTask(type2), sufficient_knowledge⟩,
⟨pick_up(type1),cube1_at_p⟩

**Table 28: Results of Cozmo tests**

| | Single Cozmo | Naïve Approach | Coordination |
|---|---|---|---|
| Run1 | 140.73 | 87.45 | 79.04 |
| Run2 | 153.37 | 84.79 | 87.30 |
| Run3 | 131.98 | 76.88 | 104.11 |
| Run4 | 141.95 | 81.45 | 79.18 |
| Run5 | 141.41 | 78.19 | 70.55 |
| Average | 141.89 | 81.75 | 84.03 |

We run each test 5 times and show the average in Table 28. For the single Cozmo approach, the average was 141.89 seconds as our baseline to improve on. For the naïve coordination approach, the average was 81.75 seconds, a notable improvement of slightly over 60 seconds. However, there were some problems observed between the two Cozmos in this approach, as occasionally they would attempt to go after the same cube. Usually, one Cozmo would arrive first, collect the cube, and leave before the other Cozmo could arrive. The second Cozmo would continue to attempt to collect the already collected cube rather than focus on another cube, as the alert that a cube was collected does not come until after the cube had been deposited, wasting some time.

For the self-integrating approach, where each Cozmo was in more constant communication about its current mission and could make adaptive decisions on its mission, the average was 84.03 seconds. Surprisingly, this is worse than the naïve approach, though with coordination Cozmo completed the mission in the fastest single time (Run 5). This difference can be attributed to Run 3, where the two Cozmo took 104.11 seconds to collect all cubes. We observed this time was due

to the method Cozmo uses to perform predefined actions. Cozmo is designed to be perceived as a virtual pet, requiring a personality. To simulate a personality in a robot, the developers have multiple methods of performing each action. In most cases, Cozmo will move toward the cube in a straight-forward manner, line up with the cube, and move forward to pick it up. However, in reviewing Run 3, Cozmo chose to slowly move forward as if to pounce on the cube, before suddenly turning and picking up the cube. This behavior took approximately 20 seconds more and accounts for the additional time.

#### 3.2.9.3 Integration of Self-Adaptive Testbeds

Testbeds are common when researching problems that require a large number of functionally similar but programmatically different parts (Siboni, 2016) or are too large or expensive to own a full version of the testable system though the individual parts are reasonably priced (Bellman, 2014). Testbeds are often designed to simulate a real environment as accurately as possible. However, corners may be cut to focus functionality so that the testbed can be used to exploit the specific problem that is being researched. Thus, while capable of advancing knowledge through direct experimentation, testbeds have limited usefulness outside of their problem domain.

At the same time, in real world operations there are interconnected systems that cannot be easily simulated using a single testbed. To address this challenge, multiple, smaller testbeds capable of examining specific problems could be linked together and collaboratively solve problems through inter-testbed connectivity. Additionally, by linking testbeds together, it becomes possible to improve the abilities of a single testbed without major modifications to the testbed architecture.

A unique problem arises when examining testbeds capable of self-adaptation. It is possible that a self-adaptive system will choose to adapt such that any other system relying on its operation will violate system requirements (Bellman, 2018), even if the adapting system maintains its individual requirements through the adaptation. This scenario is especially true when attempting to integrate multiple special-purpose or domain-specific testbeds, as each is capable of adaptation that has been verified for its specific use cases and runtime operations, but that may inadvertently affect the runtime stability of the linked system.

We have examined the difficulties of linking two existing, distinct testbeds. We outline use cases in which each testbed incorporates or is influenced by information from the other testbed to examine new problems. Though each testbed is capable of verifying its critical requirements and mechanisms for performing self-adaptation, they rely on different components, are implemented differently, employ distinct verification, and have unique approaches to data communication. We discuss the potential process of testbed cooperation based on their distinct factors.

#### 3.2.9.4 Existing Testbeds

Previously, we created two testbeds to examine self-adaptive and verification capabilities for different systems. The first testbed uses Raspberry Pi 3s to simulate near-future wearables, allowing the wearables to intercommunicate with Bluetooth and self-adapt their communication to potential security vulnerabilities (Walter, 2018a) as discussed in Section 3.2.5.1. Our second testbed focuses on mission collaboration between autonomous Cozmo robots discussed in Section 3.2.8.2.

#### 3.2.9.4.1 Integrating the Testbeds

Figure 55 shows the proposed architecture of a system with multiple intercommunicating, but distinct testbeds. On the left are the self-adaptive, coordinating Cozmos, focused on completing their local and the global missions. They would interact with the wearable testbed on the right through a Bluetooth link between base stations, similar to their normal communication with each other. The direct communication between testbeds through Bluetooth requires each testbed to parse data is it not designed to accept and has become a hindrance in integration. To solve this, we shift the testbeds to rely on the cloud to capture relevant testbed data for sharing, provide AI and machine learning techniques to discover potential adaptations and rationale for their use to influence the testbeds toward self-improvement, which may even impose additional requirements that are subject to verification and validation. Currently, each testbed makes use of the cloud in different ways. The wearable testbed would rely on cloud-based machine learning to discover insecure environments, prompting adaptation. The Cozmo testbed would use the cloud to primarily store information about the current mission and facilitate coordination, including potential adaptation needs. Testbed integration requires a cloud that both stores information and makes use of cloud-based AI and machine learning algorithms to understand the unique adaptation needs for both testbeds.

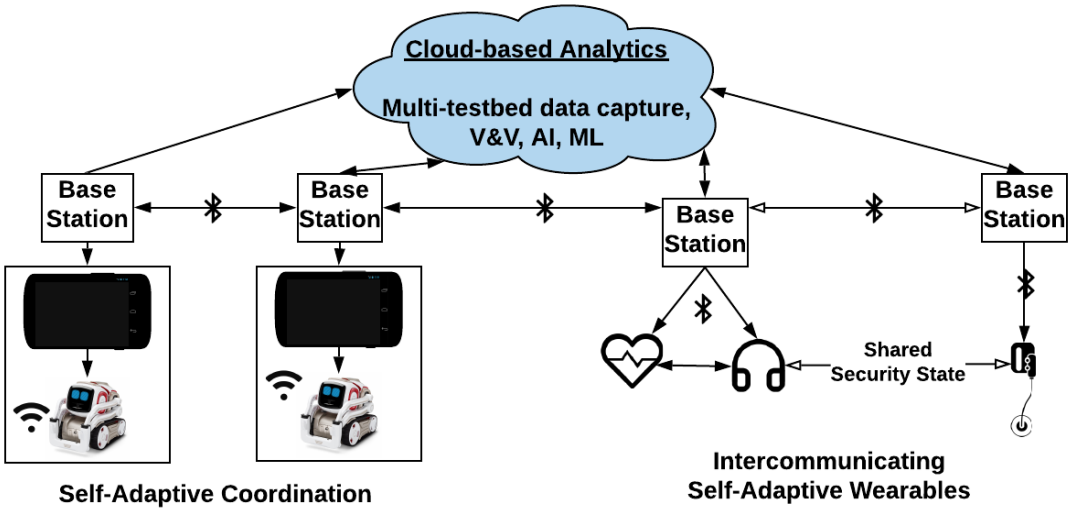



**Figure 55: Architecture for integrating testbeds**

If an adaptation in one testbed translates to the need for adaptation in the other, cloud services can intervene to prevent an infinite adaptation loop, testbed deadlock by forcing change at the wrong time periods, or other testbed from moving into a state that violates its requirements.

We describe three potential use cases that would benefit from integrating the two testbeds. The first is the scenario where an adaptation to a Cozmo's local mission is needed to complete the global mission. However, in order to complete the local mission, additional sensor data is needed from other nearby sensors. In this case, the adaptation plan will be shared between the Cozmo testbed and the wearable testbed along with a request for sensor information through the cloud. The base station of the wearables will respond with information about the available and active

sensors. If a needed sensor is available and active, the Cozmo testbed will be able to request the specific sensor information it needs and the data will be shared. The cloud will use cloud-based AI algorithms to determine the appropriate sensor information from the wearables, providing the format the wearables use for data communication to the Cozmo testbed and, if needed, translating this information to the Cozmo testbed in a format it can utilize.

An obvious example of this use case is using the Cozmo to simulate a search and rescue robot, searching for survivors in a disaster situation. The Cozmo can connect to a nearby base station, indicating that there may be a survivor in the area. When connected, it adapts to complete the global mission of finding survivors rather than continuing its local mission of searching a prescribed area. It then requests sensor information from the base station. The base station may be connected to wearables that include heartrate or other health information. The base station can be expected to have GPS data. This data is useful for the Cozmo's current adapted mission, so it requests heartrate and GPS data though the cloud. The heartrate information can be used to get a general sense of the health of the survivor and the GPS data can be used to pinpoint the location of the survivor for rescue. The machine learning algorithms running on the cloud can be trained to prioritize users whose health data shows the user needs immediate medical attention or those whose data shows increased health risk. While this would not result in ignoring survivors, it would allow survivors to be rescued in priority order, potentially resulting in a larger number saved.

A second use case is where the testbeds can be jointly used to assess adaptations produced by the cloud that can improve the communication between devices. The wearable testbed already includes adaptation options that result in adjusting the communication between wearables and their base stations. Thus, it is reasonable that an adaptation on the wearable testbed could influence the cloud-based decisions to change something about the internal or external communication with the Cozmo testbed. The two primary adaptations of the wearable testbed on Bluetooth adaptation are sending empty packets (to prevent eavesdropping on sensitive data) and disconnecting and awaiting a secure reconnection. In both of these cases, communication with the Cozmo testbed would be interrupted in some way. In the first case, the Cozmo testbed would remain connected, but would be unable to receive any data from the wearables, as only empty packets would be sent. In the second, the testbeds would disconnect and reconnection would only occur once the wearable testbed requests reconnection. These communication changes may force a new adaptation to the Cozmo as to how it can be interrupted when communication and data transfer are reestablished.

In both of these cases, the adaptation plan will be sent from the wearable testbed to the Cozmo testbed to inform the Cozmo testbed of the expected change to the communication. Once the Cozmo testbed is aware of the adaptation, it will not attempt to request data from the wearables, potentially limiting the available adaptations for the Cozmo but ensuring integrity of the wearable adaptations. In the case of disconnection, the Cozmo testbed will not attempt to force a reconnection and will not accept a reconnection request from any device other than the wearable testbed that caused the disconnection. However, as these adaptations only apply to the Bluetooth communication, it is still possible to gain information from the cloud, though this information will only allow old data, not up-to-the-minute information as is possible with the Bluetooth connection.

The third use case for testbed integration occurs when an adaptation is required by one testbed based on the expectation of an adaptation in the other testbed. For example, a Cozmo may be

simulating an autonomous vehicle, using the wearable testbed to simulate pedestrians and other vehicles in a simulated VANET. In this case, the Cozmo may detect the potential for a collision with another vehicle, requiring it to adapt to the potential issue. This adaptation needs to be transmitted to all other autonomous vehicles in the area. It may be beneficial to transmit this information to pedestrians so they can react, if possible. In this case, the Cozmo will send its plan to the wearable testbed before adapting. The wearable testbed must then adapt its operation to alert pedestrians to the issue and adjust the sensor information to react to the new Cozmo plan. In this case, a cloud service may be used to alert those not in the area, such as emergency personnel, if there is potential danger with the adaptation, providing a faster response time to a wreck.

#### 3.2.9.5 Difficulties with Testbed Integration

There are a number of challenges to integrating multiple testbeds. The largest issue is ensuring there is an existing and consistent protocol for the request and transfer of data between the testbeds. There have been middleware systems proposed (Burzlaf, 2019) to help solve this issue but, at present, there is no universal solution. It is especially problematic for Bluetooth communication, as it requires the protocol to be run and interpreted locally. Currently, the best option is to ensure that the receiver requests only specific data that it knows the other testbed has. It makes integrating additional testbeds difficult, as each new testbed requires all previous testbeds to be updated to accept the data of the new testbed.

An additional issue is the need for a method to communicate adaptation plans that can be parsed and examined by other testbeds. For example, with the current systems a message must be sent to inform all testbeds about an adaptation to ensure that the adaptation does not negatively affect the other testbed. However, this sharing requires all testbeds to have methods of handling all possible adaptations of all other testbeds. An intermediate system must be designed to allow a testbed to have an understanding of the more global effects of the adaptation. One assumption is that a cloud service could serve as the intermediary, providing an understanding of the possible adaptations each testbed can perform and the impacts both locally and propagated through the interconnection.

### 3.2.10 Evaluating the Use of GenProg within the Framework

#### 3.2.10.1 Revisiting the Multi-Mode Traveler System

In order to use GenProg (Goues, 2011) with our case studies, we adapted the Multi-Mode Traveler System (MMTS) to be repairable by GenProg. We chose MMTS because it has clearly defined requirements, has been proven to always maintain its requirements under normal operation, and has adaptations already created that were tested to show they could potentially cause it to fail to maintain its requirements.

The basic workflow for MMTS (as detailed in Figure 2) is shown in Figure 56. There are three main components that are executed during every update loop, each with subcomponents that have been proven to maintain the requirements. The first component, getCurrentStatus, checks the current position of the traveler to verify that the traveler is not already in a position that violates one of the three requirements. If the current status of the traveler does break one of these

requirements, it will not continue forward. If the current status satisfies both safety requirements, it moves to getNextPosition. In getNextPosition, the traveler gets information about the eight possible moves it has. It determines if the move would cause it to violate one of the safety requirements. It could violate the requirement either by gaining or losing fuel to push it outside the threshold or by attempting to move into a space occupied by an enemy. If there are no moves available, it returns to getCurrentStatus, as the traveler is unable to move without violating a safety requirement. Given the way requirement 3 is stated, if the traveler cannot move, whatever action it takes will not violate the requirement. If it can move, it shifts to setPosition, where the traveler is moved to a position that has been previously shown to not be in violation of the safety requirements.

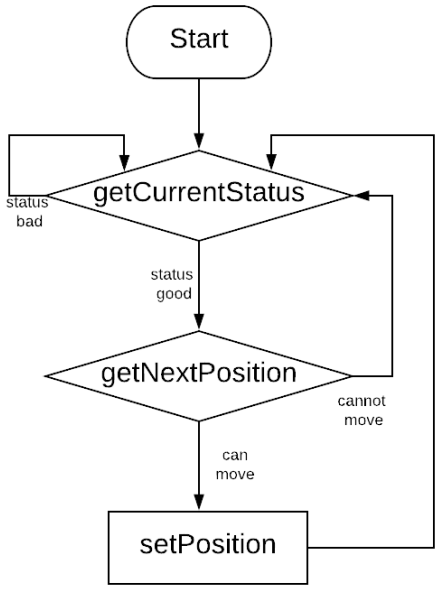



**Figure 56: Control Flow of MMTS**

### 3.2.10.2 Transitioning MMTS for GenProg

In order to use GenProg to repair MMTS, we must first convert MMTS from Java, the language it was originally written in, to C, the language the public version of GenProg is capable of repairing. This process was non-trivial even though the sample program is not very complex. We manually had to ensure that the C code (and use of its libraries) maintains the same architecture and functionality as the Java code, which was proven to meet the three requirements using the KIV theorem prover.

Figure 57 shows the CheckEnemy function in Java. Figure 58 shows the same function in C. Note the Java function is simpler, as Java includes the *contains()* function for checking the location of an enemy compared to the current position. The Java CheckEnemy code also includes a separate class/object containing the traveler's position information (both the currPos variable and the

enemyPos variables). In contrast, the C version of CheckEnemy uses arrays to represent the enemy positions and checks against x and y values represented as ints. The CheckEnemy function is run both during getCurrentStatus (Figure 56) to check if an enemy is currently at the same spot as the traveler, and in getNextPosition (Figure 56) to check if an enemy is in the position that the traveler may move to). The C version is longer, primarily because of the lack of built in contains() methods. Without contains(), the program must manually check if the location being examined contains an enemy by comparing all possible positions with the location of each enemy. If the enemy is in the position being checked based on the _x and _y coordinates, the C program returns a 1, equivalent to *true* in Java. Otherwise, the C program returns a 0, showing that there is no enemy in the checked location.

```
138     private boolean CheckEnemy(State state)
139     {
140         Position currPos = state.getTraveler().Get_Position();
141         Set<Position> enemyPos = state.getEnvironment().Generate_EnemyEnvironment();
142         if (enemyPos.contains(currPos))
143             return true;
144         else
145             return false;
146     }
```

**Figure 57: Example of CheckEnemy Code in the Java Version of MMTS**

```
204 int CheckEnemy(int ENEMY_POSITION[MAX_SIZE][MAX_SIZE], int X_POSITION[MAX_SIZE][MAX_SIZE]
205     , int Y_POSITION[MAX_SIZE][MAX_SIZE], int _x, int _y)
206 {
207     int enemyPos = 0;
208     for (int x = 0; x < MAX_SIZE; x++)
209     {
210         for (int y = 0; y < MAX_SIZE; y++)
211         {
212             if (X_POSITION[x][y] == _x && Y_POSITION[x][y] == _y)
213             {
214                 if (ENEMY_POSITION[x][y] == 1)
215                 {
216                     //waiting = 1;
217                     enemyPos = 1;
218                 }
219                 else
220                 {
221                     enemyPos = 0;
222                 }
223                 break;
224             }
225         }
226     }
227     return enemyPos;
228 }
```

**Figure 58: Example of CheckEnemy Code in the C Version of MMTS**

To work with GenProg, a bug must be forcibly introduced into the MMTS code for GenProg to repair. The bug should cause the traveler to violate a requirement. Instead of forcing a violation of

one of the three functional requirements directly, we introduced a mission requirement that could be violated through the normal operation of the functionally correct code. Our new requirement is that the traveler *must travel to four specific spots on the grid to pick up designated "targets"*. This could potentially be violated if enemies are placed on the grid that prevent the traveler from reaching the targets. By introducing the new requirement, we had to ensure that there were no random or non-deterministic actions that the traveler could take that would cause a violation of this new requirement that GenProg could not fix. To provide additional consistency for GenProg, we introduced a priority path into the code for the traveler so that it collects all targets. Should a position on the priority path not be a viable move given functional requirements 1 and 3, the traveler will move randomly outside of the priority path to comply with requirement 2.

An example of the code setting the priority path is shown in Figure 59. Essentially, the priority path is represented as a large switch statement based on the current move number. This new code provides GenProg with more code to use for repair of a bug and allows GenProg to adjust the priority path by only changing one or two of the switch cases. An example of the code that allows the traveler to choose the priority path when possible but still choose randomly when it is not possible to move along the priority path is shown in Figure 60. Note that, if the priority path position has already been shown to be invalid, it will skip the code for returning to the priority position. When it skips, it randomly searches all available options for a valid move on the grid. Once a move is found, it returns the index of the move, allowing the traveler to move.

```
int checkMovePriority(int moveNumber){
    switch(moveNumber){
        case 0:
            priorityPath[4] = 1;
            moveTraveler(priorityPath, moveNumber);
            priorityPath[4] = 0;
            break;
        case 1:
            priorityPath[4] = 1;
            moveTraveler(priorityPath, moveNumber);
            priorityPath[4] = 0;
            break;
        case 2:
            priorityPath[3] = 1;
            moveTraveler(priorityPath, moveNumber);
            priorityPath[3] = 0;
            break;
```

**Figure 59: MMTS Setting the Priority Path**

```
for(int i = 0; i < 8; i++){
    // printf("prioritypath[%i]: %i", i, priorityPath[i]);
    if(priorityPath[i] == 1 && validPosition_X[i] > -10000){
        priorityPath[i] = 0;
        return i;
    }
}
int randomIndex = -1;
while (randomIndex == -1)
{

    int index = rand() % 8;
    if (validPosition_X[index] == -100000 && validPosition_Y[index] == -100000)
    {
        randomIndex = -1;
    }
    else
    {
        randomIndex = index;
    }
}
return randomIndex;
```

**Figure 60: Priority Path Code vs Random Movement**

### 3.2.10.3 Preparing Additional Files needed for GenProg

With MMTS converted to C and the ability to insert a bug into the system, we began the final preparations of MMTS for GenProg's self-repair method. This preparation required creating test cases testing that validate the code will never go below the minimum fuel value, will never go above the maximum fuel value, and is able to collect all the targets. GenProg will run these tests to discover any differences that occur between the passing and failing test cases in an attempt to optimize the possible repair locations, weighting the paths unique to the failing test cases higher than the paths common between passing and failing test cases. Thus, GenProg will first attempt to make modifications only to parts of the code that are run when failing the test cases initially, though it will branch out if a repair has not been found.

For GenProg to execute the code and test cases, the preparation of a Makefile is needed that is capable of compiling the project, running the test cases, and outputting the test case results to a file for GenProg to examine. This Makefile is different from a standard Makefile that only compiles the code properly for eventual execution. We created this Makefile based on existing examples of GenProg Makefiles, ensuring the method of collecting the output from the test cases was identical to previous working versions. It also required minor modifications to the test cases, forcing them to print to the command line that they had failed without actually failing in a standard C test case manner. An example of the Makefile for one of the test cases is found in Figure 61. It

shows that, when the command “make collectSecondTarget.log” is run, the machine will first compile the test case. Once compiled, it will run the test case with all of the output sent to the file “collectSecondTarget.log”. This file is then re-written to the console, which GenProg can access to verify that the system has passed or failed.

```
collectSecondTarget.log: collectSecondTarget.c $(SOURCES)
    gcc -o collectSecondTarget collectSecondTarget.c $(SOURCES) $(CFLAGS)
    ./collectSecondTarget > collectSecondTarget.log
    cat collectSecondTarget.log
```

**Figure 61: Makefile Commands to Run Test Case Collecting Second Target**

An example test case is depicted in Figure 62. This test case checks if the traveler can collect the second target once it has collected the first target. Note that, even when it fails, it still returns 0 rather than returning 1 to alert to an error or failure. There are similar test cases for each of the other targets that can be collected, as well as test cases checking to ensure the minimum and maximum fuel is never exceeded.

```
1   #include "../MMTS/BaseComponent.h"
2
3   int main(){
4
5       setup(2,2,82,1);
6       for(int i = 2; i < 7; i++){
7           update(i);
8       }
9       if(!targetsCollected[1]){
10          printf("FAIL");
11          return 0;
12      }
13      printf("PASS");
14      return 0;
15
16  }
```

**Figure 62: Example Test Case Collecting the Second Target**

### 3.2.10.4 Running GenProg on MMTS

Having modified the test cases and created a working Makefile, we set enemies up directly in the path of the traveler. This configuration causes two of the test cases, collectFirstTarget and collectAllTargets, to fail. Because the test cases related to collecting the second, third, and fourth targets do not have an enemy directly in their path, they all still pass, though a change to the code may cause them to begin to fail. We created a configuration document for use with GenProg that contains the specific test cases that pass and that fail, as well as basic information about the

compilation and location of the Makefiles associated with MMTS. We then attempted to run GenProg to repair the failure to collect all targets bug in MMTS.

Once all of the setup of MMTS had been completed, we downloaded the existing GenProg virtual machines, with a working version of GenProg, to repair MMTS. Unfortunately, running GenProg on MMTS was impossible due to a major issue. Specifically, GenProg was unable to come up with a weighted list of different lines used between the failed and successful test cases. Initially, we attempted to minimize the C code by adjusting the location of the targets, thus decreasing the number of lines to create the priority path, but this did not solve the issue. We then moved to a fresh install of GenProg, taken from the official GenProg Github account. This install did not fix the issue, leaving the default GenProg only capable of brute-force attempts at repair, which all failed. To get GenProg working as expected, we contacted GenProg developers' team for advice on getting around the error. The team was unable to get GenProg working with our code. They suggested getting MMTS set up to use Darjeeling, an updated version of the GenProg code repair project that is language independent, which created the list of locations differently, and proposed that it would fix the issue.

#### 3.2.10.5 The Shift to Darjeeling

Darjeeling uses BugZoo to keep track of the associated passing and failing test cases, as well as the lines that are run in each. The information from BugZoo is used by Darjeeling to create a weighted path of possible repair locations, resolving the issue we had with GenProg. Additionally, Darjeeling contains the initial GenProg repair options, allowing us to run a slightly updated GenProg repair directly on MMTS. Running Darjeeling, we came up with over 500 candidate repairs. However, most of the repairs violate MMTS constraints. Two such repairs can be seen in Figures Figure 63 and Figure 64. Figure 63 is the first repair Darjeeling found. The repair is made to the CheckEnemy() function, shown originally in Figure 58. The issue here is that, rather than changing the priority path, Darjeeling removes the code to check if an enemy is in a given position (see the "-" preceding the code at lines 7-23 in Figure 63). This deletion causes the enemy blocking the priority path to be ignored, allowing the traveler to collect all targets. Darjeeling also adds a single line (see the "+" preceding the code at line 24) setting the priority path of one of the directions to zero. This addition does nothing functionally but is a byproduct of the genetic algorithm's attempt to repair the code. This solution would not be accepted as valid, as it breaks requirement 2 of MMTS.

```
1   --- MMTS/BaseComponent.c
2   +++ MMTS/BaseComponent.c
3   @@ -207,23 +207,7 @@
4       int enemyPos = 0;
5       for (int x = 0; x < MAX_SIZE; x++)
6       {
7   -       for (int y = 0; y < MAX_SIZE; y++)
8   -       {
9   -           if (X_POSITION[x][y] == _x && Y_POSITION[x][y] == _y)
10  -           {
11  -               if (ENEMY_POSITION[x][y] == 1)
12  -               {
13  -                   //waiting = 1;
14  -                   enemyPos = 1;
15  -               }
16  -               else
17  -               {
18  -                   enemyPos = 0;
19  -               }
20  -               break;
21  -           }
22  -       }
23  -   }
24  +       priorityPath[3] = 0;    }
25      return enemyPos;
26  }
```

**Figure 63: Example Repair from Darjeeling that Removes Enemy Checking**

Figure 64 shows a second potential patch crafted by Darjeeling. Specifically, it shows a pointless change (adding at line 8 a second “validPosition_Y[7] = -100000”) to the code that sets a valid position to an invalid position. Then, in the code for setting the priority path, it sets the targetsCollected value for the first target to true (line 17). This solution does not actually collect the target, as needed. Rather, Darjeeling evolves a solution that satisfies the requirement (collecting all targets) without actually needing to collect all targets. Because the randomization of MMTS results in the other targets getting collected based on the location after the random movement, this solution works, though it does break our new requirement because the traveler does not travel to the location required to collect the target.

```
--- MMTS/BaseComponent.c
+++ MMTS/BaseComponent.c
@@ -192,7 +192,7 @@
    }
    else
    {
-       validPosition_X[7] = -100000; validPosition_Y[7] = -100000;
+       validPosition_X[7] = -100000; validPosition_Y[7] = -100000;validPosition_Y[7] = -100000;
    }
    for (int i = 0; i < 8; i++)
    {
@@ -345,7 +345,7 @@
            case 13:
                priorityPath[2] = 1;
                moveTraveler(priorityPath, moveNumber);
-               priorityPath[2] = 0;
+               targetsCollected[0] = 1;
                break;
            default:
                moveTraveler(priorityPath, moveNumber);
```

**Figure 64: Example Repair from Darjeeling that Sets the Target to Collected**

### 3.2.11 Experimentation with Darjeeling and Genprog

One important thing to note with our implementation of Darjeeling is the difficulty of getting Darjeeling working with our code. While, in theory, Darjeeling/Genprog should be incredibly simple to get running, we have had repeated problems getting it running with our code. We began working with Genprog specifically, as it is a command line tool designed for use with the C language and seemed to be the simpler option. Using the existing bugs that Genprog can fix as a template, we built our MMTS C code from the ground up to be used by Genprog while still following the requirements of MMTS. Primarily this involved ensuring the Makefile used to build and test the code was formatted the way Genprog expected it and the tests printed out a pass or fail when they completed.

Once we had a version of MMTS in C that could be tested with the Makefile, we had to create a perl script to run the tests individually that checked if each test printed “PASS” or “FAIL” after completing and exited appropriately from that. This adds additional confusion when creating the tests, as the tests must print the result of the test (PASS/FAIL) but must not include an exit code that could cause Genprog to crash. This file also includes the complete list of test cases. With the Perl script, we must create a “test.sh” shell script to actually run the specific tests. This script also specifies which tests pass and which tests fail with the initial, unpatched code.

Genprog must also know the specific files that are able to be changed in the “bugged-program.txt” file. This can include either a single or multiple files. The file(s) listed in bugged-program.txt must be preprocessed through a pre-compiling command of the compiler and placed in the “preprocessed” folder. For our code, only a single file needed to be repaired, BaseComponent.c.

Finally, before running Genprog, we had to create a configuration file with the needed flags. We had to specify the type of search (genetic algorithm), compiler command when compiling the main code, test command for testing, type of crossover (for the genetic algorithm), the extension

of the files, number of passing and failing tests, bugged-program.txt, the location of the preprocessed folder, the test script command, the population size, the number of generations to run, information about the mutation rates/types, the cache location, and the location for the coverage path for positive and negative test cases. The coverage paths show the lines that are touched when running the file, so Genprog will optimize by initially attempting to change lines that only exist in the negative path.

Once all this had been created for MMTS, we can run Genprog to attempt to repair MMTS. We chose to use the existing Genprog virtual machines available from the repair benchmarks website (https://repairbenchmarks.cs.umass.edu/), as recommended to us when initially examining the use of Genprog. We used these machines as they have been shown to work with existing bugs that have been examined and ensured that no errors when running Genprog stemmed from an issue with installing Genprog from scratch.

When attempting to run Genprog on the MMTS code, Genprog would crash. We examined specific problems, tracking down small changes that needed to be made to the created scripts to allow the system to move forward, but eventually ran into a roadblock that we could not overcome. Specifically, Genprog could not find the path through the code and could not determine the lines that it could change, choosing to not make any changes rather than change randomly. This is odd, as it was able to run all the tests successfully and confirm MMTS was working as expected. After contacting the Genprog team, they recommended the shift to Darjeeling for examination.

Darjeeling has some similarities to Genprog in its approach. While it is language independent, it does use a very similar genetic algorithm approach as one of its possible repair methods, albeit with some optimizations and improvements. Most importantly, it uses the Bugzoo project to keep track of each bug and the path within the code of both the passing and failing test cases. While this requires a bit more installing and verifying, it does simplify the number of additional files that must be created for repair.

Darjeeling requires only a Dockerfile for creation of a docker instance to test the repairs, the file for running tests, the test.sh script, and two .yml files, one for Bugzoo and one for the Darjeeling repair function. The mmts.bugzoo.yml file lists basic information about the Dockerfile that is created and the location of the bug within the docker container that is created. It also specifes the passing/failing test case numbers and the type of repair to be tested. The mmts.repair.yml files defines the number of usable threads, language, specific files, algorithm information, type of transformations available, and potential optimizations for the repair function. With this, and with significant help and back-and-forth with the Darjeeling team (specifically Chris Timperley), we were able to get Darjeeling running successfully on our MMTS code.

#### 3.2.11.1 Examining Repair

Darjeeling repairs code based on test cases. It requires some passing and some failing test cases and will continue attempting a repair until all test cases are passing. For MMTS, we have a total of 7 test cases, two to ensure fuel consumption is within acceptable ranges, one to collect each target (with the traveler being relocated to the position it would be before collecting the previous target), and one to collect all targets. Before repair, the test cases for collecting all targets and collecting the first target fail. However, all other test cases succeed.

When running Darjeeling's repair, we utilize the '--continue' flag to continue searching for patches after an initially successful patch has been discovered. This provides more potential patches to analyze. For analysis, we created a script that applies the patch and runs the test cases against the patched code to verify the success of the repair. We have analyzed a total of 356 potential patches to MMTS. Of these, 138 pass 7 test cases. There are 188 patches that result in at least one failed test case and 30 that result in endless recursion and a stack overflow error, meaning they cannot be tested.

We then run the patches through an additional script that analyzes the type of fix we see most commonly. These are removing the traveler's ability to check for enemies (seen in Figure 65), setting the goal to have been collected without actually making it to the goal (Figure 66), and removing the X or Y coordinate from the environment (Figure 67). All 138 successful patches used one of these methods of patches.

Removing the enemy check, as seen in Figure 65, violates the MMTS requirement focused on avoiding enemies, as it allows the traveler to land on the enemy if it is in the way. This is done by deleting the line "enemyPos = 1;", but could be achieved by deleting more of the lines around it, as seen in the red lines in Figure 65. Setting the goal as already collected results in the newly introduced requirement related to collecting all the targets to be violated, as the traveler was unable to move to the position of the target. This is done by adding "targetsCollected[0] = 1;" anywhere in the code that is reachable when run, as seen in Figure 66.

```
--- MMTS/BaseComponent.c
+++ MMTS/BaseComponent.c
@@ -207,23 +207,7 @@
    int enemyPos = 0;
    for (int x = 0; x < MAX_SIZE; x++)
    {
-       for (int y = 0; y < MAX_SIZE; y++)
-       {
-           if (X_POSITION[x][y] == _x && Y_POSITION[x][y] == _y)
-           {
-               if (ENEMY_POSITION[x][y] == 1)
-               {
-                   //waiting = 1;
-                   enemyPos = 1;
-               }
-               else
-               {
-                   enemyPos = 0;
-               }
-               break;
-           }
-       }
-   }
+       priorityPath[3] = 0;    }
    return enemyPos;
 }
 int getChangeFuel(int _x, int _y, int _newPosX, int _newPosY)
```

**Figure 65: Removal of Enemy Check**

```
--- MMTS/BaseComponent.c
+++ MMTS/BaseComponent.c
@@ -345,7 +345,7 @@
            case 13:
                priorityPath[2] = 1;
                moveTraveler(priorityPath, moveNumber);
-               priorityPath[2] = 0;
+               targetsCollected[0] = 1;
                break;
            default:
                moveTraveler(priorityPath, moveNumber);
```

**Figure 66: Set First Target to Collected**

```
--- MMTS/BaseComponent.c
+++ MMTS/BaseComponent.c
@@ -28,7 +28,7 @@
    _y = Set_PositionY(y);
    enemy_Position(ENEMY_POSITION,manualEnemySet);
    x_Coordinate(X_POSITION);
-   y_Coordinate(Y_POSITION);
+   enemy_Position(ENEMY_POSITION, manualEnemySet);
    printf("x = %d, y = %d", x, y);
    printf("_x = %d, _y = %d", _x, _y);
 }
```

**Figure 67: Remove y_Coordinate**

Removing the X or Y coordinate from the environment is unusual. It does not directly affect the travelers code. It works because of the way the proven code sets enemies. The enemies are set based on the environment, while the traveler has its own internal knowledge of its location. By removing the X or Y coordinates from the environment, the enemies are set either the Y or X coordinate only. This means that the traveler can avoid the enemies by moving off the set environment to the targets. Figure 67shows the removal of the "y_Coordinate(Y_POSITION)" line that creates the Y portion of the environment. Adding additional lines here (such as "enemy_Position(…);", as seen in Figure 67) does not affect the patch, but is also not needed.

Of the 188 patches that result in a failed test case, 140 only failed a single test case, with 48 failing multiple test cases. The test case checking the traveler never goes above its maximum fuel value was the most common failing test case, failing 161 times total. The test case checking all targets were collected failed 44 times. The test cases for each individual target failed 27 times for the first test case, 7 for the second, 31 for the third, and 20 for the fourth. Finally, the test case checking that the traveler does not go below the minimum fuel value failed 9 times. This could be related to the required randomization of MMTS, though that would not account for the number of failures of the maximum fuel check. It is far more likely to be the cause of the failing tests for collecting specific targets, as a different choice of direction when avoiding an enemy can result in the traveler making it back onto its priority path and allow it to collect all targets.

The final 30 patches, those that result in the stack overflow error, all place the "moveTraveler" function call within the "moveTraveler" method, resulting in endless recursion. As there is not enough memory to store all the calls, the memory stack overflows and MMTS crashes. This means that none of the test cases are able to complete. While it is difficult or impossible to predict what patches may result in a stack overflow error (would require solving the np-complete halting problem), it is unusual to see these patches are seen as valid. It is possible this is a bug in the Darjeeling code.

We have run into one additional major issue with Darjeeling in our testing. Specifically, when using more than 40 possible patches in the population size or more than 10 generations, Darjeeling will freeze saying it has evaluated three candidates in a row without showing that it is evaluating other patches. These patches were queued up to be evaluated earlier in the run and other candidate

patches have been queued for evaluation, but it freezes after evaluating three candidates in a row. Interestingly, there are times where three candidate patches will be evaluated in a row before another candidate patch is queued with no freezing at all. There is nothing printed that would indicate what is causing the freeze. This freeze may be the result of Darjeeling repairing itself into an infinite loop with no timeout to kill the program, but this seems unlikely as, in our testing, this issue only occurred when either the population size is greater than 40 or there are more than 10 generations.

We have contacted Darjeeling team about the occasional freezing, the stack overflow issues, and the failing test cases. As of this writing, we are still communicating with them about the cause and the possible fix.

#### 3.2.11.2 Issues with Automatic Code Repair

While Genprog and Darjeeling can, in theory, be used to repair any code with minimal changes, there are some best practices that we have come across when adjusting MMTS for use with each of them. Both require extensive test cases covering all possible functionality and every edge case needed to ensure the repair does not break additional requirements. For MMTS, we are only able to test the requirements related to fuel and target collection completely and even within these test cases it is possible to modify the code such that the requirements are violated without violating the test cases.

This is especially true in repairs like removing the enemy check code and setting the targetsCollected variable to 1. Even though these violate the requirements of the system, they pass the required test cases. When designing a system that Darjeeling or Genprog will be used on, it would be best to move any code related to the requirements of the system into a file that will not be changed by the automated repair. While this may not always be possible, it does ensure that required code is not modified or removed during the repair. Alternatively, the test cases can be created in such a way as to detect when there is an issue that may result in a violation of the requirements without violating the “spirit” of the test case (that is, the specific section the test case is examining).

With the issues we have been having with Darjeeling, specifically related to the freezing on large populations or generations, it may also be best to only provide Darjeeling with relatively small code samples to maximize the likelihood a suitable repair is found in a reasonable amount of time. It should be noted that, quite often, a single repair is found within minutes, rarely more than 5. However, because a large number of our repairs failed to pass all test cases on re-evaluation, it is not a guarantee that the repair will be good.

It is possible that, at least for some of our failed re-evaluations, the randomness required by MMTS is the cause of the failing test cases. If automated code repair is expected to be used, the code should minimize the need of randomization. Ideally, the code would have no randomization involved. If randomization is required, the code should be built in such a way that the non-randomized portions can be tested with the test cases when repairing, allowing the repair software to repair with consistency. It is possible to use a seed for the randomizer to ensure the same random values are used each time, but this may lead to the solution only working for that specific seed, making it ineffective for normal operation.

For Genprog, it is important to have test cases that print the words “PASS” or “FAIL” rather than crashing, though crashing may be acceptable after the printing. This is because the Perl “run-tests.pl” file will crash if the test crashes. The “run-tests.pl” file looks at the output to determine if the test has passed or failed, meaning that modifications to existing test cases may be needed and, if the code is going to be repaired with Genprog, should be coded this way from the start.

# 4 Results and Discussion

Each of the prior sections have shown the accumulated results of the research, as well as the progression. The compliance awareness technology encompassing both functional verification and security certification with respect to adaptative and resilient systems has been defined and tested on multiple platforms and case studies. In addition, we have incorporated experiments with 3rd party technology to show where the gaps are in the research to have a fully automated MAPE loop.

To round out the results, we compare our Framework with two existing self-adaptive system frameworks, Rainbow and ActivFORMS, which are focused on model checking to verify predefined adaptation behavior. For comparison purpose, we discuss their adaptation approaches below. We implement to an allowable extent by each framework their overall approaches for Rainbow and ActiveFORMS using our MMTS case study discussed in Section 3.2.1. Due to issues with their available code bases we are unable to fully implement them even at the level of complexity for MMTS. However, given the literature available for the framework, we infer certain constructs for comparison. We develop MMTS model to perform model checking to verify adaptation behavior within a probabilistic model using PRISM, which is used by Rainbow, and a formal model using Uppaal, which is used by ActiveFORMS. Then we compare the allocated resource and time to perform the verification for three approaches as the comparison.

## 4.1 Rainbow

As an architectural-based self-adaptation framework (Garlan, 2004), Rainbow uses a high-level system architectural abstraction and model. The benefits of architectural-based approach is having system level global perspective and exhibits important system properties and constraints. Rainbow implements a system measurement mechanism which connect probes to the managed system’s model and queried about the state of the system. The model manager has access to query and update the model and execute constraint evaluator to check the model for violation. If the violation is detected, an adaptation engine triggers adaptation and carry out necessary action to modify the system through effectors. There is a translation repository that maps system information to the model and a resource discovery component having knowledge about the system resource and their properties. The adaptation manager incorporates adaptation knowledge about the changed model, changed components and their behavior. When the adaptation is triggered, the model manager queries the adaptation manager to get the appropriate model that satisfies the changes and enables adaptation operators to adapt the model. The adaptation strategy is chosen based on quality dimensions and utility preferences across the dimensions (Cheng, 2008).

To choose the tactics, Rainbow develops a probabilistic model of the system by including the probability of achieving the system property to incorporate the nondeterministic behavior of the system. Then they perform probabilistic model checking using PRISM. They simulate the different

tactics and collect the updated variable values to instantiate the model. For model checking, the system requirements are specified as model properties, which are checked against the model and provide the satisfiability level as a form of "Yes/No". They analyze the state space graph during verification to ensure both the trace of the system is valid and all states are reachable. This analysis is done external to the program that is being adapted. For our MMTS case study as discussed in Section 3.2.1, we specify its three requirements as three properties of the probabilistic model as shown below

**R1:** $A\ [\ fuel >= minFuel \& fuel <= maxFuel\ ]$

**R2:** $A\big[\big((curX = e1\_x\ \&\ curY = e1\_y)\big|(curX = e2\_x\ \&\ curY = e2\_y)\big|$
$(curX = e3\_x\ \&\ curY = e3\_y))\ = false\ ]$

**R3:** $A\ \big[\ \big(G\ canMove\ =\ true\ \&\ (X\ changePos\ =\ true)\big)\big|$
$(G\ canMove\ =\ false\ \&(X\ changePos\ =\ false))\ ]$

The property satisfiability level is considered as a quality objective. Rainbow uses a quality dimension corresponding to a business quality of concern, weights them based on the preferences, and generates a utility profile by specifying the bound for quality concerns. The formula for the utility function is below.

$$U = \sum (w_d u_d)$$

## 4.2 ActiveFORMS

Another architectural-based approach, ActiveFORMS (Iftikhar, 2017), has three primary components to perform the model checking. The first is the Managed System, which is instrumented to monitor the program and adapt the system according to presumed adaptive plans.

The two central components of this approach are: the Active Model Engine and the Goal Management. The Active Model Engine consists of a formally modeled feedback loop called the active model, which performs MAPE-K actions and a virtual machine, which performs the model checking. They develop the feedback loop using timed automata and use timed computation tree logic expressions (TCTL) to express the system behaviors as goals. Formal models with verified adaptation goals are deployed within the virtual machine and MAPE-K feedback loop that are connected with a model of the managed system. The model monitors the behavior by collecting information about the system through integrated probes and provides signals to the planner to determine adaptations. ActivForms has designed an exclusive goal model for different qualities of a system to incorporate them in modules with similar qualities. The models are deployed into the virtual machine and are translated into the graphs. When the planner initiates a plan for adaptation, the virtual machine executes all of the available graphs and performs verification on whether the goals are satisfied for the changed situation using model checking tool Uppaal. The system goal is specified as a Boolean expression and verified against the loaded formal model into the virtual

machine. Adaptation options are chosen based on verification results of the models. All of this occurs external to the program that is being adapted.

Uppaal has a built-in verifier called Verifyta, which is designed to make building verifiable queries easier. Using this, we specified three queries, one corresponding to each requirement MMTS has. The queries are:

1. $A[]\ Traveler1.currFuel <= maxFuel\ and\ Traveler1.currFuel >= minFuel$
2. $A <>\ Traveler1.start\ imply\ forall\ (x: int[0,2])\ Traveler1.currPosX\ != (enemyPos[2 * x]\ +\ enemyPos[2 * x])\ ||\ Traveler1.currPosY\ != (enemyPos[2 * x\ +\ 1]\ +\ enemyPos[2 * x + 1])$
3. $A[]\ ((Traveler1.canMove\ \&\&\ Traveler1.moveNum\ < 100)\ imply\ !Traveler1.noMove\ and\ !Traveler1.Fail)$

The Goal Management component supports developing a goal model, monitoring the goal, adapting, and managing the goal. The benefit of having the Goal Management component is it supports gradual improvement when system knowledge is incomplete during design time.

## 4.3 Comparison with our Approach

Due to the dynamicity of the environment and system, adaptation planning has to perform within a time limit and maintain the resource constraints. If the runtime verification of the adaptive plan maintains time and resource constraints, it provides more assurance about the system's maintainability of its requirements. We perform an experiment to compare the necessary resource and time needed to perform the planning phases in Rainbow, ActiveFORMS and our framework. We run the PRISM model of MMTS for 100 times for each of the adaptations discussed in Section 3.2.3.5. We found that Adaptation A1 and A4 have the potential to cause the system to fail to comply with all three requirements. But A2 and A3 maintain the system requirements for all 100 verifications through the PRISM model checker. In Table 29, the system failure percentage to maintain the requirements for all four adaptations are shown.

**Table 29: Percent Failure when Running PRISM Verification MMTS requirements**

| % Fail | R1 | R2 | R3 | Total |
|---|---|---|---|---|
| A1 | 46 | 0 | 0 | 46 |
| A2 | 0 | 0 | 0 | 0 |
| A3 | 0 | 0 | 0 | 0 |
| A4 | 0 | 0 | 100 | 100 |

We also analyze the CPU load and time needed for PRISM verification because every system has some resource limitation and if verification requires significant memory and time, it would be too costly to use at run time for a self-adaptive system. PRISM has a limitation on scalability and the allowed complexity of the system. Thus, we analyze the necessary resource allocation by PRISM for our small MMTS system. In Table 30, the average minimum and maximum CPU load along with the average time needed for verification for each adaptation by PRISM is shown.

**Table 30: Average CPU Load Running PRISM Verification of MMTS Model**

| | Max CPU load | Min CPU load | Difference | Time (ms) |
|---|---|---|---|---|
| A1 | 0.813783 | 0.152546 | 0.661237 | 2461.16 |
| A2 | 0.798274 | 0.186897 | 0.611377 | 1999.65 |
| A3 | 0.796227 | 0.187946 | 0.608282 | 2006.55 |
| A4 | 0.828876 | 0.130855 | 0.698071 | 2086.541 |

The difference shows the approximate impact of the verification on the overall CPU load which is almost more than 60%. This shows that PRISM verification causes a huge impact on the CPU. As MMTS is a small model, if PRISM causes a large CPU load, then for a large system it would be infeasible to verify. We found that PRISM needs almost 2s to verify all three requirements of MMTS. A real-world system with many requirements means their verification using PRISM would be infeasible for runtime verification. In a similar manner, we assess the Uppaal verification tool, Verifyta, to verify the active model of the MMTS as a feedback loop. The system failure percentage to maintain the requirements for all four adaptations for the MMTS active model are shown in Table 30.

**Table 31: Percent Failure when Running Verifyta on Abstracted Model**

| % Fail | R1 | R2 | R3 | Total |
|---|---|---|---|---|
| A1 | 50 | 0 | 0 | 50 |
| A2 | 0 | 0 | 0 | 0 |
| A3 | 0 | 0 | 0 | 0 |
| A4 | 0 | 0 | 100 | 100 |

Table 32 shows the average maximum and minimum CPU load when verifying the abstracted Uppaal model. The difference shows the approximate impact of the verification on the overall CPU load. Note that the load is minimal, with a maximum increase of 0.01 or 1% of the overall CPU. This shows that, for small models, Verifyta is efficient for verifying each adaptation.

Table 32 also shows the average time in milliseconds to verify all requirements. Note that the average hovers around 425 ms, showing that Verifyta is an eligible technology for verifying a model at runtime.

**Table 32: Average CPU Load/Time Running Verifyta on Abstracted Model**

| | Max CPU load | Min CPU load | Difference | Time (ms) |
|---|---|---|---|---|
| A1 | 0.467787 | 0.457698 | 0.010089 | 425.6 |
| A2 | 0.537985 | 0.537985 | 0 | 450.34 |
| A3 | 0.827726 | 0.82728 | 0.000446 | 445.19 |
| A4 | 0.716289 | 0.716289 | 0 | 408.17 |

Running the verification on the non-abstracted model shows a very different result. Table 33 shows the failure percentage of the non-abstracted model. Interestingly, because a deadlock occurs when maxFuel < currFuel, Verifyta is able to complete. If maxFuel > currFuel, Verifyta fails with an Out of Memory error. All other adaptations result in an Out of Memory error before any requirements have been verified.

**Table 33: Failure Rate of Verifyta on Non-Abstracted Model**

| % Fail | R1 | R2 | R3 | Total |
|---|---|---|---|---|
| A1 | 49 | 0 | 0 | 49 |
| A2 | Out of Memory | Out of Memory | Out of Memory | Out of Memory |
| A3 | Out of Memory | Out of Memory | Out of Memory | Out of Memory |
| A4 | Out of Memory | Out of Memory | Out of Memory | Out of Memory |

As one might expect, failing to verify with Out of Memory errors results in a much higher CPU load and much longer verification times, as can be seen in Table 33. There is a larger difference between the maximum and minimum CPU load, with the difference being close to 0.30 or 30% as shown in Table 34. Though A1 has the smallest difference, this can be explained when recognizing that A1 was able to complete 49 verifications due to their failure. As the difference is close to 0.15 between maximum and minimum, we can say that the difference during the potentially passing verification runs is around 30% as well. The average time for verification is significantly higher than with the abstracted model. The maximum time was 75819.2 ms, or 1.26 minutes. This is obviously not reasonable for runtime verification, showing that, if ActivFORMS is used, it must be as deterministic as possible to minimize the time it takes for verification, or should reach a stable state very quickly. It should be noted that, with the relatively short time it takes to verify the abstracted model, it may be possible to re-verify at each step with ActivFORMS.

**Table 34: Average CPU Load/Time Running Verifyta on Non-Abstracted Model**

| | Max CPU load | Min CPU load | Difference | Time (ms) |
|---|---|---|---|---|
| A1 | 0.63542 | 0.476547 | 0.158873 | 37554.33 |
| A2 | 0.693056 | 0.385565 | 0.307491 | 75819.2 |
| A3 | 0.708425 | 0.411999 | 0.296427 | 57814.32 |
| A4 | 0.685995 | 0.391253 | 0.294742 | 56767.65 |

We also have run our MMTS VFlow model to compare with Rainbow and ActivFORMS. In Table 35 we show that the VFlow model needs minimal time to perform the overall assessment along with assigning a minimal amount of CPU load. From minimum and maximum CPU load data, the VFlow model assigns a consistent amount of CPU load while performing risk assessment. With an increase in the number of requirements in the system, VFlow needs more time to complete the assessment. It is important to recognize that VFlow must examine all available adaptations, where PRISM and ActivForms may choose to examine one adaptation at a time until finding one that works. We have included the total time it takes to verify all adaptations on average, about 1911.91ms or 1.9 seconds.

**Table 35: Average CPU Load/Time Running VFlow Model for MMTS**

| | Max CPU load | Min CPU load | Difference | Time (ms) |
|---|---|---|---|---|
| A1 | 0.315633 | 0.306837 | 0.008796 | 335.52 |
| A2 | 0.428503 | 0.405414 | 0. 023089 | 372.82 |
| A3 | 0.416318 | 0.320624 | 0. 095694 | 604 |
| A4 | 0.364413 | 0.252124 | 0. 11229 | 599.57 |

VFlow provides the success probabilities computed from the alert token set given the design verification of system properties and the expected utility of the plan based on the probabilities and the requirements' utility weights. Table 36 shows the expected utility of the plan based on the probabilities and the requirements' of MMTS properties to be satisfied.

**Table 36: Risk assessment Using VFlow model of MMTS**

| | A1 | A2 | A3 | A4 |
|---|---|---|---|---|
| R1 | 0.007164045 | 0.006461016 | 0.334125 | 0.011390625 |
| R2 | 0.080380451 | 0.006461016 | 0.334125 | 0.011390625 |
| R3 | 0.006375625 | 0.006302470 | 0.4455 | 0.000001 |
| Expected Utility | 0.093920121 | 0.019224504 | 1.11375 | 0.02278165 |

The expected utility of the adaptation plan supports the PRISM and Uppaal's verification result for A1, A3 and A4, but not for A2. Though both PRISM and Uppaal provide 0% failure rate for A2, VFlow's expected utility does not conclude that A2 is a good adaptation option. Because A2 has altered multiple verification concerns on which the original verification process relied, this

alteration is assumed to inhibit the reuse of the verification process on the adapted. Rainbow and ActivFORMS just verify the properties based on established verification process model, but our framework determines the riskiness of a plan given alternative plans.

## 5 Conclusion

Run-time adaptation in systems poses major challenges. The first challenge is how to determine that the changes dictated by an adaption plan do not violate critical properties with which the system has been proven or certified to comply. A second challenge is determining if any proof or certification processes will be obstructed by the self-repair plan. A third challenge is how to measure the risk that an adaptive that may invalidate a property or proof yet is needed to prevent the failure of the overall system to complete its mission. To address these challenges, we constructed and demonstrated technology to perform compliance status assessment and risk assessment of an adaptation plan for both functional and security properties. The technology developed directly assesses the resilience of a system coupling architecture awareness, verification awareness, and security certification awareness using different modeling and assessment methods.

## 6 Future Work

Although we have had substantial achievements during this project, there is still a great deal of work to be completed to understand the long-term implications of this research. First and foremost, additional 3rd party technology must be evaluated to fully automate the MAPE-K loop and test the framework on plans that have not be designed for the experimentation. This technology includes determining uncertainty properties of the system environment, what should be monitored, and how plans can be dynamically formulated. Additional future work will include raising the level of abstraction and representation by which we describe entities within the framework to ensure they are broadly applicable.

## References


(Abie, 2012) H. Abie, and I. Balasingham, "Risk-Based Adaptive Security for Smart IoT in eHealth," Proceedings of the 7th International Conference on Body Area Networks, ICST, Oslo, Norway, 2012.

(Almeida, 2011) R. Almeida, and M. Vieira, "Benchmarking the Resilience of Self-Adaptive Software Systems: Perspectives and Challenges," Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, ACM, Waikiki, Honolulu, HI, USA, 2011.

(Bellman, 2014) K.L. Bellman, P.R. Nelson, and C. Landauer. "Active experimentation and computational reflection for design and testing of cyber-physical systems," Complex Systems Design & Management (Posters), 2014.

(Bellman, 2018) K. Bellman, "What reasonable guarantees can we make for a SISSY system?" 2018 IEEE 3rd International Workshops on Foundations and Applications of Self* Systems (FAS* W), 2018.

(Burzlaf, 2019) F. Burzlaff, C. Bartelt, "A conceptual architecture for enabling future selfadaptive service systems," 52nd Hawaii International Conference on System Sciences (HICSS 52), 2019.

(Calinescu, 2012) R. Calinescu, C. Ghezzi, M. Kwiatkowska, and R. Mirandola, "Self-adaptive software needs quantitative verification at runtime," Communications of the ACM, vol. 55, no. 9, pp. 69-77, 2012.

(Camara, 2013) J. Camara et al., Assurances for Self-Adaptive Systems: Principles, Models, and Techniques, vol. 7740, Springer-Verlag, 2013.

(CertWare, 2007) CertWare, https://nasa.github.io/CertWare/

(Cheng, 2008) S. W. Cheng, and D. Garlan, "Rainbow: cost-effective software architecture-based self-adaptation, " Carnegie Mellon University, Pittsburgh, PA, 2008

(Cheng, 2009) B.H.C Cheng et al., Software Engineering for Self-Adaptive Systems, vol. 5525. SpringerVerlag, 2009.

(Cordy, 2013) M. Cordy et al., "Model checking adaptive software with featured transition systems," Assurances for Self-Adaptive Systems, LNCS, vol. 7740, pp. 1–29. Springer, Heidelberg, 2013.

(Cotroneo, 2014) D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo, "A survey of software aging and rejuvenation studies," ACM Journal on Emerging Technologies in Computing Systems, vol. 10, no. 1, 2014.

(Cox, 2005) J. Cox and E. Durfee, "An efficient algorithm for multiagent plan coordination," In Proc. of the 4th Int'l Joint Conf. on Autonomous Agents and Multiagent Systems, 2005.

(Cozmo , 2018) "Cozmo | Meet Cozmo https://www.anki.com/en-us/cozmo, 2018.

(Damiani, 2011) F. Damiani, J. Dovland, E. B. Johnsen, and I. Schaefer, "Verifying traits: A proof system for fine-grained reuse," in Proc. of the 13th Workshop on Formal Techniques for Java-Like Programs, Lancaster, United Kingdom, 2011.

(Ernst, 2015) G. Ernst et al., "KIV: Overview and VerifyThis competition," International Journal on Software Tools for Technology Transfer. vol. 17, no. 6, pp. 677–694, 2015.

(Garlan, 2004) D. Garlan et al., "Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure," in Computer, vol. 37, pp. 46-54, 2004. doi:10.1109/MC.2004.175

(Goues, 2011) C. Le Goues et al., "GenProg: A Generic Method for Automatic Software Repair," IEEE Trans. on Software Engineering, 38:1(54-72), 2011

(Hale, 2017) M. Hale and R. Gamble, "Semantic Hierarchies for Extracting, Modeling, and Connecting Compliance Requirements in Information Security Control Standards," Requirements Engineering, Dec., pp. 1-38, 2017

(Hoare, 1985) C. A. R. Hoare, "Communicating Sequential Processes," Prentice Hall, 1985

(Iftikhar, 2014) M. U. Iftikhar, and D. Weyns, "ActivFORMS: Active Formal Models for Self-Adaptation," 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, 2014

(Jahan, 2017)S. Jahan, A. Marshall, and R. Gamble, "Embedding Verification Concerns in Self-Adaptive System Code," 11th IEEE International Conference on Self-Adaptive and SelfOrganizing Systems, IEEE, Tucson, AZ, USA, 2017

(Jahan, 2018) S. Jahan, C. Walter, S. Alqahtani, and R.F. Gamble, "Adaptive coordination to complete mission goals," 2018 IEEE 3rd International Workshops on Foundations and Applications of Self* Systems (FAS*W), 2018.

(Jensen, 2009) K. Jensen, and L.M. Kristensen, Coloured Petri Nets: Modelling and Validation of Concurrent Systems, SpringerVerlag, 2009.

(Jensen, 2015) K. Jensen, and L.M. Kristensen, "Colored Petri Nets: A Graphical Language for Formal Modeling and Validation of Concurrent Systems," ACM, Communications of the ACM, New York, Vol. 58, No. 6, pp. 61-70, 2015.

(Kephart, 2003) J.O. Kephart, and D.M. Chess, "The vision of autonomic computing," Computer, vol. 1, pp. 41–50, 2003.

(Kobayashi, 2016) N. Kobayashi et al., "Quantitative Non-Functional Requirements Evaluation Using Softgoal Weight," J. of Internet Services and Information Security, Institute of Engineering – Polytechnic of Porto, 6:1(37-46), 2016

(Lemos, 2013) R. de Lemos et al., Software Engineering for Self-Adaptive Systems II, vol. 7475. Springer-Verlag, 2013.

(Lichtenstein, 1985) O. Lichtenstein, and A. Pnueli, "Checking that Finite State Concurrent Programs Satisfy their Linear Specification," Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, ACM, New Orleans, USA, 1985.

(Marshall, 2018) A. Marshall, S. Jahan, and R. Gamble. 2018, "Assessing the Risk of an Adaptation using Prior Compliance Verification," Proceedings of the 51st Hawaii International Conference on System Sciences.

(Mylopoulos, 1992) J. Mylopoulos, L. Chung, and B. Nixon, "Representing and Using Nonfunctional Requirements: A Process-Oriented Approach," IEEE Trans. on Soft. Eng., 18:6(483-497), 1992.

(NIST, 2013) NIST, Assessing Security and Privacy Controls in Federal Information Systems and Organizations, Special Publication 800-53 Revision 4, NIST, 2013.

(NIST, 2014) NIST, Assessing Security and Privacy Controls in Federal Information Systems and Organizations. NIST Special Publication 800-53A Revision 4, 2014.

(Rushby, 2015) J. Rushby, "The Interpretation and Evaluation of Assurance Cases," Technical Report SRI-CSL-15-01, SRI International, Jul. 2015.

(Sharifloo, 2015) A. M. Sharifloo, "Models for Self-Adaptive Systems," Proceedings of the 2015 European Conference on Software Architecture Workshops, September 07-11, 2015, Dubrovnik, Cavtat, Croatia. doi>10.1145/2797433.2797457

(Siboni, 2016) S. Siboni et al., "Advanced security testbed framework for wearable IoT devices," ACM Transactions on Internet Technology (TOIT), 16(4), p.26. 2016.

(Tamura, 2013) G. Tamura et al., "Towards practical runtime verification and validation of self-adaptive software systems," Software Engineering for Self-Adaptive Systems II, R. de Lemos, R., Giese, H., Müller, and M. Shaw, (Eds.), vol. 7475, Springer-Verlag, 2013

(Walter, 2018a) C. Walter, I. Riley, and R.F. Gamble, "Securing wearables through the creation of a personal fog," Proceedings of the 51st Hawaii International Conference on System Sciences, 2018.

(Walter, 2018b) C.W. Walter, "The personal fog: an architecture for limiting wearable security vulnerabilities," PhD Dissertation, 2018.

(Wei, 2018) D. Wei, X. Zhang, and S. Mahadevan, "Measuring the vulnerability of community structure in complex networks," Reliability Engineering and System Safety, Vol. 174, pp. 41- 52, 2018

(Weyns, 2012) D. Weyns, M. U. Iftikhar, D. G. de la Iglesia, and T Ahmad, "A survey of formal methods in self-adaptive systems," Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering, pp.67-79, 2012. doi>10.1145/2347583.2347592

(Whittle, 2010) J. Whittle et al., "RELAX: a language to address uncertainty in self-adaptive systems requirement," Requirements Engineering, vol.15 no.2, pp.177-196, 2010.

(Zuo , 2011) Y. Zuo, and S. Lande, A logical framework for proof-carrying survivability," International Joint Conference of IEEE TrustCom- 11/IEEE ICES-11/FCST-11, 2011.

## List of Acronyms

AOM – Adaptation Operator Manager

AFRL – Air Force Research Labs

CPN – Colored Petri Net

CPU – Central Processing Unit

FBTL – Fuzzy Branching Temporal Logic

GSN – Goal Structuring Notation

HRVM – Heart Rate Variability Monitor

IP – Insulin Pump

IR – Infrared

KIV – Karlsruhe Interactive Verifier

LTL – Linear Temporal Logic

MAPE – Monitor-Analyze-Plan-Execute

MMTS – Multi-modal Traveler System

NIST – National Institute of Standards and Technology

POCL – Partial-Order, Causal-Link

SIMS – Smart Inventory Management Systems

SAC – Security Assurance Case

SCN – Security Control Network

SDK – software development kit

SGW– Soft Goal using Weight

SIG – Softgoal Interdependency Graph

STCP – Split Temporal Contract Proposition

TCP – Temporal Contract Proposition

TCTL – Timed Computation Tree Logic

VANET – Vehicular Ad-Hoc Network

VC – Verification Concern

VFlow – Verification Workflow