Version of Record: https://www.sciencedirect.com/science/article/pii/S0167739X21000443
Manuscript_40f33e4f2dc449318f774fc44b589a61

Evaluating Verification Awareness as a Method for Assessing Adaptation Risk

Ian Riley[1], Sharmin Jahan[1], Allen Marshall[1], Charles Walter[2], Rose F. Gamble[1]
1. Tandy School of Computer Science, University of Tulsa, Tulsa, Oklahoma, 74104, USA
2. Department of Computer and Information Science, University of Mississippi, University, MS, 38677, USA

Corresponding author(s)
Rose F. Gamble (gamble@utulsa.edu)

Abstract

Self-integration requires a system to be self-aware and self-protecting of its functionality and communication processes to mitigate interference in accomplishing its goals. Incorporating self-protection into a framework for reasoning about compliance with critical requirements is a major challenge when the system's operational environment may have uncertainties resulting in runtime changes. The reasoning should be over a range of impacts and tradeoffs in order for the system to immediately address an issue, even if only partially or imperfectly. Assuming that critical requirements can be formally specified and embedded as part of system self-awareness, runtime verification often involves extensive on-board resources and state explosion, with minimal explanation of results. Model-checking partially mitigates runtime verification issues by abstracting the system operations and architecture. However, validating the consistency of a model given a runtime change is generally performed external to the system and translated back to the operational environment, which can be inefficient.

This paper focuses on codifying and embedding verification awareness into a system. Verification awareness is a type of self-awareness related to reasoning about compliance with critical properties at runtime when a system adaptation is needed. The premise is that an adaptation that interferes with a design-time proof process for requirement compliance increases the risk that the original proof process cannot be reused. The greater the risk to limiting proof process reuse, the higher the probability that the requirement would be violated by the adaptation. The application of Rice's 1953 theorem to this domain indicates that determining whether a given adaptation inherently inhibits proof reuse is undecidable, suggesting the heuristic, comparative approach based on proof meta-data that is part of our approach. To demonstrate our deployment of verification awareness, we predefine four adaptations that are all available to three distinct wearable simulations (stress, insulin delivery, and hearables). We capture meta-data from applying automated theorem proving to wearable requirements and assess the risk among the four adaptations for limiting the proof process reuse for each of their requirements. The results show that the adaptations affect proof process reuse differently on each wearable. We evaluate our reasoning framework by embedding checkpoints on requirement compliance within the wearable code and log the execution trace of each adaptation. The logs confirm that the adaptation selected by each wearable with the lowest risk of inhibiting proof process reuse for its requirements also causes the least number of requirement failures in execution.

Keywords: Self-awareness, verification awareness, adaptation, risk assessment

1. Introduction

Self-improving system integration (SISSY) has become a prominent approach for effective self-adaptation where heterogeneous systems are interconnected [1]. Self-integration considers mutual influences among the actions, trustworthiness, performance, and collaboration of the interconnected subsystems. Integration often changes the system structure, affecting each subsystem's operational goals and exposing potential security vulnerabilities [2, 3]. To self-integrate systems with adequate results, each individual integrated system must be self-aware. The self-awareness we advocate is *verification awareness*. This form of awareness focuses system monitoring of and reaction to its own requirements compliance processes if it must adapt to integrate with another system or improve interaction with a new environment. With verification awareness, systems can manage their requirements compliance locally, rather than rely on global controls to manage them across an integrated system. Such higher-level controls have been shown to become unstable as a distributed system grows more complex, which limits the scalability of the integrated system [4].

When a system prepares for self-integration, it may incorporate one of many forms of self-awareness [5]. By doing so, it can adapt its functionality and communication processes while still protecting itself. For these actions, the system must be able to house and access embedded information regarding its processes, architecture, and critical requirements. It must allow for reasoning about uncertainty in its environment and enable technology that can assess the risk of an adaptation against presented alternatives prior to performing a runtime change. The need for these base capabilities leads to the recognition that maintaining the formal verification of a system's compliance with critical requirements is challenging. The system's operational environment may have uncertainties resulting in runtime changes that could violate requirements. Runtime verification (RV) can provide an effective solution when a system cannot be assured by conventional means. RV involves a process of formalizing a system as a set of specifications with observable inputs that support traceability of the system state [6]. However, runtime verification suffers when the verification framework is not fault-tolerant, the system has constrained computational resources, or state explosion occurs. Model-checking partially mitigates runtime verification challenges by representing the system abstractly, but performs external, not embedded, validation of the model consistency given an adaptation [7].

Runtime and design time verification can be supported by *compositional analysis*, which employs various techniques, such as *decomposition* [8, 9], *process algebras* [10], *labeled transition systems*, *reachability analysis* [11], and *refactoring* [12, 13], to address state space concerns and the scalability of formal methods [14]. The essential strategy of compositional analysis is to model the target system as a hierarchy of system specifications, where each specification can be verified by some composition of verified specifications that exist lower in the hierarchy. The lowest level of the hierarchy is comprised of system specifications that can feasibly be verified using formal methods. In this way, compositional analysis can address state space concerns by identifying the program states associated with specifications that have already been verified when applied to specifications that exist higher in the hierarchy. In practice, compositional analysis must often be applied to formal models of system behavior since source code does not neatly fit into such a hierarchy.

Runtime adaptation requires reasoning over a range of impacts and tradeoffs in order for the system to immediately address an issue. It is a significant research challenge to investigate how a model can be updated to remain effective at runtime as adaptations take

place [15]. Self-reflection [16] includes self-modeling, analysis, and decision processes that incorporate system integration status. One of the research challenges associated with self-reflection is the consistency and results validation of the current model. As integration into a larger system changes a local system's structure and/or functionality, reusing the prior model may no longer be valid. Other approaches that support runtime verification for self-integration do not have a framework that includes protocols, formats, and interfaces to provide strong guarantees that the required parts of a system model are maintained [17, 18, 19].

Verification awareness is a type of self-awareness directly related to system knowledge regarding its requirements compliance. The assumption is that adaptive systems have critical properties that have been formally proven at design time [20]. A runtime adaptation can alter one or more state variables at a point in the system code that could affect its scope or allowable value range. Doing so can interfere with the process and structure of the code analysis that was performed during the proof of a critical requirement. The interference constitutes an increased risk that the original, design time, verification (proof) process cannot be reused. The greater the risk of inhibiting proof process reuse, the higher the probability that a new proof process would be needed for the adaptive system to prove its compliance with a requirement. Needing an alternate proof process or strategy increases the risk that the requirement would be violated by the adaptation. Thus, a change to a particular state variable constitutes a *verification concern* that should be part of the systems' verification awareness across its critical requirements. The impact of that change should be measurable for the system to reason over an adaptation's viability at runtime.

In this paper, we extend our adaptation assessment framework [21] to embed verification awareness into a target self-adaptive system (SAS). We express critical safety and liveness properties in Linear Temporal Logic (LTL) [22] and prove them against their code using the KIV theorem prover [23]. We apply compositional analysis using a combination of decomposition and refactoring techniques to address state space concerns within the proof process. In addition, we demonstrate how decomposition can be applied to systems with LTL properties through the introduction of *temporal contract propositions* (TCPs) and *split temporal contract propositions* (STCPs). We devise heuristic rules to capture the meta-data that defines the structure of the proof process, such as state variables and state transitions that directly impact the requirement verification. This meta-data and system architecture are codified into an embedded Colored Petri Net (CPN) [24]. When runtime changes are needed, the CPN is executed by the system to calculate the number of impacted places in the proof process and the density of that impact by the adaptation. Risk assessment is performed on the CPN output among potential adaptations across all critical requirements to determine which among them provides the best opportunity for proof reuse. The application of Rice's theorem [25] to this domain indicates that determining whether a given adaptation inherently inhibits proof reuse is undecidable, suggesting a heuristic, comparative approach based on proof process meta-data is a viable solution.

We target an experimental testbed of heterogeneous, communicating wearable simulations (stress, insulin delivery, and hearables) on Raspberry Pi 3's within a personal fog [26], in which runtime verification and model checking would be too resource intensive. We predefine four adaptations. We apply a KIV proof to each wearable requirement and deploy the CPNs within each wearable to assess the risk among the potential adaptations. The results show that adaptations affect proof process reuse for wearable requirements differently. We separately evaluate the results by embedding requirement validation checkpoints into the wearable code and logging the execution trace of each adaptation. The logs confirm that the adaptation selected by each wearable during the risk assessment process also causes the least

number of requirement failures, which validates our approach to making the wearable verification aware.

The paper is structured as follows. Section 2 outlines the research on self-adaptation, self-awareness, runtime verification, and model checking. Section 3 discusses the wearable testbed that we use to employ verification awareness. Section 4 details the KIV proof strategies that allow for extracting proof process meta-data. These strategies include the compositional analysis techniques that use the introduced contract propositions to address state space concerns and the heuristic rules to direct the meta-data extraction. The methodology is applied to the testbed. Section 5 overviews the adaptation risk assessment process. Section 6 demonstrates how our proof process methodology can be applied to other theorem provers, evaluates the execution of adaptations within the testbed against the results of the risk assessment process, and provides a comparison of our adaptation assessment framework to the Rainbow framework. Section 7 provides discussion and concludes the paper.

2. Background and related work

A self-adaptive system operates in a dynamic environment and is expected to adapt its behavior automatically in response to situational changes to improve system reliability [27]. The reliability of a self-adaptive system refers to the system's resiliency to maintain compliance with requirements within a dynamic environment [28, 29]. This dynamism demands increased attention when considering SISSY initiatives with respect to the adaptation of self-integrating systems [1]. For effective system integration, incorporating self-awareness is necessary to assess integration status from the perspective of a system's relation to other systems and their social standing [2, 16]. Incorporating a form of self-awareness within the system is necessary to recognize the operational context for the change and reasoning needed regarding the choice of adaptation direction [17, 30]. Key issues associated with incorporating self-awareness into a system to allow for dynamic adaptation include 1) understanding the minimal amount of information that is required, 2) efficiently analyzing the information to produce actionable outcomes, and 3) enabling appropriate adaptation mechanisms to change the system's functionality, including runtime changes [16, 31].

Changes to a system's functionality can cause the system to no longer comply with certain requirements. This compliance interference may be necessary and acceptable in certain contexts. However, as part of the system's self-awareness, it should know the extent of its ability to maintain compliance, including where and how an adaptation impacts that compliance. Runtime verification is one approach to assess requirements compliance. System requirements are expressed using unambiguous specification languages, such as LTL or Computational Tree Logic (CTL) [22, 32]. These formal specifications are verified against runtime changes. Performing runtime verification for distributed systems in an interactive environment is challenging since each subsystem has local and global requirements [33]. Deriving specifications from requirements, precisely formalizing properties, and observing the behaviors that have changes are one of the RV's principal engineering challenges [6]. Moreover, a change can have a different impact on each system due to their dynamic characteristics and asynchronous computations. Network latency, non-deterministic behaviors, and independent failures are also involved in distributed systems' life cycle. Traditional RV assumes centralized computation, making it difficult to address a wide range of adaptation scenarios.

To address state space concerns, such as state space explosion, with traditional RV and design-time verification methods, authors can employ compositional analysis [14]. Clarke et

al. [8] apply decomposition to systems composed of many parallel threads to verify temporal properties in CTL. More recently, Cho et al. [9] implement BLITZ, which employs bounded model checking for decomposition to detect bugs in systems with over 100,000 lines of code. Yeh et al. [10] demonstrate how process algebras can be used to overcome state space explosion for processes and structures that can be simplified using a congruence relation. The most significant efforts in reachability analysis have been made by Cheung et al. [11], who show how context constraints can be automatically deduced from pre-specified system conditions. In most cases, system engineering results in systems that are not easy to verify using traditional forms of decomposition. In such cases, Cheng et al. [12] provide refactoring techniques to produce adequate models of complex systems that can be verified using traditional decomposition. Their work has been incorporated into ARCATS [13], a tool that procedurally refactors models in Promela (a modeling language) and reduces state space using branching bisimulation minimization.

Tamura et al. [34] define "viability zones" to separate out the concerns for verifying a requirement that can be impacted by changes and the set of viable states to be maintained so that system operation is not compromised. They examine those states at runtime to determine if a viability zone may be compromised by an adaptation. To formally verify a requirement at runtime, appropriate formalisms expressing system properties and their interactions within a dynamic environment are needed to assess system correctness. This process is very challenging for a self-adaptive system [6, 35, 18]. Filieri et al. [36] apply an offline verification process before the deployment of an adaptive system and verify adaptation changes at runtime by reusing the knowledge from that verification process. However, their approach requires having full adaptation knowledge prior to deployment, which may not be possible during a self-integration discovery and assertion process where adaptations can be configured and negotiated at runtime. We take a different approach by expressing how a requirement was proven and embedding that knowledge to determine the risk of inhibiting the original proof process. Thus, the adaptations can be configured and assessed at runtime.

Model checking is another prominent approach for validating and monitoring runtime behavior that may mitigate some of the runtime verification challenges. Blair et al. [7] proposed a concept *models@run.time* that extends model-driven engineering (MDE) approaches to support runtime verification. A model is an abstraction of the system using a description of a transition system that reflects the system's behavior and knowledge about its environment. Models are considered first-class entities for self-reflection and are employed as external entities in order to investigate the system's status with respect to its requirements [7]. Challenges with the integration of models@run.time for SISSY initiatives have been investigated [17]. One such challenge is supporting distributed self-reflection so that attempted improvements to the integrated system do not result in performance degradation [37] or even expose security vulnerabilities [38]. Model-checking introduces some additional challenges that include issues with model capture from code, appropriate abstraction of the model for effective checking, and providing efficient verification algorithms for verifications [18, 19]. Code should be structured in such a way that model extraction and verification becomes easier to automate and reduces human overhead [18].

There are a variety of system models that have been used for runtime model checking. Probabilistic model checking is currently very popular since it can employ stochastic models to incorporate uncertainty. These models are used to predict the reliability of the adaptation changes and improve the planning phase by updating the impact vector used as decision criteria for adaptive plan selection. Filieri et al. [36] develop a mathematical framework for runtime probabilistic model checking. The framework focuses on reliability requirements and models as a form of Discrete Time Markov Chain (DTMC), which describes a system's

interaction profile and failure probabilities. The framework is effective for systems with a limited number of variability points but struggles with state explosion problem. PRISM [39], a model checking tool to construct and analyze probabilistic models, is applied for different types of probabilistic models but has limitations with handling large-scale and dynamic systems.

Multiple researchers have proposed architectural-based model checking to provide general and reusable infrastructures with well-defined customization capabilities for self-adaptive systems in an effort to improve runtime verification [40, 41, 42, 45]. Garlan et al. [40] developed Rainbow, an architectural-based self-adaptation framework that uses a high-level architectural abstraction of the system as a model. The model manager accumulates system state information, updates the model, and assesses the system's properties and constraints against the updated model. When a violation is detected, the model manager activates an adaptation manager to get to an appropriate model that satisfies the changes based on static, pre-defined strategic rules. However, the system abstraction for modeling and defining rules for model evolution are based on system-specific knowledge acquired during design time. For an interactive, dynamic environment, pre-defined rules are not workable. Thus, incorporating runtime changes into the model to perform model checking continues to be a challenge.

Other researchers have pre-specified formal models of adaptive component expectations [43, 44]. Iftikhar et al. [43] implemented ActiveFORMS, a formal modeling approach for engineering self-adaptive systems. Formally expressed models and adaptation goals are pre-defined and assessed against changes using model checking tool, Uppaal [43], to determine an optimal valid model for guaranteeing the behavior correctness. Adaptation options are chosen based on model verification results. These options are verified against the model offline, so the outcomes must be translated and transformed back into the system. In addition, it is possible that adaptations configured at runtime may not be aligned at all with the adaptive component expectations because their potential behavior was not accounted for.

In prior work [38], we created a model for a risk-based assessment approach, which starts by formally expressing requirements and verifying them against the code by following the approach in [22]. To validate our proofs, we examined the outcomes of both manually proving properties and employing the KIV automated theorem prover [23]. Our prior work identified issues with automated theorem proving in the adaptive systems domain and extended our risk assessment process to security requirements [20]. Another method of runtime verification is to embed invariant checkpoints into code and perform runtime analysis on their performance to debug any errors that occur [45, 46]. We use this process as a way to validate our approach. However, it is not scalable for use in a deployed adaptive system.

While ubiquitous, wearables continue to face challenges with battery life, reconfigurability, mobility, security, and analytics [47], security vulnerabilities have been in the news because of data sharing, GPS tracking, and issues with Bluetooth communication. The NIST guide to Bluetooth [48] lists 27 known Bluetooth vulnerabilities and 8 Bluetooth threats. Multiple researchers [49, 50, 51, 52] have analyzed security on wearables and have shown that personal information can be exposed. To experiment with mitigating these challenges, the concept of a personal fog and associated application has been introduced, in which the wearables exhibit the future of edge computing capabilities [53, 54]. The personal fog architecture allows for improved situational awareness for wearables by aggregating data for analytics, increasing wearable security through a learned suspicious behavior, and allowing for communication to devices external to their personal fog.

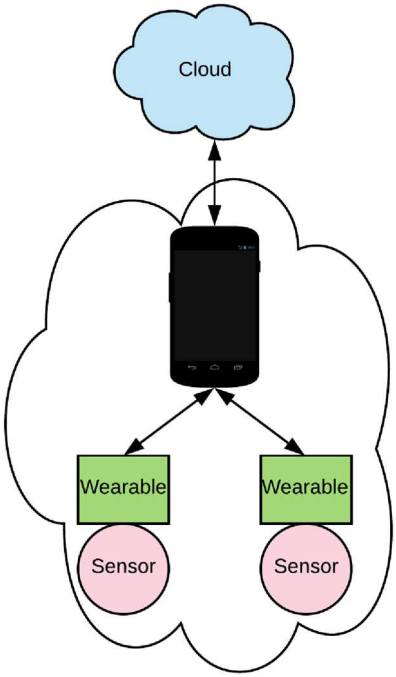

Figure 1: A single instance of the personal fog architecture containing two wearable nodes and a base station.

Figure 1 illustrates a personal fog architecture. A personal fog is defined as containing multiple, computationally powerful wearables at the edge with sensors and processing capabilities that connect to a base station owned by the wearable user [55]. The wearables and base station both act as personal fog nodes. The wearables in the personal fog can make local decisions based on their collected data. The base station is capable of analyzing the aggregated data of all connected wearables and communicating with the cloud to deliver information and analysis, as well as capture historical trends in the data. The personal fog also allows for intercommunication within the same fog [26]. This intercommunication allows edge nodes to form a personal distributed computing system and to provide additional computing power to each wearable. If needed, information can be shared between personal fogs, creating a large distribution of constantly moving compute nodes.

3. Wearable testbeds

To illustrate and evaluate our adaptation assessment framework for assessing adaptation risk, we use an experimental wearable testbed comprised of multiple personal fogs [26] that house the same application, allowing them to exchange security alerts. The testbed uses Raspberry Pi 3's to emulate both base stations and three distinct wearables: hearables, heart-rate variability monitors (HRVM), and insulin pumps [55]. The hearables simulation focuses on the expected capabilities of smart audio-streaming devices, such as the Here One [56], the Bragi Dash [57], and the Jabra Elite Sport [58] with requirements for streaming music and providing accelerometer data from a small earpiece. The HRVM simulation focuses on the stress monitoring capabilities of fitness-devices, such as the Garmin® fitness watches [59] that provide stress alerts as a requirement. The simulation of the insulin pump focuses on measuring a user's blood-sugar level and administering insulin as specified. The requirements for each simulation are not meant to be comprehensive. Rather, they are used to illustrate different wearable behaviors that can affect adaptation.

Each simulated wearable shares data and security status with its base station via a Bluetooth Low Energy (LE) connection. Figure 2 shows the architecture of the testbed. The Raspberry Pi 3's in the center simulate the functionality of the base stations. The Raspberry Pi 3's at the bottom, containing an additional screen for showing security status, simulate the wearables. A branch with a base station and its wearables forms a personal fog because, within the testbed, the simulated wearables have increased computational power for complex edge computing and, therefore, decision making. In addition, the base stations and their wearables all run the same security application that allows them to alert other personal fogs about potentially insecure environments.

The dotted lines in the figure represent the potential communication connections between fogs based on the security application usage [26]. The testbed components are programmed in Python. The base stations wait for a connection request from a wearable to establish a connection and collect and log data. The wearables attempt to connect to their base station. For each wearable device, Bluetooth functionality, e.g., establishing connections and monitoring connections status, and the wearable's primary functionality, e.g., operating as a hearable, are executed on parallel threads. This configuration provides a foundation for security experimentation and evaluation.

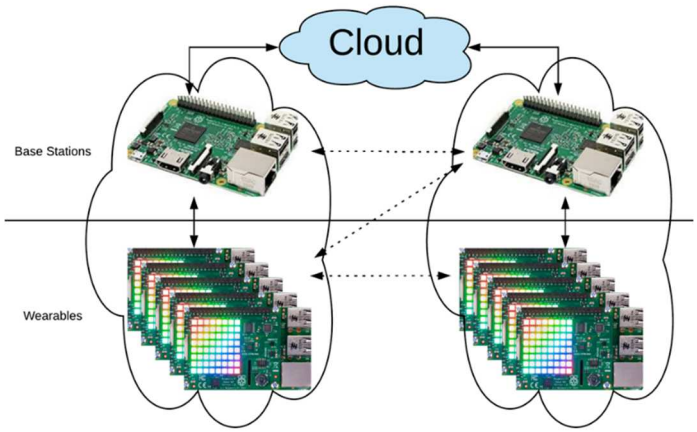

Figure 2: Wearable security experimentation testbed shows two base stations, each with four wearables. Dotted lines represent possible communication between fogs.

3.1. Targeting key wearable functionality

The pseudocode algorithms for the hearables, HRVM, and insulin pump are shown in Algorithm 1, Algorithm 2, and Algorithm 3, respectively. Each algorithm is provided in the dynamic language required by the KIV theorem prover. Variable names have been shortened to format each algorithm for print. The three wearable algorithms as well as the checkpoints that we use to monitor system behavior are all implemented in Java. The base station code is implemented in Python. *BuffChng* is a local variable that indicates if the buffer has been altered while the wearable is connected to an authorized user. The potential alterations include sending data in the buffer, overwriting the buffer, and increasing the buffer's capacity. This variable is later used during evaluation to determine if buffer integrity is being maintained in each iteration of the while loop. *MaxCap* is a variable that indicates the maximum physical capacity of a buffer and must be specified by the system implementer.

Then, an *infinite loop* simulates the lifecycle of a wearable device operating on a stream of

packets coming from its sensor(s). Each iteration of the loop represents one packet from the wearable's stream(s). Each packet contains raw sensor data and a timestamp. In the case of the hearables, there are two sensors, one for accelerometer packets and one for music packets. Accelerometer data is generated by the hearable while music data is received over a Bluetooth connection. We assume that both sensors input data at the same rate, so each iteration includes one packet of accelerometer data and one packet of music data. Each packet is considered nullable, where a null value indicates an empty packet, i.e., no raw data or timestamp. If the input data is assigned to a Boolean variable, then null values are treated as being equivalent to inputs of *false*.

```
def-program:
program(; I, Len, Cap, MaxCap, MusicIn, MusicOut, Buff, AccIn, AccOut, Conn, AuthConn, BuffChng) {
    I := 0;
    Len := length(Buff);
    Cap := Len;
    BuffChng := true;
    while true do {
        I := I + 1;                                                    ;; S1
        if AccIn ≠ null-opt then {
            Buff := Buff ++ AccIn.get-opt;
            AccIn := null-opt;
        };
        BuffChng := false;
        if Conn then {                                                 ;; S2
            if MusicIn ≠ null-opt then {
                MusicOut := MusicOut ++ MusicIn.get-opt;
                MusicIn := null-opt;
            };
        };
        if I ≤ Cap ∧ ¬ BuffChng then {
            if Conn ∧ AuthConn ∧ I < Cap then {                        ;; S3
                if Buff ≠ ∅ then {
                    AccOut := AccOut ∪ Buff;
                    Buff := ∅;
                };
                I := 0;
                Cap := Len;
                BuffChng := true;
            }
            else if Conn ∧ I = Cap ∧ I < MaxCap then {                 ;; S4
                Cap := MaxCap;
                BuffChng := true;
            }
            else if I = Cap then {                                     ;; S5
                I := 0;
                Cap := Len;
                BuffChng := true;
            };
        };
    };
};
```

Algorithm 1: Pseudocode algorithm for hearables.

The core of the algorithm is divided into five statements that compose the body of the *infinite loop*. In the first statement, S1, an index variable *I* is incremented by 1, accelerometer

data is input from the hearable's sensor and is stored in a buffer, and the variable *BuffChng* is set to *false*. The buffer is assumed to be 1-indexed so *I* is incremented at the beginning of each iteration rather than being incremented at the end. In statement S2, if the hearable is currently connected, it streams music data from the connected device. In S3, the hearable sends accelerometer data to the connected device. Accelerometer data can be both streamed and synced, where syncing is the sharing of data that has been buffered. In addition, accelerometer data is only shared with authorized devices. The environmental variables *Conn* and *AuthConn* are related to Bluetooth connection status. Environmental variables are variables that are mutated outside the scope of the program. In this case, both variables are managed by the thread that is governing the hearables Bluetooth functionality. In statement S4, if the hearable is currently connected but to a non-authorized device, it increases the capacity of its buffer to store additional accelerometer data so that the data can be synced if the connected device becomes authorized. However, in S5, if the hearable is not connected or if the buffer capacity has already been increased to its physical limit, as indicated by *MaxCap*, the index variable *I* is set to 0 and *BuffChng* is set to *true*, which indicates that the buffer can be overwritten.

Algorithm 2 presents the pseudocode algorithm used to simulate the heart-rate variability monitors (HRVMs). The hearables algorithm and the HRVM algorithm have the same initial conditions. In statement S1, the index variable *I* is incremented by 1, heart-rate data is input from the HRVM's sensor and stored in a buffer, a sync request is input from the Bluetooth sensor, the user's stress level is computed from their heart-rate data, and the local variable *BuffChng* is set to *false*. The HRVMs only sync data upon request from the base station, which is represented by the local variable *Sync* or the environmental variable *Stream*. Unlike *Sync*, which represents a request sent by the wearable base station, *Stream* represents a connection state and can be mutated by the thread managing the Bluetooth functionality. Both *Stream* and *Sync* are environmental variables. In addition, if a device is currently connected, heart-rate data is always shared once the buffer becomes full. The local variable, *Stress*, determines whether a user's heart rate indicates that they are currently in a stressed state. The value of *Stress* is calculated by the procedure *computeStressLevel*, which we assume can be tailored to individual devices and/or users. In statement S2, the heart-rate data is synced with the connected device if the device has requested to sync or if the buffer is full. Statement S3 sets the index variable *I* to 0 and sets *BuffChng* to *true*, which allows the buffer to be overwritten unless the user is stressed. If the user is determined to be stressed and the buffer has not reached its physical limit, then the capacity of the buffer is increased instead in statement S4.

The pseudocode algorithm, presented in Algorithm 3, for the insulin pump includes an additional initial condition *D*, which is the maximum blood sugar that the user is allowed and is used to determine when the user should be administered insulin. This value is set by the user. The insulin pump has five statements. In statement S1, the index variable *I* is incremented by 1, blood sugar data is input from the insulin pump's sensor and then stored in a buffer, an administer insulin request is input from the Bluetooth sensor, and the local variable *BuffChng* is set to *false*. The variable *AdminIns* is a Boolean variable that is set to *true* if the user requests insulin. Blood sugar data is shared with a connected device if the device is authorized in statement S2. Similar to the hearables, both *Conn* and *AuthConn* are environmental variables related to Bluetooth connection status. If no authorized device is connected and a user does not need insulin, then *I* is set to 0 and *BuffChng* is set to *true* in statement S3, which allows the buffer to be overwritten. However, if the user's blood sugar level is too high, as determined by the variable *D*, and the buffer has not reached its physical limit, then the buffer's capacity is increased instead in statement S4. Lastly, if the user has

requested insulin or if their blood sugar level is too high, then the insulin pump administers insulin in statement S5.

```
def-program:
program(; I, Len, Cap, MaxCap, In, Buff, Out, Conn, AuthConn, In2, Sync, Stream, Stress, BuffChng) {
    I := 0;
    Len := length(Buff);
    Cap := Len;
    BuffChng := true;
    while true do {
        I := I + 1;                                              ;; S1
        if In ≠ null-opt then {
            Buff := Buff ++ In.get-opt;
            In := null-opt;
        };
        if In2 ≠ null-opt then {
            Sync := In2;
            In2 := null-opt;
        };
        Stress := computeStressLevel(Buff);
        BuffChng := false;
        if Conn ∧ AuthConn ∧ ¬ BuffChng then {                   ;; S2
            if Sync ∨ I = Cap ∨ Stream {
                if Buff ≠ ∅ then {
                    Out := Out ∪ Buff;
                    Buff := ∅;
                };
                I := 0;
                Cap := Len;
                BuffChng := true;
            };
        }
        else if I = Cap ∧ ¬ BuffChng then {
            if ¬ Stress ∨ I = MaxCap then {                      ;; S3
                I := 0;
                Cap := Len;
                BuffChng := true;
            } else {                                             ;; S4
                Cap := MaxCap;
                BuffChng := true;
            };
        };
    };
};
```

Algorithm 2: Pseudocode algorithm for HRVM.

```
def-program:
program(; I, Len, Cap, MaxCap, In, Buff, Out, Conn, AuthConn, In2, AdminIns, InsAdmin, D, BuffChng)
{
    I := 0;
    Len := length(Buff);
    Cap := Len;
    BuffChng := true;
    while true do {
        I := I + 1;                                                    ;; S1
        if In ≠ null-opt then {
            Buff := Buff ++ computeBloodSugar(In.get-opt);
            In := null-opt;
        };
        if In2 ≠ null-opt then {
            AdminIns := In2;
            In2 := null-opt;
        };
        BuffChng := false;
        if I ≤ Cap ∧ ¬ BuffChng then {
            if Conn ∧ AuthConn then {                                  ;; S2
                if Buff ≠ ∅ then {
                    Out := Out ∪ Buff;
                    Buff := ∅;
                };
                I := 0;
                Cap := Len;
                BuffChng := true;
            }
            else I = Cap ∧ bloodSugarLevel(Buff) ≤ D ∨ I = MaxCap {  ;; S3
                I := 0;
                Cap := Len;
                BuffChng := true;
            }
            else I = Cap {                                             ;; S4
                Cap := MaxCap;
                BuffChng := true;
            };
        };
        if D ≤ bloodSugarLevel(Buff) ∨ AdminIns then {                 ;; S5
            InsAdmin := true;
        };
    };
};
```

Algorithm 3: Pseudocode algorithm for insulin pumps.

3.2. Fostering as a security alert

By experimenting with simulated wearables, we can use their programmability to explore communication options that wearables could, but currently do not, include. For example, simulated wearables can act as Bluetooth beacons that advertise the existence of Bluetooth servers and share data with other Bluetooth devices regarding the perceived security state of their mutual environment. We use the term *fostering* for this particular communication ability [26], which is especially important for devices that are constantly changing their location. As indicated by the dotted lines between portions of the different personal fog components in

Figure 2, fostering is designed to allow either the base station or wearable to connect temporarily to a service advertised by a Bluetooth beacon. This service is a separate server that can run on both wearables and base stations but is not always active. When a device connects to a fostering server, a single message is sent between devices before the connection is terminated. This allows for an exchange of information from other devices in the area, which introduces the opportunity for a rapid understanding of application-specific information that may not be known to devices entering the area. While fostering is designed to only send data between two devices, when it is implemented within a personal fog architecture, fostering essentially creates a temporary mesh network. This allows information to be passed between fogs through only a single temporary connection.

Fostering requires the devices to employ a separate application as part of their security protocol and has been evaluated against security threats in [55]. Fostering was employed to protect co-located personal fogs from Bluetooth eavesdropping, Man-in-the-Middle attacks, and Denial of Service attacks. When detected, either by a human recognizing the threat or unexpected data being received by the base station, the security state is sent through fostering. This resulted in the wearables adapting their communication to not send the requested data, successfully preventing eavesdropping and Man-in-the-Middle attacks. It does not prevent Denial-of-Service attacks, though it also does not provide the data to other users. Fostering was effective within a range of up to 8 meters, though it can, in theory, work up to the maximum Bluetooth range of 100 meters in open space.

Within the wearable testbed architecture used for the experimentation presented in this paper, the server is designed such that, when connected, it will send only the devices' current security state (either *secure* or *insecure*). Thus, devices entering a new environment can receive alerts regarding potential security threats from devices already in the area or devices that have recently adapted to become insecure based on threat knowledge. There is a risk to the device employing a fostering server by virtue of opening an additional connection point with no additional pairing or connection requirements. It is possible for an attacker to provide false information on a fostering server, though the fostering application only responds to an insecure state alert, not a secure state alert. It is also possible that, when examining fostering in a broader context, data that needs to be private or protected can be shared through fostering.

As a preview to the experimentation and evaluation performed in this paper, additional consideration is needed when fostering is part of an adaptation, since it may require modification to methods associated with the sending of data between connected devices. In the wearable testbed, adding fostering to the wearables requires modification of their code by integrating it with the fostering server application. For example, if an adaptation containing fostering is chosen, a new *send* command must be inserted to send the security state to external devices attempting to foster that are not authorized. Alternatively, the *send* command can be reused if the security state is added to the buffer and the wearable is permitted to send collected data to an unknown, potentially insecure device.

4. Proving functional requirements

Each wearable in the testbed has a set of functional requirements that it must satisfy. We present a sample of the requirements for each wearable to illustrate the approach. The requirements are partitioned across safety (invariant) and progress (liveness) properties. In this paper, all statements of functional requirements are followed by an expression stated in LTL [22] that extends first-order predicate logic to include temporal operators, such as always (□), eventually ( ), and next ( ).

The simulated hearable has two safety properties and two progress properties that govern the handling of its music and accelerometer data. They are as follows.

HR.1: Music is streamed from any connection.

$$\square\Big(\text{Conn} \rightarrow \quad \big(\text{MusicIn} \neq \text{null-opt} \rightarrow (\text{MusicIn.get-opt} \in \text{MusicOut})\big)\Big)$$

HR.2: Buffer is only sent on an authorized connection.

$\square(\text{send}(\text{Buff}) \rightarrow \text{AuthConn})$, where $\text{send}(\text{Buff}) = (\text{Buff} \cap \text{Out} \neq \emptyset)$.

HR.3: Accelerometer data that is collected is stored.

$$\square\big((\text{AccIn} \neq \text{null-opt}) \rightarrow \quad (\text{AccIn.get-opt} \in \text{Buff})\big)$$

HR.4: The buffer does not overflow.

$\square(\text{I} \leq \text{Cap})$

HR.1 is a progress property that translates from LTL to "it is always the case that if the device is connected, then eventually it will have streamed music." Thus, if the hearable is connected to a device, then it must reach a state where it is streaming music from the connected device infinitely often. It should be noted that if the connection ends abruptly, then HR.1 is still satisfied. If there is no connected device to send music data to the hearable, *MusicIn* becomes a null-opt. HR.2 is a safety property that states that "it is always the case that if the hearable shares accelerometer data, then it is on an authorized connection." HR.3 is a progress property that can be translated as "it is always the case that all captured accelerometer data will have eventually been stored in the buffer." HR.4 is a safety property that states "it is always the case that the buffer has not overflowed (i.e., the index variable, *I*, is never greater than buffer's capacity, *Cap*)."

The simulated HRVM has one safety property and one progress property, which ensure that heart-rate data is not lost. They are as follows.

HRVM.1: The buffer does not overflow.

$\square(\text{I} \leq \text{Cap})$

HRVM.2: Heart rate is stored when the user is stressed.

$$\square\big((\text{Stress} \wedge \text{In} \neq \text{null-opt}) \rightarrow \quad (\text{In.get-opt} \in \text{Buff})\big)$$

HRVM.3: Buffer is only sent on an authorized connection.

$\square(\text{send}(\text{Buff}) \rightarrow \text{AuthConn})$

HRVM.1 is the same safety property as HR.4. HRVM.2 is a progress property that can be translated as "it is always the case that if heart-rate data has been input while the user is stressed, then eventually the HRVM will have stored the heart-rate data in the buffer." HRVM.3 is the same safety property as HR.2.

The simulated insulin pump has two safety properties and one progress property that ensure buffer integrity, that connections are authorized, and that insulin is administered when necessary.

INS.1: The buffer does not overflow.

$\square(I \leq Cap)$

INS.2: Buffer is only sent on an authorized connection.
$\square(send(Buff) \rightarrow AuthConn)$

INS.3: Insulin is administered when needed.
$\square\big((D \leq bloodSugarLevel(Buff) \vee AdminIns) \rightarrow \ InsAdmin\big)$

INS.1 is the same safety property as HR.4, and INS.2 is the same safety property as HR.2. INS.3 is a progress property that states "it is always the case that if the user's blood sugar level is too high or if the user has requested insulin, then insulin will have been administered in the next state."

4.1. Employing the KIV theorem prover

To show that each wearable device complies with its functional requirements, we employ the use of the KIV theorem prover against the pseudocode algorithms provided in Algorithm 1, Algorithm 2, and Algorithm 3 that represent the implemented functionality for each wearable emulator. KIV allows for LTL expressions and can be used to prove both safety and progress properties. Our adaptation assessment framework requires the extraction of meta-data from the proof process or structure that will allow for adequate reasoning over dynamically configured adaptations prior to their runtime deployment. This section defines the approach to identify and capture the meta-data as it is strictly tied to the use of KIV and its features.

The first step in the process is to create a taxonomy of specifications for each wearable device. Each specification is a data type associated with axioms that KIV can use to construct proofs concerning variables of that type. Figure 3 presents the taxonomy for the insulin pump. The *blood-sugar* specification is an opaque data type, i.e., an object with unknown properties that is treated as a set, for blood-sugar level, and *blood-sugar-info* is a tuple type of *blood-sugar* and a *timestamp*, which is the *nat* type, or non-negative integers. Most of the specifications also include axioms that KIV needs to construct proofs, such as the *nat* specifications' axiom $n > i + 1 \rightarrow n > i$, which is used to prove HRVM.1 and INS.1.

The three taxonomies for the wearables use the same construction, with *set* specifications and *nat* specifications imported from KIV's core library. The *set* specifications are used to represent the buffer and an out-queue, which is employed to prove requirements where the device must output data, such as INS.2. The *nat* specifications are used for timestamps, the index variable *I*, and buffer capacity *Cap* to prove requirements that restrict buffer overflow, such as INS.1. Also included in the taxonomies are opaque types, tuple types, optional types, and set types. The opaque types are used to represent input data, such as blood-sugar data. These are enriched, designated by an incoming arrow, by a tuple type that joins the opaque data type with a timestamp. For example, *blood-sugar-info* enriches *blood-sugar* to represent a packet of raw blood sugar data and a timestamp. The optional specification is used to introduce nullable values. The *blood-sugar-info* specification is enriched with an *optional-blood-sugar-info* specification that allows the tuple type to be nullable. In addition, both the *blood-sugar* and the *blood-sugar-info* specifications are enriched with a set type. The *set-blood-sugar-info* specification, which enriches *blood-sugar-info*, represents the input streams, which stores *blood-sugar-info* tuples. Since blood-sugar tuples are nullable, only non-null values are

stored in the buffer. The *set-blood-sugar* specification represents an out-queue that is used to prove INS.2. That is, if the out-queue contains the blood-sugar data that was stored in the buffer, then the requirement holds.

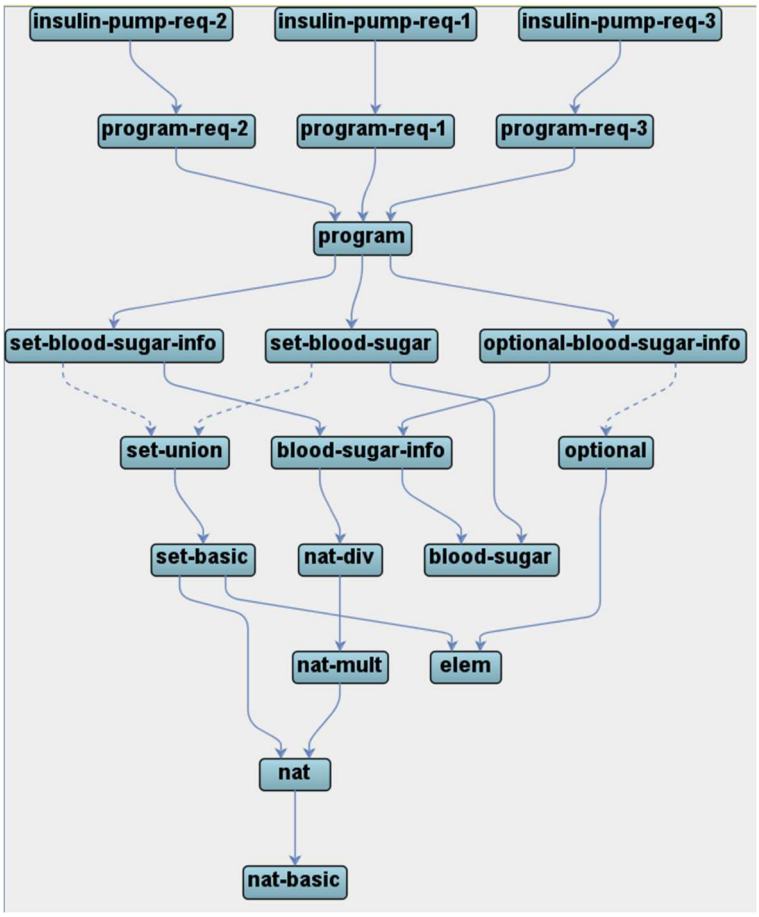

Figure 3: Insulin pump requirement specifications (top), program specifications (middle), type specifications (bottom), and dependencies as specified in KIV.

On top of the set and optional types rests the *program* specification. In this specification, we declare the pseudocode algorithm and all program variables using KIV's dynamic language. Each pseudocode algorithm from Section 3 is accurately represented, including its local, global, and environmental variables.

When constructing a proof, the KIV theorem prover will explore all possible states of the program. Each additional variable significantly increases the number of possible states, e.g., a Boolean variable will double the number of possible states, and each program statement can both increase the number of consecutive states and create branches within the proof. To reduce the program specification and proof complexity, we refactor program variables and program statements depending on the requirement that is being proven. Each time such a removal occurs, we ensure that the removed variable or statement has no impact on the target requirement. For example, to prove INS.1 and INS.2, we can remove program variables and program statements related to administering insulin as they would have no impact on those requirements. We could not however, remove those same program variables and statements when proving INS.3 as they are necessary to prove that the insulin pump can correctly administer insulin. In most cases, program variables will relate to some but not all requirements. As such, one reduction is performed per requirement, which results in one

reduced specification per requirement. Extra care must be taken during this process to incorporate hardware concerns. For example, the buffer's capacity (*Cap*) can never be ignored when considering storing data in a buffer, since every buffer has a physical limit to how much data it can store before it overflows. Reducing the number of program variables and statements also aids in the identification and capture of meta-data for verification awareness, which will be further discussed in Section 4.4.

KIV employs its own heuristic rules to prune branches that can be shown by an invariant to have no impact on the target requirement. For example, KIV can simplify conditional branching where each branch can be shown to satisfy the same desired properties. While we rely on refactoring and KIV simplification rules at this stage, designers can incorporate their own techniques by writing custom lemmas. Once defined, designers can apply their custom lemmas during the proof process to perform compositional analysis using such techniques as decomposition, as we demonstrate in Section 4.2 and Section 4.3. Alternatively, designers can employ their own techniques and invariants to simplify the proof process for their specific system implementation.

For the insulin pump, this process of reduction results in three specifications, which are *program-req-1*, *program-req-2*, and *program-req-3* shown at the top of Figure 3. Each reduced program specification is enriched by a corresponding requirement specification, e.g., *program-req-1* is enriched by *insulin-pump-req-1*. These final specifications are different than the others in two ways. First, they do not describe data types. Instead, they describe a requirement that needs to be proven against a program and program conditions that are necessary or helpful for constructing the proof. Lastly, they contain a lemma, stated as a sequent in sequent calculus [60], which is the requirement that needs to be proven. The KIV theorem prover uses the description provided in each requirement specification to prove its sequent against the enriched program for a single requirement.

4.2. Proving invariants with TCPs

When implementing a requirement specification, we can restate each program statement as a *temporal contract proposition* (TCP) [20] first defined as a way to tailor KIV's proof structure for meta-data capture. We show how its use can be newly explored as part of our adaptation assessment framework for the wearable experimentation testbed. A TCP is a tuple of the form

$$tcp(Pre, Code, Mid, Post),$$

which asserts that when *Code*, a program or program statement, is executed in a state that satisfies property *Pre*, then *Code* will terminate in a state that satisfies property *Post* and every state prior to termination will satisfy property *Mid*. By doing so, we can redefine a program statement with the following sequent

$$\mathrm{Pre}, [: \mathrm{V_{in}}, \mathrm{V_{inout}} \mid \mathrm{Code}(\mathrm{V_{in}}, \mathrm{V_{inout}}); \ [\mathrm{RestProg}]] \vdash \mathrm{Mid}\ \textbf{until}\ (\mathrm{Post} \wedge \mathrm{RestProg}),$$

where V_{in} and V_{inout} are a collection of program variables and *RestProg* is the remainder of the program that executes after *Code* terminates. If there are no remaining instructions after *Code* terminates, then *RestProg* is set to *true*.

If the requirement that is being proven is an invariant, such as $\Box(I \leq Cap)$, then it must be the case that, to prove the invariant, *Pre*, *Mid*, and *Post* must all imply $I \leq Cap$. For example,

after reducing the *program* specification to *program-req-1* (Figure 3) for the insulin pump, only program statements S1-S4 remain and statement S5 is removed. To prove requirement INS.1, we redefine statements S1 through S4 using the following TCPs. Note that the pre-condition of each TCP should mirror the *if* condition of each statement. The pre-conditions have been reduced for brevity. However, it should be clear from the statements that each pre-, mid-, and post-condition implies $I \leq Cap$, which is the invariant to be proven in INS.1.

$$\textbf{INS.Code.S1: } \mathrm{tcp}(\mathrm{I} < \mathrm{Cap}, \mathrm{S1}, \mathrm{I} \leq \mathrm{Cap}, \mathrm{I} \leq \mathrm{Cap})$$
$$\textbf{INS.Code.S2: } \mathrm{tcp}(\mathrm{I} \leq \mathrm{Cap}, \mathrm{S2}, \mathrm{I} \leq \mathrm{Cap}, \mathrm{I} = \mathrm{Cap})$$
$$\textbf{INS.Code.S3: } \mathrm{tcp}(\mathrm{I} = \mathrm{Cap}, \mathrm{S3}, \mathrm{I} \leq \mathrm{Cap}, \mathrm{I} = \mathrm{Cap})$$
$$\textbf{INS.Code.S4: } \mathrm{tcp}(\mathrm{I} = \mathrm{Cap}, \mathrm{S4}, \mathrm{I} \leq \mathrm{Cap}, \mathrm{I} < \mathrm{Cap})$$

Then, for each statement, V_{in} would include a collection of program variables that are needed for, but not modified by, program statements S1-S4, such as *Conn*, *AuthConn*, *D*, and *bloodSugarLevel,* while V_{inout} would include a collection of program variables that are both needed for and modified by program statements S1-S4, such as *I*, *Cap*, and *BuffChng*. By convention, V_{in} and V_{inout} are the same across all statements. Lastly, *RestProg* would include all subsequent statements with the current statement appended to the end. For example, for statement S1, *RestProg* would be equal to $[\mathrm{S2}, \mathrm{S3}, \mathrm{S4}, \mathrm{S1}]$. The current statement must be appended to the end since all statements are executed within an *infinite loop*.

In KIV, proofs are often constructed by contradiction. Therefore, to prove a property *p*, KIV will attempt to construct a proof for its contradiction, which can be stated as "there does not exist a natural number M that increases by one for each time step until *p* is false" and is expressed as

$$\neg\, (\mathrm{M} = \mathrm{M}'' + 1 \;\textbf{until}\; \neg\, \mathrm{p}),$$

where M is a natural number variable introduced by KIV for proof purposes. KIV performs the proof of contradiction by first assuming the hypothesis $M = M'' + 1\ \boldsymbol{until}\ \neg\, p$ and then applying a proof by induction using a well-founded induction over *M*. KIV will execute a full iteration of the main loop of the program, consider all program branches, and prove p at each step. If KIV can then reach a subgoal with a smaller value for M where p holds, then the inductive hypothesis can be applied to complete the proof by contradiction. It should be noted that, after assuming the statement $M = M'' + 1\ \boldsymbol{until}\ \neg\, p$, KIV asserts that there exists a value $M = m$, such that after step *m* is executed, the property $\neg\, p$ holds. Therefore, after each step where the property p holds, KIV asserts that it must have been that $M < m$, since it has been assumed that $M = M'' + 1\ \boldsymbol{until}\ \neg\, p$.

To perform a proof by induction for INS.1, KIV begins with the following proof goal.

$$\mathrm{M} = \mathrm{m}, \mathrm{Pre}, [: \mathrm{V_{in}}, \mathrm{V_{inout}} \mid \textbf{while}\ \mathrm{true}\ \textbf{do}\ \mathrm{S1}, \mathrm{S2}, \mathrm{S3}, \mathrm{S4}],$$
$$\neg\, \big(\mathrm{M} = \mathrm{M}'' + 1\ \textbf{until}\ \neg\, (\mathrm{I} \leq \mathrm{Cap})\big)$$
$$\vdash$$

Since we are attempting to prove a requirement that uses temporal operators, KIV requires that we specify environmental constraints as part of the initial conditions. Each program variable X has a corresponding variable X' and X'', where X' is the value of X at the end of the current state and X'' is the value of X at the beginning of the next state. If X'' is not equal

to X', then that means that X is being mutated in the environment outside the current program. For local variables, such as *I* and *Cap* in our example, or global variables that are locked, such as *Buff*, we use the constraint $X'' = X'$, so that these variables can only be mutated within the program. However, for program variables, such as *Conn* and *AuthConn*, that can be mutated outside the program, we use the constraint $X' \rightarrow (X'' = X')$. This constraint is necessary for statements where connection status is checked prior to sending data since KIV does not know that a send instruction would not be able to succeed if the device is not connected when the send instruction is called. Otherwise, KIV will create a branch representing the possibility that a connection is lost after the status is checked but before the send instruction is executed. In this branch, KIV cannot finish the proof as the branch will contain conditions *not connected* and *send*, which is a logical impossibility.

The most significant benefit provided by the introduction of TCPs is that, given KIV's approach to constructing a proof, we can use TCPs to skip large portions of proof steps that would otherwise be required to be performed manually. To do so, we derive a new proof goal and apply the following *lemma-invariant*.

$$
\begin{array}{l}
\mathrm{M} = \mathrm{m}, \\
\square(\mathrm{After_I} \wedge \mathrm{InvProp} \wedge \mathrm{M} \leq \mathrm{m} \rightarrow \neg\, (\mathrm{M} = \mathrm{M}'' + 1 \;\mathbf{until}\; \neg\, \mathrm{InvProp})), \\
\square(\mathrm{Mid_I} \rightarrow \mathrm{InvProp}), \\
\mathrm{Mid_I}\ \mathrm{until}\ (\mathrm{InvProp} \wedge \mathrm{After_I}) \\
\vdash \\
\neg\, (\mathrm{M} = \mathrm{M}'' + 1 \;\mathbf{until}\; \neg\, \mathrm{InvProp})
\end{array}
$$

To apply the *lemma-invariant* to INS.1, we make the following substitutions: $After_I = Post \wedge RestProg$, $Mid_I = Mid$, and $InvProp = I \leq Cap$. It is the case that, since $Post$ implies $InvProp$, Mid $\boldsymbol{until}$ $(Post \wedge RestProg)$, which can be derived from the TCP, implies Mid_I $\boldsymbol{until}$ $(InvProp \wedge After_I)$. KIV can determine that both Mid_I $\boldsymbol{until}$ $(InvProp \wedge After_I)$ and $M = m$ hold when the *lemma-invariant* is applied. Consequently, two new proof goals emerge that match the second and third formulae of the *lemma-invariant*. Using KIV's *execute always* rule, we reach the following new proof goals. KIV can automatically close the proof goal $Mid_I \rightarrow InvProp$ using its *simplifier*.

$$
\begin{array}{l}
\mathrm{After_I} \wedge \mathrm{InvProp} \wedge \mathrm{M} \leq \mathrm{m} \rightarrow \neg\, (\mathrm{M} = \mathrm{M}'' + 1 \;\mathbf{until}\; \neg\, \mathrm{InvProp}) \\
\mathrm{Mid_I} \rightarrow \mathrm{InvProp}
\end{array}
$$

To satisfy the remaining proof goal, we manually apply each TCP and follow each TCP with an application of *lemma-invariant*. Afterwards, we reach a proof goal where the program formula is:

$$[: \mathrm{V_{in}}, \mathrm{V_{inout}} \mid \mathbf{while}\ \mathrm{true}\ \mathbf{do}\ \mathrm{S1}, \mathrm{S2}, \mathrm{S3}, \mathrm{S4}]$$

This is the same program formula that we started with. We can use KIV's *induction* rule to close the proof by induction. KIV can deduce that the natural number variable *M* has decreased without a violation of the invariant. Given that the proof has reached a state where the properties are the same as they were for a larger value of *M* and the invariant has not been violated, KIV determines that *M* can increase indefinitely without violation of the invariant. This is sufficient to complete the proof by contradiction.

4.3. Proving progress properties with TCPs and STCPs

Just as we used TCPs to prove safety properties, we can similarly use them to prove progress properties. When using TCPs to prove progress properties, we must partition the code into *q-preserving* code and *non q-preserving* code. Given a progress property $\square(q \rightarrow \diamondsuit p)$, *q-preserving* code is code where the property q is satisfied before, during, and after execution. For example, if we are trying to prove INS.3, then program statement S4, which increases the capacity of the buffer, is *q-preserving* code since it does not affect the pre-condition of INS.3, which is $(D \leq bloodSugarLevel(Buff) \vee AdminIns)$. As such, we can restate S4 as

$tcp(q, S4, q, q)$, where $q = (D \leq bloodSugarLevel(Buff) \vee AdminIns)$, without loss of validity.

To prove a requirement with *q-preserving* code, we first apply the TCP, and then we apply the following *lemma-progress*.

$\mathrm{M} = \mathrm{m},$
$\square(\mathrm{After_P} \wedge \mathrm{M} \leq \mathrm{m} \rightarrow \neg\, (\mathrm{M} = \mathrm{M}'' + 1 \textbf{ until } (\mathrm{PreProgress} \wedge \neg \diamondsuit \mathrm{PostProgress}))),$
$\square(\mathrm{After_P} \wedge \mathrm{PreProgress} \rightarrow \diamondsuit \mathrm{PostProgress}),$
$\square(\mathrm{Mid_P} \wedge \mathrm{PreProgress} \rightarrow \bullet \mathrm{PreProgress}),$
$\mathrm{Mid_P} \textbf{ until } \mathrm{After_P}$
$\vdash$
$\neg\, (\mathrm{M} = \mathrm{M}'' + 1 \textbf{ until } (\mathrm{PreProgress} \wedge \neg \diamondsuit \mathrm{PostProgress})),$

where • is the operator 'weak next' indicating that *PreProgress* holds in the next state if there is a next state. This lemma is applied using the following substitutions: $After_P = q \wedge RestProg$, $Mid_P = q$, $PreProgress = q$, and $PostProgress = p$, where $p = InsAdmin$ for INS.3. By applying *lemma-progress*, KIV can essentially skip all instructions executed in the *q-preserving* code, which is a significant reduction in proof steps. This has no negative impact on the validity of the proof since, by definition, *q-preserving* code does not alter the pre-condition of the progress property. After applying *lemma-progress*, we can apply KIV's *execute always* rule to produce the following new proof goals.

$\mathrm{q} \wedge \mathrm{RestProg} \wedge \mathrm{M} \leq \mathrm{m} \rightarrow \neg\, (\mathrm{M} = \mathrm{M}'' + 1 \text{ until } (\mathrm{q} \wedge \neg \diamondsuit \mathrm{p}))$
$\mathrm{q} \wedge \mathrm{RestProg} \wedge \mathrm{M} \leq \mathrm{m} \wedge \mathrm{q} \rightarrow \diamondsuit \mathrm{p}$
$\mathrm{q} \wedge \mathrm{q} \rightarrow \bullet \mathrm{q}$

On the other hand, code can be *non-q-preserving*. Such code may either (a) not change q from *true* to *false* once p has been satisfied or (b) preserve some other condition, r, that ensures $\diamondsuit p$. To handle *non-q-preserving* code, we use *split temporal contract propositions* (STCPs) [20] which are defined as

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

where $Code$ is a program, or program statement, that is executed in a state that satisfies property *Pre* and either (a) eventually terminates with condition $Post_1$ and all intermediate states satisfying condition *Mid* or (b) eventually terminates with condition $Post_2$. Code represents the *non-q-preserving* code and $Post_2$ represents the intermediate condition, r, that ensures $\diamondsuit p$. We represent STCPs in KIV using the following template.

Pre, [: V_{in}, V_{inout} | Code(V_{in}, V_{inout}); [RestProg]]
⊢
(Mid **until** ($Post_1$ ∧ RestProg)) ∨ □($Post_2$ ∧ RestProg)

When proving a requirement that requires that use of a STCP, we first apply the STCP, then apply *lemma-progress*, and then apply the following *lemma-progress-split*.

$$
\begin{array}{l}
\mathrm{M} = \mathrm{m}, \\
\square(\mathrm{After_{PS}} \wedge \mathrm{M} \leq \mathrm{m} \rightarrow \neg\, (\mathrm{M} = \mathrm{M}'' + 1 \ \mathbf{until}\ (\mathrm{PreProgress} \wedge \neg \ \ \mathrm{PostProgress}))), \\
\square(\mathrm{After_{PS}} \rightarrow \ \ \mathrm{PostProgress}), \\
\ \ \mathrm{After_{PS}} \\
\vdash \\
\neg(\mathrm{M} = \mathrm{M}'' + 1 \ \mathbf{until}\ (\mathrm{PreProgress} \wedge \neg \ \ \mathrm{PostProgress})),
\end{array}
$$

lemma-progress-split is applied with the following substitutions: $After_{PS} = Post_2 \wedge RestProg$, $PreProgress = q$, and $PostProgress = q$. To prove INS.3, we use $Post_1 = Post_2 = q$. Application of *lemma-progress-split* results in the following open proof goals.

$$
\begin{array}{l}
\mathrm{q} \wedge \mathrm{RestProg} \wedge \mathrm{M} \leq \mathrm{m} \rightarrow \neg\, (\mathrm{M} = \mathrm{M}'' + 1 \ \mathrm{until}\ (\mathrm{q} \wedge \neg \ \ \mathrm{p})) \\
\mathrm{q} \wedge \mathrm{RestProg} \rightarrow \ \ \mathrm{p}
\end{array}
$$

By combining TCPs, STCPs, *lemma-progress*, and *lemma-progress-split*, we can skip to the end of the program's main loop. Once there, just as we did with the invariant proof, we can apply KIV's *induction* rule with respect to *M* to close the proof. This, along with KIV's *simplifier*, is sufficient to complete each proof goal introduced by *lemma-progress* or *lemma-progress-split*.

Together, *lemma-invariant*, *lemma-progress*, and *lemma-progress-split* can be employed by system designers to verify more complex system specifications using TCPs and STCPs. Applying these lemmas can address state space explosion by simplifying program variables and branches out of the proof process. We recommend that system designers apply the same methodology that we have presented in this paper by following these steps.

1. Create a model of the target system's behavior using the dynamic language provided by KIV.

2. Partition the instructions of the model into a sequence of Statements, where each Statement can contain one or more instructions. All instructions in the same Statement should have the same pre-condition, so that either (a) all of the instructions are executed sequentially or (b) none of the instructions are executed.

3. Refactor the model, where needed, for each requirement to reduce the number of Statements that are considered when proving the target requirement. System designers can use the heuristic rules provided in Section 4.4 to assist in identifying which program variables relate to each requirement and each Statement.

4. Decompose each refactored model depending on its target requirement. If the target requirement is an invariant, decompose the model by translating each Statement into a TCP. If the target requirement is a progress property, decompose the model by translating q-preserving Statements into TCPs and non-q-preserving Statements into STCPs. System designers can use heuristic rules provided in Section 4.4 to assist in identifying program variables that relate to each requirement to minimize the number of program variables that are included in the

pre-, mid-, and post- conditions of each TCP and STCP.

5. Complete the proof process for each requirement using the provided *lemma-invariant*, when proving invariants, and the provided *lemma-progress* and *lemma-progress-split,* when proving progress properties.

Our approach does not restrict system designers from using their own compositional analysis techniques to address state space concerns through custom lemmas, given those techniques are compatible with the heuristic rules defined in Section 4.4 for meta-data extraction. Furthermore, TCPs and STCPs can assist system designers in extending their own formal techniques to systems with temporal properties. In practice, systems may not be fully modular, which can make it more difficult to decompose the system into a neat set of TCPs and STCPs. In such cases, we recommend that system designers combine steps 1-3 to create a modular representation of their system's behavior in KIV's dynamic language for each requirement.

4.4. Capturing meta-data from proof process

Given the proof process described in Sections 4.1–4.3, we inspect each process and capture the verification concerns (VCs) in the form of program variables. To capture VCs, we inspect the various branches of the proof process and apply a series of heuristic rules as presented below.

Rule 1: When a requirement, stated as an invariant, specifies an independent property, p, that includes program variables, the program variables are captured as VCs.

For example, INS.1 uses the invariant $I \leq Cap$, which includes program variables I and Cap. As such, we capture I and Cap as VCs for INS.1.

Rule 2: When a requirement, stated as an invariant, specifies a dependent property, $q \rightarrow p$, that includes program variables, the program variables are captured as VCs.

For example, INS.2 uses the invariant $\square(send(Buff) \rightarrow AuthConn)$. From this invariant, we capture *send, Buff,* and *AuthConn* as VCs for INS.2.

Rule 3: When a requirement is stated as a progress property, $\square(q \rightarrow \ p)$, that includes program variables, the program variables are captured as VCs.

For example, INS.3 uses the progress property
$\square\big((D \leq bloodSugarLevel(Buff) \vee AdminIns) \rightarrow \ InsAdmin\big)$.

From this progress property, we would normally capture D, *bloodSugarLevel*, *Buff*, *AdminIns*, and *InsAdmin* as VCs. A program variable can be removed if it represents a procedure or parameter that is not subject to adaptation. Since this is the case with the procedure *InsAdmin*, it is removed from consideration as a VC. Rule 3 can be generalized to all progress properties that use the 'eventually', 'next', or 'weak next' operator.

Rule 4: For every symbolic execution step that applies KIV's *if positive* or *if negative* rule, the program variables in the condition are captured as VCs.

For example, when considering the proof for INS.2, data is output in S2, which uses the *if* condition $Conn \wedge AuthConn \wedge I \leq Cap \wedge \neg\ BuffChng$. The first time that this statement is encountered by KIV, the theorem prover will apply the *if left* rule, which results in two branches. The first branch represents the scenario where the *if* condition cannot be met, and the second branch represents the scenario where the *if* condition can be met. In the first branch where the *if* condition cannot be met, KIV can deduce that INS.2 is satisfied when either *Conn* or *AuthConn* is *false*. If *Conn* is *false*, *AuthConn* must be *false* by definition. This closes the first branch. In the second branch where the *if* condition is met, KIV returns to the statement during the second iteration of the main loop and applies the *if positive* rule. As such, we can capture the program variables in the *if* condition as VCs. A simpler approach, which is the approach that we take, is to capture the pre-condition of the corresponding TCP as a VC.

When applying Rule 4, we consider the pseudocode algorithm and the other requirements to determine if any VCs that could be captured can be reduced or removed. Doing so goes hand-in-hand with our earlier efforts to the reduce program variables and statements as any captured VCs that can be removed correspond to program variables that could have been removed before the proof was constructed. For example, when we apply Rule 4 to INS.2, we capture VCs *Conn*, *AuthConn*, *I*, *Cap*, and *BuffChng*. The VCs *I* and *Cap* can be removed since the invariant INS.1 requires that condition be satisfied. In addition, the VC *BuffChng* can also be removed since it is always set to *false* in statement S1 of the insulin pump. It is always the case that ¬*BuffChng* is satisfied. As such, after applying Rule 4 to INS.2, we only capture VCs *Conn* and *AuthConn*.

Rule 5: For every symbolic execution step that applies KIV's *call left* rule, the procedure and the program variables used for the V_{in} and V_{inout} of the procedure that is called can be captured as VCs.

For example, INS.3 relies on the procedure *bloodSugarLevel(Buff)*. The procedure *bloodSugarLevel* computes the user's blood-sugar levels from the blood-sugar data stored in buffer. This leads to the capture of *bloodSugarLevel* and *Buff* as VCs.

When we apply Rule 5, we consider the procedures that are called and their parameters to determine if the procedure or its parameters can be reduced or removed. Similar to the example in Rule 3, *computeBloodSugar* is a procedure that is undefined within the scope of the algorithm and is not subject to adaptation. This determination is based on the allowable range of adaptations to the system, so the procedure can be removed from the set of captured VCs. Parameters can be reduced if the parameter acts as a temporary variable. For example, the variable *Buff* in the insulin pump is effectively an array of values assigned to the local variable *In*. As such, *Buff* can be reduced to the variable *In* when captured as a VC. As such, when
Rule 5 is applied to INS.3, we only capture VCs *bloodSugarLevel* and *In.* This is because *InsAdmin* is removed, since it is not subject to adaptation, and *Buff* is reduced to *In*.

For those designers that use a theorem prover other than KIV, to be compatible with the heuristic rules that we have provided, the target theorem prover should be able to support sequents in both Hoare logic and temporal logic. Otherwise, there is no guarantee that the

proof conditions targeted by our heuristic rules will have an equivalent analog in another proof process. In such cases, extracted verification conditions may not be logically sound for use in the risk assessment process presented in Section 5. When using custom lemmas to support alternative techniques for performing the proof process in KIV, simplification steps must not eliminate applications of the *if positive*, *if negative*, and *call left* rules to be compatible with our heuristic rules.

After VCs have been captured, we begin to articulate various circumstances under which the condition specified in the VC would be violated. Each circumstance, or change, is then rated as 'devastating', 'worrisome', or 'unconcerned'. Each condition has a corresponding impact multiplier value, M_{VC}, of 0.2, 0.5, and 0.9, respectively. These impact multipliers are used as a heuristic when performing a risk assessment. A change that is rated as devastating violates the target requirement. If the change may, but does not assuredly, violate the target requirement, then it is rated as worrisome. Otherwise, the change is rated as unconcerned. In certain circumstances, a change may be rated as worrisome because it would violate an axiom that KIV used in the proof process even though it might not violate the requirement. For example, if a change were to set the variable *Conn* to *false* but leave the variable *AuthConn* set to *true*, then requirement INS.2 would not be directly violated. However, to prove INS.2, KIV employs an axiom that states that *AuthConn* must be *false* when *Conn* is *false*. As such, this change has the potential to violate proof reuse in such a way that it is unclear if KIV could still construct a proof since it would not be able to use the aforementioned axiom.

Table 1 presents the VCs, potential value changes, and ratings for INS.1, INS.2, and INS.3. For INS.1, we capture VCs related to program variables *I*, *Cap*, and *Conn*. VCs related to I and *Cap* are captured by the application of Rule 1. Devastating changes to I would include any change that directly violates INS.1, such as setting $I > Cap$, or any change that would directly lead to a violation of INS.1, such as removing $I := 0$. Worrisome changes would include incrementing I twice in a single state, which can take place if additional data needs to be stored in the buffer to facilitate fostering. Doing so does not guarantee that INS.1 would be violated but using a larger step size than 1 can allow I to become larger than *Cap* if $I = Cap - 1$. This would negate the pre-conditions of S3 and S4, which could have either set I to 0 or increased *Cap*, and result in a violation of INS.1. Devastating changes to *Cap* would include reducing the value of *Cap* when I is not set to 0 since that could directly cause a violation of INS.1 while unconcerned changes would include increasing the value of *Cap*. Lastly, we chose to capture *Conn* by Rule 4 since, when considering a physical system, the buffer has a maximum capacity to which it can be increased. Therefore, the program must connect to a device at some point or the buffer will be overwritten before data can be shared. Thus, devastating changes to *Conn* would include inhibiting all connections. *AuthConn* was not captured since INS.2 ensures that all connections must be authorized.

Table 1: Verification concerns and impact multipliers across insulin pump wearable.

REQUIREMENT INS.1 FOR INSULIN PUMP WEARABLE			
Verification Concerns (VCs)	**Devastating (0.2)**	**Worrisome (0.5)**	**Unconcerned (0.9)**
I	• Remove I := 0 • Set I > Cap	• Increment I twice in a single state change	
Cap	• Reduce Cap • Reset Cap to Len where is it not currently performed		• Increase Cap to Len where is it not currently performed
Conn	• Inhibit connection		
REQUIREMENT INS.2 FOR INSULIN PUMP WEARABLE			
Verification Concerns (VCs)	**Devastating (0.2)**	**Worrisome (0.5)**	**Unconcerned (0.9)**
Conn	• Set to true when connected to a non-authorized device	• Set to false when connected to an authorized device	• Set to true when connected to an authorized device
AuthConn	• Set to true when connected to a non-authorized device	• Set to false when connected to authorized device	• Set to true when connected to an authorized device
send	• Inhibit send while connected		• Send null data
REQUIREMENT INS.3 FOR INSULIN PUMP WEARABLE			
Verification Concerns (VCs)	**Devastating (0.2)**	**Worrisome (0.5)**	**Unconcerned (0.9)**
In	• Inhibit read		
AdminIns	• Set to false when instructed to administer	• Set to true when not told to administer	
bloodSugarLevel	• Prevent processing		
D	• Increase D	• Reduce D	

For INS.2, we captured VCs related to *Conn*, *AuthConn,* and *send*. *Conn* is captured by the application of Rule 4. *AuthConn* and *send* are captured by the application of Rule 2. Devastating changes would include setting either to *true* when the device currently connected is not authorized since doing so would be a violation of INS.2. Worrisome changes would include setting either to *false* when the currently connected device is authorized since that would contradict an axiom that KIV employs to prove INS.2. Inhibiting *send* while connected is considered devastating as that would directly violate the requirement as stated in KIV, while sending null data would be unconcerned.

Lastly, for INS.3, we capture VCs related to *In*, *AdminIns*, *bloodSugarLevel*, and *D*. *In* is captured by the application of Rule 5. Devastating changes to *In* would include inhibiting the read so that blood-sugar data and administer insulin requests can't be processed by the insulin pump. This would directly violate requirement INS.3. *AdminIns*, *bloodSugarLevel*, and *D* are captured by application of Rule 3. Devastating changes would include altering their values when insulin should be administered which would violate INS.3. Worrisome changes would include altering their values when insulin does not need to be administered, which does not violate INS.3, but could violate proof re-use and affect the user's health.

5. Risk assessment

For our experimentation, we specify four adaptations that the wearables in the testbed can employ if they enter an insecure state. These adaptations are as follows.

A1: Stay connected to base station; send empty packets (so there is no value in sniffing); do not participate in fostering

A2: Stay connected to base station; send empty packets; participate in fostering

A3: Disconnect from base station (sending no packets but collecting data locally); do not participate in fostering

A4: Disconnect from base station; participate in fostering

Each adaptation has a set of affected VCs and conditions in Table 1 that must be identified by the system designer. Adaptation A1 allows the wearable to stay connected without any fostering, which affects the *AuthConn* and *Conn* VCs. The condition for both *AuthConn* and *Conn* in A1 is "Set to true when connected to an authorized device". Since this statement expresses the default behavior of each wearable, the changes provide have an unconcerned impact (column 4 in Table 1). A1 also affects the *send* VC by changing its condition to "Send null data". This change has an unconcerned impact according to the insulin pump's requirements. Adaptation A2 allows the wearable to remain connected but allows fostering. The change condition for *Conn* is the same as A1 but fostering may set *AuthConn* to true when there is an unauthorized connection. This setting is considered is a devastating condition (column 2 in Table 1) for *AuthConn*. A2 also sends null data, where the impact is unconcerned. Adaptation A3 disconnects the device but does not permit fostering. The change condition for *Conn* is "Set to false when connected to an authorized device" but the conditions for *send* and *AuthConn* are same as in A1. Adaptation A4 disconnects the device and allows fostering, so the condition for *AuthConn* is "Set to true when connected to a non-authorized device" and connected is "Set to false when connected to an authorized device." Both conditions have a devasting impact on *AuthConn* and *Conn*.

The notation VC_A refers to the VCs affected by adaptation A, where A is A1, A2, A3, or A4, e.g., $VC_{A1} = \{Conn, AuthConn, send\}$. The notation $conditions_A(vc)$ refers to the change conditions of the verification concern vc with respect to adaptation A, e.g., $conditions_{A1}(Conn) = \{"Set\ to\ true\ when\ connected\ to\ an\ authorized\ device"\}$. When assessing how an adaptation affects a particular requirement, VCs in Table 1 that are not associated with that requirement are ignored. For example, though VC *send* is part of adaptation A1, *send* is ignored when assessing INS.1, since it is not associated with INS.1.

As wearables operate in a dynamic environment, a security solution requires some knowledge of their situation, which provides the capability to prevent security threats. The situational knowledge may be obtained from historical scenarios in which the security of the device was vulnerable. Using this knowledge, designers assign a change impact value $\hat{p}$, which is a heuristic that quantifies the planner's belief that the adaptation will result in a successful outcome. We assign $\hat{p}$ for each adaptation based on situational knowledge for the insulin pump as shown in Table 2.

Table 2: Change impact for adaptations A1-A4 on insulin pump.

$\hat{p}$(A1)	$\hat{p}$(A2)	$\hat{p}$(A3)	$\hat{p}$(A4)
0.55	0.65	0.60	0.75

Using the captured meta-data (Table 1), impact multipliers (Table 1), and change impacts (Table 2), we employ our adaptation assessment framework based on the meta-data, code architecture, and developer input to evaluate the risk that an adaptation inhibits the re-use of the proof process [38]. This risk assessment approach provides the system with an opportunity to select the most appropriate, or least risky, adaptation from the set of potential adaptations. From each requirement and proof process, we embed a verification workflow (VFlow) based on a Colored Petri Net (CPN) [24] that models the software architecture of the system and the proof process to output the risk alerts for each adaptation.

A CPN is a bipartite, directed graph that includes a collection of *places* and *transitions* as vertices. In general, colored *tokens,* representing different data types, traverse the CPN places according to a set of pre-defined *transition rules*. Places, transitions, tokens, and transition rules are defined for the VFlow CPN. The VFlow design restricts each place to allow only tokens of a single color. Transitions in the VFlow design allow for all colored tokens.

Figure 4 depicts the VFlow for the insulin pump with its three architectural components as described in Algorithm 3. These components are *Initialize*, *Adjust*, and *Administer*. *Initialize* represents the initialization instructions (as specified in S1). *Adjust* represents the changes to the buffer (as specified in S2, S3, and S4). *Administer* represents the blood sugar level assessment and the insulin administration (as specified in S5).

Each component in Figure 4 is specified using two places and a transition. *Initialize* includes the places *Initialize* and *Initialize Pink,* with *Initialize Transition*. *Adjust* includes the places *Adjust* and *Adjust Pink,* with *Adjust Transition*. *Administer* includes the places *Administer* and *Administer Pink,* with *Administer Transition*. Designers assign a place impact multiplier, M_{PL}, to each component to quantify the significance of the component in the requirement proof process. For simplicity, we assign 0.5 to each component of the insulin pump algorithm for each requirement as shown in Table 3.

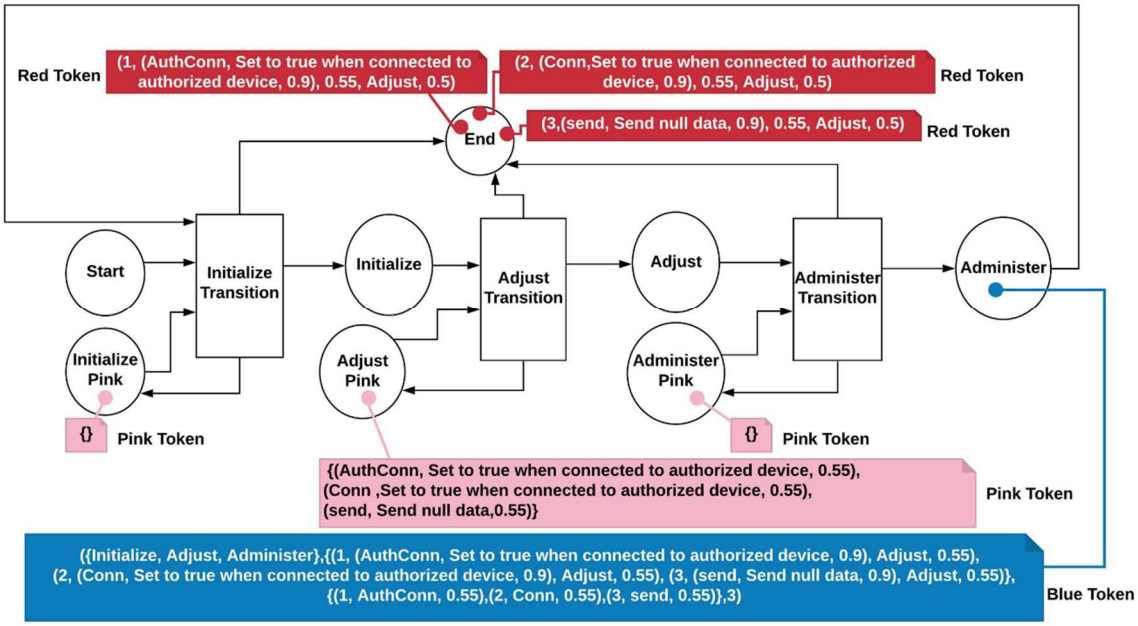

Figure 4: VFlow CPN for insulin pump's INS.2 for adaptation A1.

Table 3: Place impact multiplier for the insulin pump's VFlow.

	Initialize	**Adjust**	**Administer**
INS. 1	0.5	0.5	0.5
INS. 2	0.5	0.5	0.5
INS. 3	0.5	0.5	0.5

There are two additional places, *Start* and *End*, that must also be included in the VFlow, also shown in Figure 4. The *Start* place is the starting point of VFlow execution and expresses the initial state of the verification process. The *End* place accumulates tokens that express risk alerts when conflicts are present because of adaptation changes to the VCs extracted from the requirement proof process. Each of the eight places presented in Figure 4 acts as a container for specific colored tokens. The places *Initialize Pink*, *Adjust Pink*, and *Administer Pink* are containers for *pink tokens*. The places *Start*, *Initialize*, *Adjust*, and *Administer* are containers for *blue tokens*. *End* is a container for *red tokens*. Each of the three transitions are used to perform complex processing based on embedded verification knowledge, and to allow token instantiation and traversal through the VFlow. Colored tokens may only traverse their respective places through transitions.

Pink tokens represent the qualities of the adaptation as input. There is one pink token for each adaptation. For the VFlow, the pink token P_A for an adaptation A is defined as

$$P_A = \{(vc, condition, \hat{p}(A)) \mid \forall\, vc \in VC_A, \forall condition \in conditions_A(vc)\},$$

where

- $\boldsymbol{VC_A}$ are the VCs affected by adaptation A,
- $\boldsymbol{conditions_A(vc)}$ are the conditions of the change to a vc due to adaptation A, and
- $\hat{p}$ is the change impact value associated with adaptation A (Table 2).

When the VFlow is instantiated, pink tokens are constructed and assigned to their corresponding places, which are *Initialize Pink*, *Adjust Pink*, and *Administer Pink* in Figure 4. As shown in Figure 4, empty pink tokens are assigned to the *Initialize Pink* and *Administer Pink* places since these architectural components are not affected by adaptations A1-A4. Table 4 provides the pink token specification for adaptations A1-A4 for the insulin pump. All four adaptations affect *AuthConn*, *Conn*, and *send* VCs and the *Adjust Component*. As such, the specifications provided in Table 4 are used to construct the pink token that is assigned to the *Adjust Pink* place. Each pink token, including empty pink tokens, traverses between the place to which it was assigned and its adjacent transition.

Table 4: Pink Token specification for adaptation A1-A4 for insulin pump.

P_{A1}	{(AuthConn, Set to true when connected to an authorized device, 0.55), (Conn, Set to true when connected to an authorized device, 0.55), (send, Send null data,0.55)}
P_{A2}	{(AuthConn, Set to true when connected to a non-authorized device, 0.65), (Conn, Set to true when connected to an authorized device, 0.65), (send, Send null data,0.65)}
P_{A3}	{(AuthConn, Set to true when connected to an authorized device, 0.60), (Conn, Set to false when connected to an authorized device, 0.60), (send, Send null data,0.6)}
P_{A4}	{(AuthConn, Set to true when connected to a non-authorized device, 0.75), (Conn, Set to false when connected to an authorized device, 0.75), (send, Send null data,0.75)}

A single blue token (initially empty) traverses through the VFlow to determine conflicts with VCs at transitions. The blue token collects and carries information related to how VCs are affected. For the VFlow, the blue token B_A for an adaptation A is defined as

$$B_A = (visited, vcConflicts, vcChanges, maxIdx)$$

where,

- *visited* is a set of traversed places within the VFlow,
- *vcConflicts* is a set of identified conflicts that are each of the form $\boldsymbol{conflict} = \left(\boldsymbol{idx_{conflict}}, (\boldsymbol{vc}, \boldsymbol{condition}, \boldsymbol{vcImpact}), \boldsymbol{conflictPlace}, \hat{\boldsymbol{p}}(\boldsymbol{A})\right)$, where
  - $\boldsymbol{idx_{conflict}}$ is the unique index number assigned to the conflict, which is incremented for each identified conflict,
  - *vc* is the affected VC,
  - *condition* is the change condition of *vc*,
  - *vcImpact* is the impact associated with the *vc* and *condition* (the $\boldsymbol{M_{VC}}$ value from Table 1 that corresponds to affected VC and condition),
  - *conflictPlace* is the name of the place in the VFlow where the conflict was detected, and
  - $\hat{\boldsymbol{p}}$ is the change impact value associated adaptation *A* (Table 2),
- *vcChanges* is a set of tuples that are each of the form $\boldsymbol{change} = \left(\boldsymbol{idx_{change}}, \boldsymbol{vc}, \hat{\boldsymbol{p}}(\boldsymbol{A})\right)$, where
  - $\boldsymbol{idx_{change}}$ is the unique index number assigned to the change, which is incremented for each identified change,
  - *vc* is the affected VC, and
  - $\hat{\boldsymbol{p}}$ is the change impact value associated with adaptation *A* (Table 2), and
- *maxIdx* is the largest *idx* among conflicts in *vcConflicts*.

When the VFlow is instantiated, an empty blue token is assigned to the *Start* place. This blue token traverses places (not including *Initialize Pink*, *Adjust Pink*, *Administer Pink*, and *End*) and transitions within the VFlow. The attributes of the blue token are updated according to transition rules, which will be discussed later. Since the blue token that is assigned to the

Start place is initially empty, we use a flag *trigger*, which is initially set to 1, to indicate that the blue token is ready for traversal. Once the blue token begins traversal, *trigger* is set to 0. Traversal terminates when the blue token is empty and *trigger* is set to 0.

The blue token shown in Figure 4 is the state of the blue token after it completes its traversal from the *Start* place to *Administer* and accumulates conflict information in the *Adjust Transition*. There is no conflict information in the *Initialize Transition* or the *Administer Transition,* since both are associated with empty pink tokens. The state of the blue token is provided below.

B_{A1} = *({Initialize, Adjust, Administer},*
{(1,(AuthConn, Set to true when connected to an authorized device, 0.9), Adjust, 0.55),
(2, (Conn, Set to true when connected to an authorized device, 0.9), Adjust, 0.55),
(3, (send, Send null data, 0.9), Adjust, 0.55)},
{(1, AuthConn, 0.55), (2, Conn, 0.55), (3, send, 0.55)}, 3)

The visited set in the blue token holds the traversed place names, e.g., *{Initialize, Adjust, Administer}*. Three conflicts arise in *Adjust* which are associated with the *AuthConn*, *Conn*, and *send* VCs. Each conflict is added to *vcConflicts* in the blue token. Each conflict includes the affected VC, its condition, and the impact multiplier of the VC and condition as determined by Table 1. In addition, each conflict also has a change impact value of 0.55, which is the change impact value associated with adaptation A1. Three changes are added to *vcChanges* in the blue token, which hold the conflict information and are updated over the traversal process. This information includes the affected VCs and the change impact value, which is 0.55. Since three conflicts arise, the *maxIdx* into the blue token is updated to 3.

Red tokens hold the attributes that are needed for the risk assessment and represent alerts regarding the impacts on VCs based on the adaptation qualities. For the VFlow, a red token R_A for an adaptation A is defined as

$$R_A = (\, idx_{token}, vcImpactInfo, \hat{p}(A), conflictPlace, conflictPlacePredominance)$$

where

- $\boldsymbol{idx_{token}}$ is the unique index number assigned to the red token, which is taken from the *maxIdx* of the blue token at the time the red token is generated,
- $\boldsymbol{vcImpactInfo}$ holds the collection of information in the form $\boldsymbol{info} = (\boldsymbol{vc}, \boldsymbol{condition}, \boldsymbol{vcImpact})$, which contains the affected verification concern, the change condition of *vc*, and the impact multiplier of *vc* and *condition* (Table 1),
- $\hat{\boldsymbol{p}}$ is the change impact value associated with adaptation *A* (Table 2)
- $\boldsymbol{conflictPlace}$ is the name of the place in the VFlow where the conflict was detected, and
- $\boldsymbol{conflictPlacePredominance}$ is the place impact multiplier assigned to the affected architectural component (Table 3).

Red tokens are only generated when conflicts are detected and are output by transitions to the *End* place. The red tokens shown in Figure 4 indicate the alerts that are generated for each conflict. They hold the impact multipliers values associated with adaptation A1. The three red tokens are provided below.

$R_{A1,1}$ = *(1, (AuthConn, Set to true when connected to an authorized device, 0.9), 0.55, Adjust, 0.5)*

$R_{A1,2}$ = *(2, (Conn, Set to true when connected to an authorized device, 0.9), 0.55, Adjust, 0.5)*

$R_{A1,3}$ = *(3, (send, Send null data, 0.9), 0.55, Adjust, 0.5)*

The three affected VCs, *AuthConn*, *Conn*, and *send,* along with their conditions and impact values (Table 1) are included within *vcImpactInfo* of the red tokens. Each red token has a change impact value of 0.55, which is the change impact value associated with adaptation A1 (Table 2). All conflicts were identified in the *Adjust* component, so each red token has a *conflictPlace* of Adjust and a *conflictPlacePredominance* of 0.5, which is the value from Table 3 associated with *Adjust*.

To embed verification knowledge, we define a structure for the transitions in the VFlow. A transition T is defined as

$$T = (placeName, placePredominance, vcInfo),$$

where

- $placeName$ is the associated architectural component name,
- $\boldsymbol{placePredominance}$ is the place impact multiplier assigned to the associated architectural component (Table 3), and
- $\boldsymbol{vcInfo}$ is a set of tuples of the form $\boldsymbol{info} = (\boldsymbol{vc}, \boldsymbol{condition}, \boldsymbol{vcImpact})$, where vc is the verification concern, $condition$ is potential change, and $\boldsymbol{vcImpact}$ is the impact multiplier that is associated with the vc and $condition$ (Table 1).

The transitions shown in Figure 4 are for the *Initialize*, *Adjust* and *Administer* components, which hold the verification concerns, their change conditions and impact multipliers for associated components. The *Initialize Transition* has a *placeName* of Initialize and a *placePredominance* of 0.5 (Table 3). *vcInfo* is empty. The VCs extracted for INS.2, which are *AuthConn*, *Conn*, and *send*, are not associated with S1 (Algorithm 3). The *Adjust Transition* has a placename of Adjust and a *placePredominance* of 0.5 (Table 3). It has a *vcInfo* that includes VCs *AuthConn*, *Conn*, and *send* and their associated conditions with the corresponding impact multiplier values as shown in Table 1. The *Administer Transition* has a *placeName* of Administer and a *placePredominance* of 0.5 (Table 3). It has a *vcInfo* that is empty as the extracted VCs (*AuthConn*, *Conn*, and *send*) are not associated with S5.

Transition Conditions:

T.placeName is NOT in B.visited
VC in T.vcInfo appear in a tuple in B.vcChanges
A VC in a tuple in P conflicts with a VC in T.vcInfo

Transition Actions:

FORALL VC in T.vcInfo that appear in a tuple in P
Increase B'.maxIdx for the unique ID of red token
Create a red token, R, with proper idx_token and other information held by P and T
Update B'.vcConflicts by increasing idx_conflict and the appropriate information held by T for all matching VCs
Update B'.vcChanges by increasing idx_change and the appropriate change information from P
Add T.placeName to B'.visited
Send B' to next place
Send P to its place
Send all red tokens to End place

Figure 5: Sample transition rule for CPN.

A transition becomes enabled when blue and pink tokens are received from their respective input places. We define transition rules to process the tokens and output alerts. In Figure 5, one sample transition rule is shown. This rule applies when the blue token has not visited the place (check B.visited), does not have a conflict with a VC from a processed pink token with matching transition VCs (compares VCs into B.vcConflicts and T.vcInfo), but does have a conflict with a pink token VC and transition VCs (compares VCs from P.vc and T,vcInfo). After executing the transition rule, B.visited includes the place name and updated B.vcConflicts and B.vcChanges using the pink token and the transition's embedded verification knowledge. We have this transition rule and the other 8 transition rules that are used for the wearable simulations in Appendix A.

To assess the risk of each adaptation, we reuse a utility function from prior work [34] that calculates the expected utility of an adaptation from the risk factors accumulated by red tokens. The utility function is shown below, where R is the set of requirements and $w(r)$ is the utility weight needed to maintain system compliance with a requirement, r.

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

In Table 5, stakeholder-supplied utility weights of the requirements for the insulin pump are provided. $P(S(t))$ is the probability of risk associated with each alert (red) token, t, and estimated by multiplying the three impact multipliers together.

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

The adaptation with the greatest expected utility is considered to be the least risky and most appropriate for reusing the original proof of the verification process.

Table 5: Utility weights of the requirement.

Insulin Pump		
INS. 1	**INS. 2**	**INS. 3**
0.75	0.75	0.5

We calculate the expected utility for each of the adaptations for all three wearables. The expected utility for adaptations A1-A4 with respect to the insulin pump are shown below.

$$E[U(A1)] =$$
$$0 + (0.75*0.55*0.9*0.5)\ (0.75*0.55*0.9*0.5)\ (0.75*0.55*0.9*0.5) + 0 = 0.00639601391$$
$$E[U(A2)] =$$
$$0 + (0.75*0.65*0.2*0.5)\ (0.75*0.65*0.2*0.5)\ (0.75*0.65*0.9*0.5) + 0 = 0.00052135839$$
$$E[U(A3)] =$$
$$0 + (0.75*0.6*0.5*0.5)\ (0.75*0.6*0.5*0.5)\ (0.75*0.6*0.9*0.5) + 0 = 0.00256289062$$
$$E[U(A4)] =$$
$$0 + (0.75*0.75*0.5*0.5)\ (0.75*0.75*0.2*0.5)\ (0.75*0.75*0.9*0.5) + 0 = 0.0020022583$$

According to the risk assessment perspective, adaptation A1, which allows the device to remain connected without fostering while sending empty packets has the best utility for the insulin pump. Allowing fostering may potentially cause devastating results for requirement INS.2, which says that the insulin pump is only allowed to connect with authorized base stations. Thus, adaptation A2, A3 and A4 have lower risk assessment values. For the insulin pump, the adaptations would be ranked in order as A1, A3, A4, A2.

Similarly, we calculate the expected utility for each adaptation across the hearables and the HVRM, which are shown in Table 6. From the risk assessment, adaptation A3 is the best for hearables and adaptation A1 is the best for the HVRM.

Table 6: Expected utility calculation from risk assessment.

	A1	A2	A3	A4
Hearables	0.00843557904	0.00466998096	0.01160804925	0.00845593725
Heart Rate Variability Monitor	0.00806354516	0.00181526464	0.00300389062	0.0026913208
Insulin Pump	0.00639601391	0.00052135839	0.00256289062	0.0020022583

For the hearables, adaptation A3 is the best choice as the device gets disconnected and sends null data, which has an unconcerned impact on HR.1. It also does not allow fostering, so there is no unauthorized buffer change, which has an unconcerned impact on HR.2. Adaptation A4 is the second-best choice as the device gets disconnected, which has an unconcerned impact on HR.1, but it allows fostering. Fostering sometimes allows unauthorized connections, but as A4 disconnects hearables from available connections, fostering only has a worrisome impact. Adaptation A1 allows the device to remain connected while sending null data, which prevents music from being streamed and has a devastating impact on requirement HR.1. Similarly, A2 also has a devastating impact on HR.1, and, due to allowing fostering, has devastating impact on HR.2. The adaptations for the hearables can be ranked in order as A3, A4, A1, A2.

The chosen adaptation for the HVRM is Adaptation A1, which allows the device to remain connected with an authorized connection while sending empty packets. Both changes have an unconcerned impact since they do not affect HRVM.1 or HRVM.2. The adaptation A2, which allows fostering, is the second-best choice because fostering has the potential risk of

not being authorized. Adaptations A3 and A4 cause devastating impacts for requirement HRVM.2 as the device is disconnected and eventually data is lost. The adaptations for the HRVM can be ranked in order as A1, A2, A4, A3.

6. Evaluation

6.1. Repeatability of the formal approach using Lean

To evaluate the repeatability of our proof process in Section 4.3 and the heuristic rules in Section 4.4, we examine the methodology used within the Lean automated theorem prover [61]. Lean is an open-source, interactive theorem prover in active development at Microsoft Research [62]. While Lean does not contain built-in support for Hoare logic or temporal logic, these logics can be added to the theorem prover through custom packages. In the subsequent paragraphs, we highlight how the methodology used with KIV translates to Lean resulting in a proof-of-concept [63] that demonstrates the potential for our proof process.

Include the semantics of Hoare logic in Lean. Along with the semantics, definitions, lemmas, and sequents must also be included. Baanen et al. demonstrate how Hoare logic can be implemented in Lean [63, 64]. Their libraries, that include definitions and lemmas, can be imported into Lean. Their lemmas are definitionally equal to the rules that KIV uses to perform proof steps as both use the same underlying sequents. As long as the same sequents, with similar definitions and lemmas, are implemented in Lean, an equivalent proof process can be constructed.

Include an abstraction of the semantics of the target program. Lean currently has no abstract representation of programming language semantics. Efforts are underway to provide such support in the next version of Lean [65]. We implement an abstraction to represent the semantics necessary to define Algorithms 1, 2, and 3 (see our GitHub reference [66]). Specifications from Algorithm 3 are written using this abstraction and then symbolically translated to a Hoare triple. The initial conditions of the program are specified as the pre-condition of the triple while the post-condition represents the desired requirement being proven.

Include a definition of a lexical scope, or program state, including lemmas. To determine the pre- and post-condition of any Hoare triple defined over a program's semantics, it is necessary to implement the definition of a lexical scope. Lemmas must also be defined to update the current scope and apply it to a specific program variable within the proof process. Propositions over program variables are defined as a conjunction of terms where each term includes a sequence of scope updates and one scope application. A scope update is denoted as $s\{"I" \mapsto s\ "I" + 1\}$, which states that the value of variable I in the current scope s is to be updated to the increment of its previous value. A scope application is denoted $s\_2\ "I"$, where $s\_2 = s\{"I" \mapsto s\ "I" + 1\}$, and it is equivalent to the actual value of I. Lean's built-in command *simp* can be used to simplify each term in the proposition to its actual value. Examples can be found in Section 9.4 of *the Hitchhiker's Guide to Logical Verification* [63] of a scope (called state) defined for program variables with the natural number type. We found it necessary to implement our own scope definitions and lemmas to support all program data types [66].

Define new lemmas for the sequents for KIV's *if positive*, *if negative*, and *call left* rules. Since there is no equivalent of KIV's *if positive* or *if negative rule* in Lean or its libraries, we define new lemmas *ite_true_intro* and *ite_false_intro* [66]. When these lemmas are applied, they impose a goal that represents the expected condition of the if statement.

Rule 4 can be applied to this goal to extract verification conditions. We also define a new lemma *call_intro* in Lean [66] to represent KIV's *call left* rule. Applying the *call_intro* lemma in Lean imposes a goal to show that the arguments for V_{in} and V_{inout} exist in the current program state. Rule 5 can be applied to this goal to extract verification conditions.

Include support for temporal logic, along with definitions and lemmas. Based on previous work [67] and our efforts [66], we are confident that temporal logic can be implemented in Lean. Hoare semantics can be defined using an operator that sequentially applies instructions, as has been demonstrated [63, 66]. Sequentially executed instructions can be assigned a step number that is incremented with each instruction. By doing so, temporal logic and Hoare logic can be combined in Lean by structuring each temporal proposition as an assertion about an instruction that is executed with a specific step number. Alternatively, temporal properties can be proven by inserting the property into the post-condition of a Hoare triple and verifying that Hoare triple across all applicable execution traces.

Once temporal logic has been implemented in Lean, TCPs and STCPs can be defined in Lean to condense the proof steps needed to walk through each program specification. A program can be verified using a backward proof or a proof by contradiction. A backward proof is simpler than a proof by contradiction but is limited to deterministic programs. A proof by contradiction uses the natural number M (Section 4.2). Our lemmas, *lemma-invariant*, *lemma-progress*, and *lemma-progress-split* can be defined in Lean and applied to close resulting goals in a proof by contradiction. Having such support would be extremely valuable when verifying requirements of non-terminating or non-deterministic programs in Lean.

6.2. Validation of the adaptation assessment framework

One of the important research aspects to incorporate self-awareness within a SISSY system is to consider the impact of integration changes that will influence the behavior of the entire system [68]. This paper proposes adaptation assessment framework [21] to incorporate verification awareness within the system, which models the verification workflow by extracting meta-data from proof process for each critical requirement. Given that the nature of the requirement requires formal verification (e.g. automated theorem proving), we analyze the proof process to construct well-defined, heuristic rules to extract meta-data from the proof process workflow across architectural components to construct the model. Using the model, our approach identifies conflicts between the adaptation's changes and the requirement's original verification process.

To validate the approach's effectiveness, we check the alignment of the adaptive system behavior with the risk assessment introduced in Section 5, we simulate adaptations A1-A4 for all three wearables. We embed checkpoints within the wearables' source code and log the execution trace for simulated adaptations. We collect the log of checkpoint execution traces and accumulate the score of checkpoints that passed. Adaptations are ranked based on their cumulative scores. The adaptation with the highest score, i.e., most checkpoints passed, is considered to be the best choice. Figure 6 shows how the checkpoints are placed within the insulin pump's code to determine if (i) buffer does not overflow (CP1), (ii) data is generated (CP2), (iii) data is sent (CP3), (iv) connection is authorized (CP5), (v) connection has been established (CP5), (vi) data is read (CP6), (vii) data has been stored (CP7), (viii) buffer change has been authorized (CP8), and (ix) insulin is administered when blood sugar level is more than the defined value (CP9).

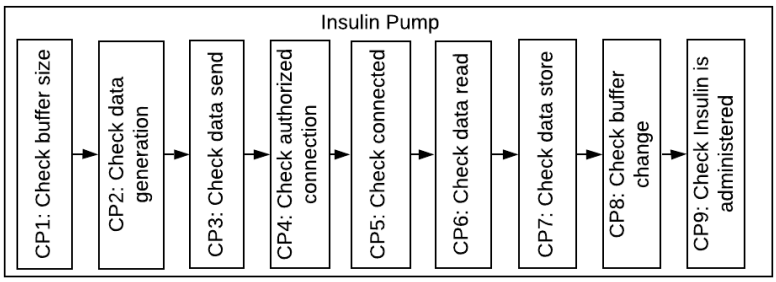

Figure 6: Checkpoints embedded into insulin pump code for validation of adaptation risk assessment.

We execute each simulation for the insulin pump 100 times. The results are shown in Table 7. With respect to the insulin pump, adaptations A1 and A3 have an impact of unconcerned as these adaptations do not allow fostering. CP4, which checks if the connection is authorized, passes all 100 runs. Adaptations A2 and A4 allow fostering and fostering sometimes allows unauthorized connections by forcefully setting the *auth_connection* flag to *true*. This results in the reduced scores of 51 and 52 for A2 and A4, respectively. Buffer change is allowed when *auth_connection* flag is *true* and connected to an authorized device. Adaptation A2 allows fostering and staying connected, which may cause a failure at checkpoint CP8. Thus, the checkpoint for authorized buffer change fails for adaptation A2. From the checkpoint log, we find that authorized buffer changes only occur in 51 trials. Checkpoint CP3 checks that data that has been sent is never passed and the score for this checkpoint is 0 for all adaptations, since all adaptations allow the sending of null data. In a similar manner, we accumulate the score for all checkpoints. The scores reflect the risk assessment results calculated by each wearable following the adaptation assessment framework described in Section 5.

Table 7: Checkpoint simulation result for the insulin pump.

	A1	**A2**	**A3**	**A4**
CP1	100	100	100	100
CP2	100	100	100	100
CP3	0	0	0	0
CP4	100	51	100	52
CP5	100	49	0	0
CP6	0	0	0	0
CP7	0	0	0	0
CP8	100	51	100	100
CP9	100	100	100	100
Score & Ranking	600 (rank 1)	451 (rank 4)	500 (rank 2)	452 (rank 3)

For the HRVM, checkpoints CP1 to CP7 from Figure 6 are included within its code. We run the adaptive system simulations for HRVM for 100 trials. From the simulation results shown in Table 8, the best adaptation remains adaptation A1, which allows the HRVM to connect with an authorized connection and send empty packets. The checkpoint for authorized connection, CP5, is passed in all 100 trials. Since empty packets are sent, CP3 for send data and CP6 for data is read always fail with score of 0. Adaptation A2 is the second-best choice, which allows fostering but fostering raises the risk of the connection not being authorized. This is gauged by CP5. The score for CP5 is less for adaptation A2 than it is for

A1. Adaptations A3 and A4 cause devastating impacts for the HRVM as they allow it to be disconnected and eventually the data gets lost as determined by the log of checkpoint CP7, which has score 0 for both A3 and A4.

Table 8: Checkpoint simulation result for HRVM.

	A1	**A2**	**A3**	**A4**
CP1	100	100	100	100
CP2	100	100	100	100
CP3	0	0	0	0
CP4	100	47	100	53
CP5	100	47	0	0
CP6	0	0	0	0
CP7	100	100	0	0
Score & ranking	500 (rank 1)	384 (rank 2)	300 (rank 4)	253 (rank 3)

Referring back to Figure 6, the hearables also have checkpoints CP1-CP7. However, checkpoint CP8 is changed to check (viii) if music is streamed for any connection, and CP9 is changed to check (ix) buffer has changed for authorized connection. We also add CP10, which is (x) accelerometer data that is collected is stored. We run the adaptive system simulations for the hearables 100 times and the results are shown in Table 9. CP3 fails for each of the adaptations due to sending null data, which results in a score of 0. CP4 is reduced for adaptations A2 and A4 because fostering is allowed. CP5 fails each run for adaptations A3 and A4 due to the device being disconnected and is reduced for adaptation A2 due to the device permitting fostering. CP6 and CP7 fail every run, regardless of adaptation, since each adaptation uses null data. Due to sending null data music can't be streamed, so CP8 fails each run for adaptions A1 and A2, where the device is connected. Adaptations A3 and A4 disconnect the device, which results in CP5 failing every run. However, A3 remains the best choice for the hearable, again aligning with the results of our adaptation assessment framework.

Table 9: Checkpoint simulation result for hearables.

	A1	**A2**	**A3**	**A4**
CP1	100	100	100	100
CP2	100	100	100	100
CP3	0	0	0	0
CP4	100	51	100	51
CP5	100	51	0	0
CP6	0	0	0	0
CP7	0	0	0	0
CP8	0	0	100	100
CP9	100	51	100	100
CP10	0	0	100	100
Score & ranking	500 (rank 3)	353 (rank 4)	600 (rank 1)	551 (rank 2)

6.3. Comparison of the adaptation assessment framework to Rainbow

To provide perspective on our approach, we refer to the taxonomy proposed by Krupitzer et al. [69] that includes a set of qualitative dimensions to compare approaches that design and

implement SASs. We reuse 9 of their dimensions, their questions used to capture qualitative information, and their input on Rainbow in Table 10, where we (1) expand on their qualities related to Rainbow [40, 41, 42, 71, 72] for particular dimensions, (2) add questions to capture qualitative information, and (3) add the qualities of our approach for each dimension.

Table 10: Reuse of the dimensions in Krupitzer, et al. [69].

Dimensions from [69]	**Captured Information** (*italics* – [69], regular font – new)	**Rainbow** (*italics* – [69], regular font – new)	**Adaptation Assessment Framework** (*italics* – [69], regular font – new)
Type of support	*What kind of support does it provide? What elements does the approach include?*	*Framework, Tools,* Methodologies	Framework, Tools, Methodologies
Reusability	*Is reusability considered? How is it achieved?*	*Reusable adaptation infrastructure consisting of system, Architecture and translation layers*	Reusable adaptation infrastructure consisting of architecture, proof-process meta-data, and risk assessment mechanisms
Use of tools	*How do the tools support the development? When are they applied?*	*Stitch script editor, Rainbow development toolkit,* Acme architectural design toolset [40, 71], SWIM to simulate target web applications [70] Applied throughout design and implementation [40, 71]]	KIV theorem prover for proof meta-data extraction, Colored Petri Net construct for adaptation impact assessment, utility function for adaptation risk assessment Applied throughout design and implementation
Support of adaptation mechanisms	*How does the approach handle the system's adaptation? What mechanisms does it utilize?* How is the adaptation plan configured?	*Not specified* Tailors adaptation strategies and tactics at design-time to support runtime adaptation mechanisms [40, 71] Chooses adaptation plans from pre-defined strategies based on pre-defined utility preferences [40, 71]	Heuristic rules extract theorem prover meta-data as verification concerns Risk assessment supports runtime choice of adaptation Chooses adaptation plan from pre-defined strategies and externally configured plans
Type of adaptation	*What is the granularity of the adaptation*	*Compositional*	Compositional and parameter expressed within change set
Special demands on developer	*What requirements does the developer have to fulfill? What type of and how much knowledge is demanded in order to use the approach?*	*Mathematical knowledge* Application of probabilities to define strategies and utility preferences [41, 42, 72]	Mathematical knowledge Formal methods knowledge to perform proof process, Application of probabilities for risk assessment
Development phase	*In which step(s) of the software development process can it be applied?*	*Implementation*	Design and implementation
Applicability	*Which systems can the approach be applied on?* Does the approach support legacy and/or blackbox systems?	*Self-Adaptive Systems* Yes, from an integration perspective [40, 71]	Self-Adaptive Systems Yes, when source code is available
Language specificity	*Does the approach require a specific programming or modeling language?*	*Java, XML*	Language independent, except for the theorem prover language requirements

The comparison presented by Krupitzer, et al. [69] pinpoints Rainbow's approach along with other existing adaptation frameworks but does not provide further elaboration in each dimension. From literature on Rainbow [40, 41, 42, 71, 72], we include additional insights in Table 10 to provide a clearer understanding of how Rainbow compares with our adaptation

assessment framework. Specifically, the *Use of tools* dimension denotes specific tools the approach uses for development. Rainbow mentions tools that are associated with specific tasks and when they are applied to support development. The *Support of adaptation mechanisms* dimension highlights significant distinctions across both approaches between the elements/processes that are required to implement adaptation mechanisms and how they support adaptation. Rainbow requires that system specific adaptations be defined at design-time in the form of strategies (adaptation conditions) and tactics (adaptation changes) that comprise decision trees [71, 72]. Rainbow also requires pre-defined utility preferences that map tactics to expected utility values [40, 71]. Pre-defined strategies and tactics are used to select an adaptation plan at runtime based on pre-defined utility preferences [40, 71]. In our own approach, we extract meta-data from the proof process at design-time and use the extracted meta-data within a model that includes the system architecture. The model provides inputs to our risk assessment procedure. Adaptation plans are then assessed and selected at runtime based on the risk assessment results. In the *Special demands on developer* dimension, we elaborate over the expertise that each approach requires. The *Applicability* dimension is further extended to include whether the approach can support legacy and/or blackbox systems. While these are not system domains per se, lack of support for legacy and blackbox systems can limit applicability within a system domain.

We extend the dimensions in Table 11 to include (i) **Intensity of human effort** to identify elements of the approach that require human implementation or interaction at one time or consistently throughout the development process, (ii) **Implementation of adaptation layer** to describe how the adaptation layer is incorporated into the system, and (iii) **Support of model/requirement modifications at runtime** to state how the adaptation layer can be updated after initial deployment. Information is captured using questions from the second column that we crafted. These dimensions have been included to further highlight points of comparison between Rainbow's approach and our own. Captured information for Rainbow is taken from prior literature [40, 41, 42, 71, 72]. The **Intensity of human effort** details human-oriented processes that, in turn, highlights areas of the approach that must be repeated when the system architecture or its requirements change. The **Implementation of Adaptation Layer** dimension provides insight into limitations of the adaptation layer. The performance of external adaptation layers, as used in Rainbow, can suffer from poor network conditions. Using internal adaptation layers, such as in our own approach, typically have stricter limitations with respect to applicability. Lastly, the **Support of model/requirement modifications at runtime** dimension provides insight as to what should be updated and/or redeployed if the system architecture and/or requirements change. Having the ability to update a system's decision making with respect to adaptations while it is deployed is desirable for many SASs.

Table 11: Extending the dimensions in Krupitzer, et al. [69].

Extended Dimensions	Captured Information	Rainbow	Adaptation Assessment Framework
Intensity of Human Effort	Which elements require human implementation and/or interaction?	Defining adaptation strategies & tactics, utility preferences, and the configuration of the adaptation infrastructure [40, 41, 42, 71, 72]	Performing requirement proof process, meta-data extraction, and defining and classifying impact multipliers
Implementation of Adaptation Layer	How is the adaptation layer incorporated into the system?	External adaptation layer that is incorporated over a network connection [40, 4, 712]	Internal adaptation layer that is embedded within the system
Support of model/requirement	How to update adaptation layer if	not specified	Update architectural model and impact multipliers to

modifications at runtime	system requirement and/or model need to be changed after initial deployment?		redeploy

7. Discussion and conclusion

This paper describes and evaluates an adaptation assessment framework to embed verification awareness on a wearable testbed that comparatively evaluates the risk of potential adaptations inhibiting the proof processes of previously verified requirements. We investigate our approach on three simulated wearables that have different verified requirements. We outline a strategy to use the KIV automated theorem prover to verify the wearable requirements. Heuristic rules are defined to capture the meta-data from the KIV proof process and embed verification awareness within the target wearables. We show that given the same four adaptations available to each wearable, the wearable independently determines the adaptation least risky to inhibiting the proof processes of its verified requirements. Incorporating verification awareness within the system is an effective approach to reason about system compliance with requirements when runtime adaptation is needed, including when adaptations are configured at runtime.

One potential shortcoming of this approach is that system requirements have to be initially specified by hand. This activity needs expert knowledge and is open to problems of interpretation as human experts translate statements of natural language to formal specifications and vice versa [60]. The challenge is that system requirements cannot be procedurally generated with a high degree of reliability. Attempting to have a generalized machine procedurally generate requirements only translates the problem of interpretation from a human to a machine that is far less equipped to navigate the problem domain [60]. A domain-specific model could be developed that would have the necessary expertise to generate system requirements. However, developing the domain-specific model would still require expert knowledge to procedurally identify and extract requirements [73]. One direction is to use structured natural language for a domain of requirements, such as our approach with the use of the NIST SP800-53 security controls [74] that can be more easily expressed as a formal specification [20].

Rice's theorem states that *any non-trivial, semantic property of a program is undecidable* [24]. Given a property P of a program, P is trivial if-and-only-if there are only programs that satisfy P or there only programs that do not. For example, if P is the invariant $\square(X \rightarrow Y)$, then we could demonstrate that P is non-trivial by showing that there is at least one program that satisfies P and at least one program that does not. Once it has been determined that P is non-trivial, we could attempt to construct an algorithm that accepts all programs and adaptations where P is satisfied, or rejects them otherwise. This, in essence, defines a decision problem where functional requirements are treated as non-trivial properties. As such, by virtue of Rice's theorem, we can determine that the decision problem to accept a program or adaptation due its ability to satisfy a functional requirement, such as P, is undecidable. It follows from that conclusion that determining whether a given program or adaptation satisfies P is undecidable. Thus, we employ a heuristic approach to assess the risk of an adaptation potentially violating a requirement by examining the meta-data captured from a proof process, strategy, or structure.

The KIV automated theorem prover does place certain limitations on our work. Each set of specifications for each program and all related requirements must be constructed by hand. In addition, each proof of each requirement must also be performed manually. KIV does employ a very helpful set of heuristics that can automate the vast majority of the proof steps. Lastly, capturing meta-data from the proof process currently requires expert intervention. However,

capturing meta-data is a straightforward endeavor since the programs must already be reduced to the bare minimum number of program variables and program states to support proof construction. Thus, reliance on a formal proof process may limit the risk assessment approach from being widely accepted until the automated technology becomes more powerful. Current efforts to implement new, more expressive theorem provers, such as Lean [61], provide us with confidence that greater and more customizable automation will be available in the future. That said, the proof process does ensure the system's correctness with respect to certain requirements and the proof process only needs to be performed once to capture meta-data information. The presented risk assessment approach expects a structured format to specify adaptation changes that we currently construct manually. We assume that an adaptation planner would have the capability to specify the change set.

State space explosion remains a significant research challenge for the self-adaptive system community [27]. In general, modeling the runtime behavior of a system using a theorem prover, such as KIV, is subject to limitations posed by state space explosion [6]. Any single requirement can be specified over any number of program variables, resulting in a state space that is exponential with respect to the number of program variables. In such cases, axioms of prepositional logic must be used to subdivide complex requirements into many simpler requirements that can each be individually proven. There is, however, no efficient means by which to reduce a requirement specified in disjunctive normal form, if it is assumed that all requirements should be satisfied.

Developers can employ compositional analysis techniques, such as refactoring and decomposition, to address state space concerns for their target system. Both KIV and Lean support custom lemmas which can be applied to the proof process to simplify the proof process, where possible. For complex systems that cannot be easily refactored, the approach presented relies on a model of system behavior as defined by our adaptation assessment methodology. In practice, such models can be more easily refactored and verified than original source code [12].

Such techniques continue to demonstrate merit in industry applications. Microsoft has been employing proofs using F*, a proof assistant programming language, and other modern proof assistants, such as Lean [61], to improve the security of HTTPS. Project Everest covers 600,000 lines of code and proofs that are integrated hundreds of times per day [75]. Ancillary to Project Everest, Microsoft released EverCrypt, a cryptographic library that has fully verified C and assembly code using F* [76]. Zelkova [77], another proof assistant, has been used by Amazon to encode the semantics of AWS service policies into satisfiability modulo theories (SMT). Policy properties are then verified using SMT solvers. With a significant understanding of their domain, researchers at both of these companies have been able to employ theorem provers to provide hundreds to millions of proofs every day [75, 77].

The most significant issues with scalability are those related to constructing an appropriate model of the system architecture that can be used for adaptation assessment. Large scale systems often have many interdependencies between components that would each have to be evaluated to gauge the full impact of an adaptation [4]. This evaluation can create timing issues that undermine the integrity of the system's assessment process. These timing issues are further exacerbated if the system must perform runtime adaptation assessments within a restrained window of time. The work presented in this paper addresses these issues by extracting proof process meta-data to institute heuristics that can be employed by runtime adaptation assessments. Employing such heuristics in a self-aware system can avoid timing issues that would otherwise limit scalability.

With respect to IoT and cloud computing, relying on an external adaptation layer to perform adaptations can introduce additional latency due to network lag time that can be

exacerbated by poor network conditions. IoT devices usually have low computational power and may operate in low bandwidth situations [52, 54]. External adaptation layers have been examined with IoT devices and cloud services to address their need to modify their requirements during deployment [78], but response time can be affected. Once meta-data has been extracted, our approach employs a lightweight and embedded risk assessment process. Thus, embedded verification awareness allows risk assessment to be performed on an adaptation without added latency introduced by a connected, external adaptation layer. In addition, with our approach, we can perform a proof process on modified requirements, develop a modified model as an external action, and then redeploy the modified model within a running system as an embedded adaptation layer. Thus, the approach may benefit SAS in IoT and cloud computing domains.

To cope with uncertain situations, cyber-physical systems (CPS) should maintain evidence that the system maintains a certain degree of requirement compliance confidence [3, 79]. Architecting trustworthiness within a CPS involves development-time modeling, verification, and synthesis of assurance evidence [80], which are all supported by our adaptation assessment framework. Our risk assessment estimates the probability of a potential requirement violation due to adaptation, which can be used as an evidence of trustworthiness.

Some of the most significant difficulties in scaling our approach exists where human intervention is currently required. After proofs have been constructed, experts must evaluate the proofs and apply the extraction rules to extract verification concerns from the proofs. These verification concerns are then associated with impact values, which are currently determined by human expertise. There is potential for automation by simulation or a design-time adaptation risk assessment. Impact values can then be used to evaluate adaptations once the adaptation has been translated to a *changeset* by an expert. Automating the translation requires a specification language for the adaptation that aligns with *changeset* expectations. The risk assessment process employs a CPN that also must be constructed by hand. It is our intent to further investigate these areas so that the need for human intervention will be reduced.

Acknowledgements

This material is based on research sponsored by Air Force Research Laboratory under agreement number FA8750-16-1-0248. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory or the U.S. Government.

References

[1] K. Bellman et al., Self-Improving System Integration – Status and Challenges After Five Years of SISSY, in Proc. 3rd IEEE Intl. Workshops on Foundations and Applications of Self* Systems (FAS*W), pp. 160–167, (2018).

[2] C. Gruhl, S. Tomforde, and B. Sick, Aspects of Measuring and Evaluating the Integration Status of a (Sub-)System at Runtime, in Proc. 3rd IEEE Intl. Workshops on Foundations and Applications of Self* Systems (FAS*W), pp. 198–203, (2018).

[3] S. Jahan, M. Pasco, R.F. Gamble, P. McKinley, and B.H.C. Cheng, MAPE-SAC: A Framework to Dynamically Manage Security Assurance Cases, 1st International Workshop on Self-Protecting Systems (SPS 2019), Umea, Sweden, pp. 146-151, (2019).

[4] A. Diaconescu, L. J. Di Felice, and P. Mellodge, Multi-Scale Feedbacks for Large-Scale Coordination in Self-Systems, IEEE 13th International Conference on Self-Adaptive and Self-Organizing Systems (SASO), pp. 137-142, (2019).

[5] P. R. Lewis et al., A survey of self-awareness and its application in computing systems, in Proc. Int. Conf. on Self-Adaptive

and Self-Organizing Systems Workshops (SASOW), pp. 102–107, (2011).

[6] A. Goodloe, Challenges in high-assurance runtime verification, In T. Margaria and B. Steffen, editors, 7th Int'l Symp. on Leveraging Applications of Formal Methods, Verification and Validation, ISoLA 2016, LNCS. Springer, (2016).

[7] G. Blair, N. Bencomo, and R. B. France, Models@ run.time, Computer, vol. 42, no. 10, pp. 22-27, (2009).

[8] E. Clarke, D. Long, and K. McMillan, Compositional model checking, In Proceedings of the Fourth Annual Symposium on Logic in computer science, IEEE Press, 353–362, (1989).

[9] C. Y. Cho, V. D'Silva, and D. Song, BLITZ: Compositional bounded model checking for real-world programs, 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), Silicon Valley, CA, pp. 136-146, (2013).

[10] W. Yeh and M. Young, Compositional reachability analysis using process algebra, In Proceedings of the symposium on Testing, analysis, and verification (TAV4), Association for Computing Machinery, New York, NY, USA, 49–59, (1991).

[11] S. Cheung and J. Kramer, Context constraints for compositional reachability analysis, ACM Trans. Softw. Eng. Methodol. 5, 4 Oct., 334–377, (1996).

[12] Y. Cheng, M. Young, C. Huang, and C. Pan, Towards scalable compositional analysis by refactoring design models, SIGSOFT Softw. Eng. Notes 28, 5 September, 247–256, (2003).

[13] Y. Cheng, Y. Cheng, and H. Wang, ARCATS: a scalable compositional analysis tool suite, In Proceedings of the 2006 ACM symposium on Applied computing (SAC '06), Association for Computing Machinery, New York, NY, USA, 1852–1853, (2006).

[14] J. C. Corbett and G. S. Avrunin, Towards scalable compositional analysis, SIGSOFT Softw. Eng. Notes 19, 5 Dec., 53–61, (1994).

[15] K. Bellman, Reflective Systems Are a Good Step Towards Aware Systems, Computer Science, (2014).

[16] K. Bellman et al., Self-modeling and Self-awareness, In: Kounev S., Kephart J., Milenkoski A., Zhu X. (eds) Self-Aware Computing Systems, Springer, Cham, (2017).

[17] P. R. Lewis, M. Platzner, B. Rinner, J. Tørresen, and X. Yao, Eds., Self-aware Computing Systems: An Engineering Approach, Sprin International Publishing, (2016).

[18] G.J. Holzmann, New challenges in model checking, In: Symposium on 25 years of Model Checking, Seattle, USA, LNCS, vol. 4925. Springer, Heidelberg, (2006).

[19] E. A. Emerson, The beginning of model checking: A personal perspective, In 25 Years of Model Checking, volume 5000 of LNCS, pages 27–45, Springer Berlin / Heidelberg, (2008).

[20] A. Marshall, S. Jahan, and R. Gamble, Toward Evaluating the Impact of Self-adaptation on Security Control Certification, 13th Int'l Conf. on Soft. Eng. for Adaptive and Self-Managing Systems, (2018).

[21] S. Jahan, I. Riley, C. Walter, and R. Gamble, Extending Context Awareness by Anticipating Uncertainty with Enki and Darjeeling, 4th Workshop on Self-Aware Computing, (2020).

[22] 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).

[23] G. Ernst et al., KIV: Overview and VerifyThis competition, Int'l J. on Software Tools for Technology Transfer. 17:6, pp. 677–694, (2015).

[24] K. Jensen and L.M. Kristensen, Coloured Petri Nets: Modelling and Validation of Concurrent Systems, Springer-Verlag, (2009).

[25] H. G. Rice, Classes of Recursively Enumerable Sets and Their Decision Problems, Transactions of the American Mathematical Society, Vol. 74, No. 2, pp. 358-366, (1953).

[26] C. Walter, I. Riley, and R. Gamble, Securing Wearables through the Creation of a Personal Fog, In Proceedings of the 51st Hawaii International Conference on System Sciences, (2018).

[27] B.H.C. Cheng, et al. Software Engineering for Self-Adaptive Systems: A Research Roadmap. In: Cheng, B.H.C., de Lemos, R.,

Giese, H., Inverardi, P., Magee, J. (eds) Software Engineering for Self-Adaptive Systems. Lecture Notes in Computer Science, vol 5525. Springer, Berlin, Heidelberg, (2009).

[28] R. de Lemos, D. Garlan, and H. Giese, Software Engineering for Self-Adaptive Systems: Assurances, Dagstuhl Seminar 13511, (2013).

[29] J. Cámara, R. de Lemos, C. Ghezzi, and A. Lopes, Assurances for Self-Adaptive Systems: Principles, Models, and Techniques, Springer Publishing Company, Incorporated, (2013).

[30] F. Faniyi et al., Architecting Self-Aware Software Systems, Proc. Working IEEE/IFIP Conf. Software Architecture (WICSA 14), pp. 91–94, (2014).

[31] F. Zambonelli, N. Bicocchi, G. Cabri, L. Leonardi, and M. Puviani, On self-adaptation, self-expression, and self-awareness in autonomic service component ensembles, in Self-Adaptive and Self-Organizing Systems Workshops (SASOW), 2011 Fifth IEEE Conference on. IEEE, pp. 108–113, (2011).

[32] E. Bartocci, Y. Falcone, A. Francalanza, and G. Reger, Introduction to runtime verification, in Lectures on Runtime Verification—Introductory and Advanced Topics, in Lecture Notes in Computer Science, vol. 10457, Springer, pp. 1–33, (2018).

[33] C. Sánchez et al., A survey of challenges for runtime verification from advanced application domains (beyond software), Form Methods Syst Des 54, pp. 279–335, (2019).

[34] G. Tamura et al., Towards practical runtime verification and validation of selfadaptive software systems, In: de Lemos, R., Giese, H., M¨ uller, H.A., Shaw, M. (eds.) Software Engineering for Self-adaptive Systems II. LNCS, vol. 7475, pp. 108–132. Springer, Heidelberg, (2013).

[35] F. Mogavero, A. Murano, G. Perelli, and M. Y. Vardi, Reasoning about strategies: On the model-checking problem, TOCL, vol. 15, no. 4, pp. 34:1–34:47, (2014).

[36] A. Filieri, G. Tamburrelli, and C. Ghezzi, Supporting self-adaptation via quantitative verification and sensitivity analysis at run time, IEEE Trans. Softw. Eng., vol. 42, no. 1, pp. 75-99, (2016).

[37] S. Tomforde et al., Know Thyself – Computational Self-Reflection in Intelligent Technical Systems, in Self-Adaptive and Self-Organizing Systems Workshops (SASOW), London, UK, pp. 150–159, (2014).

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

[39] M. Kwiatkowska, G. Norman, and D. Parker, Advances and challenges of probabilistic model checking, In 48th Allerton Conference on Communication Control and Computing, (2010).

[40] D. Garlan, B. Schmerl, S. Cheng, A. Huang, and P. Steenkiste, Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure, in Computer, vol. 37, pp. 46-54, (2004).

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

[42] S. Cheng, V. Poladian, D. Garlan, and B. Schmerl, Improving architecture-based self-adaptation through resource prediction, Software Engineering for Self-Adaptive Systems, pages 71–88, (2009).

[43] 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).

[44] D. Weyns and M. U. Iftikhar, Model-based Simulation at Runtime for Self-adaptive Systems, 13th International Conference on Autonomic Computing, (2016).

[45] M. Gorbovitski, T. Rothamel, Y. A. Liu, and S. D. Stoller, Efficient runtime invariant checking: a framework and case study, in Proc. of the 2008 Int. Workshop on Dynamic Analysis, (2008).

[46] K. Zee, V. Kuncak, M. Taylor, and M. Rinard, Runtime checking for program verification, in Proc. of the 7th Int. Conf. on Runtime Verification, (2007).

[47] S. Ray, J. Park, and S. Bhunia, Wearables, Implants, and Internet of Things: The Technology Needs in the Evolving Landscape, IEEE Transactions on Multi-Scale Computing Systems, (2016).

[48] J. Padgette et. al., Guide to Bluetooth Security, National institute of standards and technology, (2017).

[49] C. Wang, C. Guo, Y. Wang, Y. Chen, and B. Liu, Friend or Foe?: Your Wearable Devices Reveal Your Personal PIN., Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, (2016).

[50] A. Yohan, N. Lo, V. Randy, S. Chen, and M. Hsu, A Novel Authentication Protocol for Micropayment with Wearable Devices, Proceedings of the 10th International Conference on Ubiquitous Information Management and Communication, (2016).

[51] A. Maiti, O. Armbruster, M. Jadliwala, and J. He, Smartwatch-Based Keystroke Inference Attacks and Context-Aware Protection Mechanisms, Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (ASIA CSS '16), (2016).

[52] R. Goyal, N. Dragoni, and A. Spognardi, Mind The Tracker You Wear – A Security Analysis of Wearable Health Trackers, Proceedings of the 31st Annual ACM Symposium on Applied Computing - SAC 16, (2016).

[53] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, Fog Computing and Its Role in the Internet of Things, Proceedings of the first edition of the MCC workshop on Mobile cloud computing (MCC '12), (2012).

[54] L.M. Vaquero and L. Rodero-Merino, Finding your Way in the Fog: Towards a Comprehensive Definition of Fog Computing, SIGCOMM Comput. Comun. Rev. 44, (2014).

[55] C. W. Walter, The personal fog: an architecture for limiting wearable security vulnerabilities, Ph.D. Dissertation, University of Tulsa, (2018).

[56] N. Kraft, Here One. https://hereplus.me/products/here-one, (accessed September 2020).

[57] S. O'Kane, Bragi Dash Pro Review. https://www.theverge.com/2017/7/25/16017768/bragi-dash-pro-review-wireless-headphones-price, 2017, (accessed September 2020).

[58] Jabra Elite Sport. https://www.jabra.com/sports-headphones/jabra-elite-sport, (accessed September 2020).

[59] Garmin. https://www.garmin.com/en-US, (accessed September 2020).

[60] M. Jackson, The meaning of requirements, Ann. Softw. Eng. 3, pp. 5–21, (1997).

[61] L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer, The Lean Theorem Prover, 25th International Conference on Automated Deduction (CADE-25), Berlin, Germany, (2015).

[62] Lean theorem prover, v 3.4.2. https://github.com/leanprover/lean, 2020, (accessed September 2020).

[63] excaLibur Lean library. https://github.com/ttowncompiled/excaLibur/tree/example-fgcs, 2020, (accessed September 2020).

[64] S. Hudon, Embedding Specialized Proof Languages into Lean: A Case Study, Lean Together 2019. https://lean-forward.github.io/lean-together/2019/slides/hudon.pdf, 2019, (accessed September 2020).

[65] A. Baanen, A. Bentkamp, J. Blanchette, and J. Hölzl, The Hitchhiker's Guide to Logical Verification 2020 Edition. https://github.com/blanchette/logical_verification_2020/blob/master/hitchhikers_guide.pdf, 2020, (accessed September 2020).

[66] Lean Forward, VU Amsterdam. https://github.com/blanchette/logical_verification_2020, 2020, (accessed September 2020).

[67] Lean theorem prover, v4. https://github.com/leanprover/lean4, 2020, (accessed September 2020).

[68] L. Esterle and J. N. Brown, Levels of Networked Self-Awareness, in 2018 IEEE 3rd International Workshops on Foundations and Applications of Self* Systems (FAS*W), pp. 237–238, (2018).

[69] C. Krupitzer; F. M. Roth, M. Pfannemüller, and C. Becker, Comparison of Approaches for Self-Improvement in Self-Adaptive Systems, In: Proceedings of the 2016 IEEE International Conference on Autonomic Computing (ICAC), pp. 308-314, (2016).

[70] G. A. Moreno, B. Schmerl, and D. Garlan, "SWIM: An Exemplar for Evaluation and Comparison of Self-Adaptation Approaches for Web Applications," 2018 IEEE/ACM 13th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), Gothenburg, pp. 137-143, (2018).

[71] S. W. Cheng, D. Garlan, and B. Schmerl, Evaluating the effectiveness of the rainbow self-adaptive system, In Workshop on Software Engineering for Adaptive and Self-Managing Systems, (2009).

[72] S.W. Cheng and D. Garlan. Stitch: A language for architecture-based self-adaptation, Journal of Systems and Software, (2012).

[73] M. Binkhonain and L. Zhao, A Review of Machine Learning Algorithms for Identification and Classification of Non-

Functional Requirements, Expert Systems with Applications: X. 1, (2019).

[74] Joint Task Force Transformation Initiative, Security and privacy controls for Federal Information Systems and Organizations, NIST Special Publication 800-53 revision 4, (2015).

[75] N. Swamy. https://www.microsoft.com/en-us/research/blog/project-everest-advancing-the-science-of-program-proof, 2019, (accessed September 2020).

[76] J. Protzenko and B. Parno, https://www.microsoft.com/en-us/research/blog/evercrypt-cryptographic-provider-offers-developers-greater-security-assurances, (2019).

[77] J. Backes et al., Semantic-based Automated Reasoning for AWS Access Policies using SMT, In: Proceedings of the 18th International Conference on Formal Methods in Computer-Aided Design (FMCAD), pp. 1-9, (2018).

[78] S. Farokhi, P. Jamshidi, I. Brandic, and E. Elmroth, Self-adaptation challenges for cloud-based applications: A control theoretic perspective, In Proceedings of the 10th International Workshop on Feedback Computing, (2015).

[79] H. Muccini, M. Sharaf, Mohammad, and D. Weyns, Self-Adaptation for Cyber-Physical Systems: A Systematic Literature Review, 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, (2016).

[80] R. Calinescu, D. Weyns, S. Gerasimou, and I. Habli, Architecting Trustworthy Self-Adaptive Systems (Tutorial), 2019 IEEE International Conference on Software Architecture Companion (ICSA-C), (2019).

Appendix A: Colored Petri Net Transition Rules

P refers to a pink token. B refers to a blue token. R refers to a red token. T refers to a transition. B' refers to the state of the blue token after it is output by a transition.

Rule	Description	Condition	Action
1	Blue token has cycled completely	• $\boldsymbol{B.visited}$ **is empty** and $\boldsymbol{trigger = 0}$	Remove Blue and Pink tokens from CPN
2	Blue token has visited the place and had a previous conflict but does not have a conflict with a VC from a processed pink token with a matching transition VC	• T.placeName is in B.visited • T.placeName is in one of the tuples of B.vcConflicts • NO VC in T.vcInfo appear in a tuple in B.vcChanges that are not already in tuples in B.vcConflicts associated with T.placeName	• Remove T.placeName from B'.visited • Send B' to next place • Send P to its place
3	Blue token has visited the place and had a previous conflict, and does have a conflict with a VC from a processed pink token with a matching transition VC	• T.placeName is in B.visited • T.placeName is in one of the tuples of B.vcConflicts • VC in T.vcInfo appear in a tuple in B.vcChanges that are not already in tuples in B.vcConflicts associated with T.placeName	• FORALL VC in T.vcInfo that are also in tuples in B'.vcChanges ▪ Increase B'.maxIdx for the unique ID of red token ▪ Create a red token, R, with the proper idx_{token} and other information held by B and T ▪ Update B'. vcConflicts by increasing $idx_{conflict}$ and the appropriate information held by T for all matching VCs • Remove T.placeName from B'.visited • Send B' to next place • Send P to its place • Send all red tokens to End place
4	Blue token has visited the place, but has not had a previous conflict match at this place and does have a conflict with a VC from a processed pink token with a matching transition	• T.placeName is in B.visited • T.placeName is NOT in one of the tuples of B.vcConflicts • VC in T.vcInfo that's also in a tuple in B.vcChanges	• FORALL VC in T.vcInfo that are also in tuples in B'.vcChanges, ▪ Increase B'. maxIdx for the unique ID of red token ▪ Create a red token, R, with the proper idx_{token} and other information

	VC		held by B and T ▪ Update B'.vcConflicts by increasing $idx_{conflict}$ and the appropriate information held by T for all matching VCs • Remove T.placeName from B'.visited • Send B' to next place • Send P to its place • Send all red tokens to End place
5	Blue token has visited the place, has not had a previous conflict match at this place, and does not have a conflict with a VC from a processed pink token with a matching transition VC	• T.placeName is in B.visited • T.placeName is NOT in one of the tuples of B.vcConflicts • NO VC in T.vcInfo appear in a tuple in B.vcChanges that are not already in tuples in B.vcConflicts associated with T.placeName	• Remove T.placeName from B'.visited • Send B' to next place • Send P to its place
6	Blue token has not visited the place, does not have a conflict with a VC from a processed pink token with a matching transition VC, and there is no conflict with any pink token VC and transition VC	• T.placeName is NOT in B.visited • T.VC is NOT in one of the tuples of B.vcChanges • NO VC in P.VC is has conflicts a VC in T.vcInfo	• Add T.place_name to B'.visited • Send B' to next place • Send P to its place
7	Blue token has not visited the place and does have a conflict with a VC from a processed pink token with a matching transition VC, but there is no conflict with any pink token VC and transition VC	• T.placeName is NOT in B.visited • VC in T.vcInfo that's also in a tuple in B.vcChanges • NO VC in a tuple in P conflicts with a VC in T.vcInfo	• FORALL VC in tuples of in T.vcInfo that are also in tuples in B'.vcChanges, ▪ Increase B'.maxIdx for the unique ID of red token ▪ Create a red token, R, with the proper idx_{token} and other information held by B and T ▪ Update B'.vcConflicts by increasing $idx_{conflict}$ and the appropriate information held by T

			for all matching VCs • Add T.placeName to B'.visited • Send B' to next place • Send P to its place • Send all red tokens to End place
8	Blue token has not visited the place, does not have a conflict with a VC from a processed pink token with a matching transition VC, but does have a conflict with a pink token VC and transition VC	• T.placeName is NOT in B.visited • VC in T.vcInfo appear in a tuple in B.vcChanges • A VC in a tuple in P conflicts with a VC in T.vcInfo	• FORALL VC in T.vcInfo that appear in a tuple in P ▪ Increase B'.maxIdx for the unique ID of red token ▪ Create a red token, R, with proper idx_{token} and other information held by P and T ▪ Update B'.vcConflicts by increasing $idx_{conflict}$ and the appropriate information held by T for all matching VCs ▪ Update B'.vcChanges by increasing idx_{change} and the appropriate change information from P • Add T.placeName to B'.visited • Send B' to next place • Send P to its place • Send all red tokens to End place
9	Blue token has not visited the place, does have a conflict with a VC from a processed pink token with a matching transition VC, but does have a conflict with a pink token VC and transition VC	• T.placeName is NOT in B.visited • VC in T.vcInfo that's also in a tuple in B.vcChanges • A VC in a tuple in P conflicts with a VC in T.vcInfo	• FORALL VC in T.vcInfo that are also in tuples in B'.vcChanges, ▪ Increase B'.maxIdx, for the unique ID of red token ▪ Create a red token, R, with the proper idx_{token} and other information held by B and T ▪ Update B'.vcConflicts by increasing $idx_{conflict}$ and the appropriate information held by T for all matching VCs

			• FORALL VC in T.vcInfo that appear in a tuple in P ▪ Increase B'.maxIdx for the unique ID of red token ▪ Create a red token, R, with proper idx_{token} and other information held by P and T ▪ Update B'.vcConflicts by increasing $idx_{conflict}$ and the appropriate information held by T for all matching VCs ▪ Update B'.vcChanges by increasing idx_{change} and the appropriate change information from P • Add T.placeName to B'.visited • Send B' to next place • Send P to its place • Send all red tokens to End place