2018 ACM/IEEE 13th International Symposium on Software Engineering for Adaptive and Self-Managing Systems

Toward Evaluating the Impact of Self-adaptation on Security Control Certification

Allen Marshall
University of Tulsa
800 S. Tucker Dr.
Tulsa, OK USA
allen-marshall@utulsa.edu

Sharmin Jahan
University of Tulsa
800 S. Tucker Dr.
Tulsa, OK USA
shj594@utulsa.edu

Rose Gamble
University of Tulsa
800 S. Tucker Dr.
Tulsa, OK USA
gamble@utulsa.edu

ABSTRACT

Certifying security controls is required for information systems that are either federally maintained or maintained by a US government contractor. As described in the NIST SP800-53, certified and accredited information systems are deployed with an acceptable security threat risk. Self-adaptive information systems that allow functional and decision-making changes to be dynamically configured at runtime may violate security controls increasing the risk of security threat to the system. Methods are needed to formalize the process of certification for security controls by expressing and verifying the functional and non-functional requirements to determine what risks are introduced through self-adaptation. We formally express the existence and behavior requirements of the mechanisms needed to guarantee the security controls' effectiveness using audit controls on program example. To reason over the risk of security control compliance given runtime self-adaptations, we use the KIV theorem prover on the functional requirements, extracting the verification concerns and workflow associated with the proof process. We augment the MAPE-K control loop planner with knowledge of the mechanisms that satisfy the existence criteria expressed by the security controls. We compare self-adaptive plans to assess their risk of security control violation prior to plan deployment.

CCS CONCEPTS

• **Software and its engineering** → Software creation and management → Software verification and validation • **Security and privacy** → Software and application security

KEYWORDS

Security certification, self-adaptation, risk analysis

SEAMS '18, May 28–29, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5715-9/18/05…$15.00
https://doi.org/10.1145/3194133.3194139

1 INTRODUCTION

U.S. federal information systems require security certification based on the security controls found in the NIST SP800-53 [1]. Its companion document, the NIST SP800-53a [2] breaks down each security control and outlines the applicable procedures to assess the mechanisms put in place to guarantee security control effectiveness. Examination procedures detail the type of documentation, design, and code that should be reviewed. Interviewing procedures indicate which are the relevant stakeholders for questioning. Testing procedures relate not only to information system implementation, but also to organizational processing. The assessment results in a "satisfactory" rating for each part of a security control or "other than satisfactory".

With data analytics tools, information systems will have the capability to self-adapt based on existing and historical data. Adaptations may be dynamically configured at runtime and not be known to certifiers prior to deployment. Though the adaptations may be forced to comply with coding rules for specific security properties as assessed by a static code analyzer, it is plausible that they can still pass these checks and negatively impact a security control's effectiveness. Information systems should have security certification awareness to reason about and compare adaptations according to the potential risks they may impose. Our approach examines the security controls found in the NIST SP800-53 [1] and defines this awareness initially to include the prescribed existence of the mechanisms that guarantee the security controls are in place and the verification of each mechanism's guarantee of the security control's effectiveness.

In this paper, we introduce a methodology to incorporate security certification awareness within a self-adaptive information system. The methodology is based on documenting the process of verification and validation of the security controls in mechanisms that guarantee their compliance. We examine a subset of the Audit security controls in the NIST SP800-53 [1] to extract features, properties, and constraints that affect the successful determination of a security control's effectiveness by a mechanism, which is represented by a component and/or processes. We separate the security controls requirements of mechanism existence (non-functional) from the expected behavior of the mechanisms (functional). Linear Temporal Logic (LTL) [3] is used to express functional requirements, as safety and liveness properties. Existence requirements are disclosed to the MAPE-K control loop [4] planner to scope the configured adaptive plans. We use the KIV [5] theorem prover to verify the

functional requirements, applying methodological guidelines to the proof process in order to extract the state variables examined with the proof (*verification concerns*) and model the KIV process workflow (*verification workflow*) for each security control proof. Using a program example, we incorporate our prior risk assessment and adaptive plan comparison methodology [6] to assess how self-adaptations impact security certification.

2 RELATED WORK

Self-adaptive systems should strive to maintain quality of service, including requirements compliance guarantees, with an understanding that the system is in a changing and uncertain environment. Though significant research is introducing self-adaptive capabilities into a broad range of systems and domains, the inherent uncertainty associated with the need for adaptation still needs to be addressed during system design. Recognizing types of uncertainty and their sources can mitigate negative effects of changes and increase the assurance level of an adaptive plan [7]. The approach works using decomposition mechanisms and model-driven techniques for assuring a plan's compliance, and the properties for choosing effective mechanisms are outlined. Esfahani and Malek [8] also focus on identifying the sources and degree of uncertainty in a self-adaptive system. The difficulty in predicting the environmental uncertainty implies that there may be a lack of coverage in the available adaptation techniques at runtime. Thus, the effectiveness of the approach depends on the size of the self-adaptive system and its amount of allowable dynamism in creating plans.

To facilitate specification, verification, and validation of an adaptive program, one method is to use formal models of the system specification and then automatically generate adaptive plans that can then be directly validated [9], [10]. Blair, et al. [11] propose a mechanism to create models during runtime. They dynamically develop a model and evaluate its compliance with system requirements. The mechanism is used to configure a new component as part of an adaptation as a replacement for a failed component at runtime [12], [13], [14] [15]. The mechanism depends on a particular specification language for the system's behavior as a model that is interpretable by a machine at runtime, which is difficult for large systems and for complex adaptive plans. Trapp and Schneider [16] conducted a survey on these approaches and identified gaps for applying safety assurance techniques in adaptive systems.

Adaptation effectiveness depends heavily on the design and choice of adaptive plans. Proper specification of adaptation requirements can be performed by formally expressing them in the temporal logic A-LTL [17]. The approach introduces an *adapt* operator to Linear Temporal Logic (LTL) and specifies semantics for adaptation variants, clarifying the system's requirements and leading to correct implementations during the software development phase. These specifications and semantics must be known prior to plans being constructed. Coker, et al. [18] apply stochastic search techniques to design optimized self-adaptive plans by traversing the space of candidate solutions and assigning the candidate a score based on its closeness to the solution. The approach has implications for adaptive plan modeling but needs proper situational knowledge to choose the appropriate stochastic search techniques for obtaining the best plan. Other research uses formal methods to evaluate the worst-case behavior of self-adaptation mechanisms when the possible adaptations are known prior to deployment [19]. Rather than verifying a system that is ready for deployment, they verify high-level design decisions early in the development process, comparing designs based on their worst-case guarantees under a probabilistic model. The decision-making approaches for self-adaptive systems often suffer from conflicting requirements to balance quality and timeliness for adaptive plans. Pandey, et al. [20], resolves the issue by dynamically determining the level of optimization the adaptive plan must achieve, using Rainbow [21] to create suboptimal plans quickly.

Assurance techniques provide a confidence level associated with the acceptance of an adaptive plan, given claims, arguments, evidence, and expertise that it complies with system goals and requirements. Schmerl, et al. [22] illustrates the use of Goal Structuring Notation (GSN) to decompose assurance cases and associate potential adaptations to different subgoals. Evidence collection within the MAPE-K loop is described that enables this decomposition and provides the confidence that the subgoals can be met. The challenges of composing assurances cases are structuring the arguments with the dynamic changes of the system and employing mapping strategies with evidence to confirm that the higher level goal holds. Challenges with the approach include evidence reuse, model design to best obtain the evidence and allow for analysis and goal mapping, and scalability to a large number of systems for composition.

From the security perspective, Anisetti, et al., [23] focus on assurance techniques for cloud security certification requirements by introducing a test-based approach that involves the service provider, cloud provider, and certificate authority to define what security properties to target in the design and how to deploy a cloud service with evidence of the security property effectiveness. The objective is to adapt the cloud only when a valid certificate can be established for the security properties. Certificates are associated with predefined adaptations, limiting the potential changes. Using this experience, a test-based security certification framework is proposed [24], [25], [26], [27] to check if a machine-readable system architecture model complies with the system requirements, feeding into the certification process.

Ardagna, et al. [28] propose an approach of continuous verification of security certification validity, but it is a semi-dynamic approach demanding prior identification of stable security properties over the system configurations. In addition, it requires pinpointing the verification activities to query alternative configurations for the features they must satisfy. Tsigkanos, et al [29] use a model-based self-adaptation approach for security control systems. Ambient Calculus is used to represent the topology of the operational environment with Computation Tree Logic used to specify security requirements, providing topology awareness to prevent requirements violations at runtime. The approach illustrates the trade-offs between the completeness of

the model validation operations and the timeliness of determining and deploying an adaptation.

Self-protecting systems would have the capability to detect security threats and deploy runtime adaptation techniques to mitigate them. An architecture based self-protection approach is proposed [30], [31], which relies on the Rainbow framework [21] for representation of the target system and embedding the security posture needed to assess it. Both the target system and environment are monitored using probes and gauges, resulting in defined threat detection, as well as updates to the architecture model's properties based on available adaptation strategies. The challenges are defining and expressing effective security metrics at architecture level given the system functionality and requirements at design time.

3 SECURITY CERTIFICATION AWARENESS

To scope our examination of the impact of self-adaptive plans on security certification, we start with two assumptions: (1) the planning process as part of the MAPE-K (Monitor, Analyze, Plan, Execute with Knowledge) [4] can produce adaptive plans that may not have been evaluated prior to system deployment, and (2) a static analysis tool can be available to disallow any dynamically configured plans that do not follow predefined rules, such as naming conventions. The first assumption allows the planner to go beyond prefabricated plans or plans that must satisfy predefined adaptive constraints, given the situational awareness available to it. The planner can use any knowledge of predefined plans, attempted but not deployed plans, and meta-data from successfully deployed plans. The second assumption allows for constraints to be applied to the plan configuration. With this assumption, we can introduce constraints on deleting processes that explicitly serve as security control mechanisms as part of an adaptive plan configuration.

Because each requirement verification or certification process is documented, the control flow used and the states examined within the process provide meta-data in the form of verification concerns (VCs) and a verification workflow (VFlow). VCs represent the state variables needed to establish requirement compliance. A VFlow represents the strategy performed by the process as it examines the system functions and states. Our approach revolves around the premise that if an adaptive plan interferes with or alters one or more VCs at a risky point in a VFlow associated with requirement compliance guarantees, then there is an increased risk that the original verification/certification process or strategy cannot be reused. The greater the risk, the higher the probability that a new process would be needed, increasing the probability of a requirement violation by the adaptive plan. We embed VCs and VFlows as part of the planning process within the MAPE-K control loop for verification and certification awareness. Given multiple, potential adaptation plans, we perform a risk assessment to determine the least risky.

3.1 Transforming Security Controls

Because security controls express functional and non-functional requirements, certification processes involve determining the existence of certain functionality in the system, as well as the correctness of the functionality. In this section, we focus on security control AU-12(1), which appears in Figure 1, as taken directly from the NIST SP 800-53 [1, p. F-51 – F-52]. AU-12 is designated all baselines (i.e. low, moderate, and high impact systems). This means that all information systems adhering to the security controls must consider AU-12. Enhancement 1 is part of the baseline of controls for high impact systems. A high impact information system means that there is a high degree of concern, such as monetary, reputation, or life-threatening, should one of confidentiality, integrity, or availability be violated with respect to the data stored and in transit.

Figure 1 shows that AU-12 references additional audit controls, namely AU-2a, AU-2d, and AU-3. These control statements appear in Figure 2 as taken directly from the NIST SP800-53 [1, p. F-41 – F-42]. Security certification examines the requirements of AU-12, followed by the requirements for its first enhancement and assessing these against the system. The NIST SP800-53a [2, p. F-81 – F-82] guidelines state that examination can be done on the "procedures addressing audit record generation" because of the reference to AU-2a and "list of auditable events" because of the reference to AU-2d and AU-3. When enhancement (1) is included, the examination guidelines extend to "system-wide audit trail".

AU-12 AUDIT GENERATION

Control: The information system:

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

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

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

Control Enhancements:

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

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

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

3.2 Expressing the Control Statements

Formally expressing the security control statements manifests their separation of non-functional from functional requirements. Below we restate them as R1-R6 using formal notation and the ontological relationships expressed in [32]. In LTL, the square is "invariant" and the diamond is "eventually".

R1: (non-functional) For each component (C) in the information system (IS) identified to have auditable events, there

exists a function to perform auditing for predefined auditable events (AU-12a, AU-2a). Below, we call the function *generateAuditRecord*.

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

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

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

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

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

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

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

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

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

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

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

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

AU-2 AUDIT EVENTS

Control: The organization:

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

…

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

AU-3 CONTENT OF AUDIT RECORDS

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

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

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

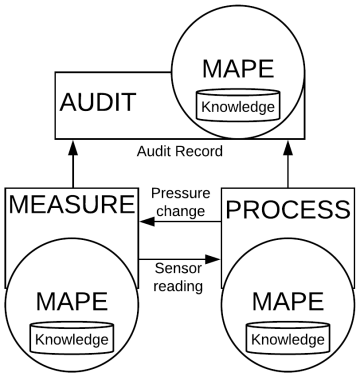

Figure 3: SIMS Architecture

4 EXAMPLE PROBLEM

Figure 3 shows the SIMS architecture, implemented as multiple threads in Java. SIMS tracks inventory's stock condition using a pressure sensor and adjusts a pressure threshold depending on the inventory movement. SIMS has three components with local MAPE-K control loops: Measure reads the pressure sensor, Process adjusts the threshold, and Audit houses audit records from Measure and Process for security certification. The Measure and Process components contain functionality for audit record generation as a requirement of security controls. This functionality includes sensing auditable events, generating audit records, and sending audit records on a channel to be received by the Audit component and appropriately placed in the audit trail. Measure sends its sensor readings to Process and receives changes in the pressure threshold associated with those readings. Process receives the sensor values and determines if the pressure threshold should change given the readings over a period of time.

4.1 Examining the Audit Security Controls

Figures 1 and 2 have bracketed, italicized text preceded by "Assignment:" that are directives to tailor the security controls. Once tailoring is performed, the resulting values cannot be changed without re-certification, and thus, form safety properties. Table 1 shows the tailored values chosen to satisfy AU-12(1), AU-2a, AU-2b, and AU-3 for the SIMS program example. Note that the auditable events in row 6 indicate that signal updates (from Measure) and threshold value changes (from Process) should cause a record generation and sending to Audit.

Once tailoring is completed, certification can begin. For the targeted audit security controls in Section 3, the first part of the process is to determine if the required mechanisms exist to satisfy **R1** and **R3**. For **R1**, Table 2 indicates that Measure and Process are responsible for record generation (row 1) and produce audit records from the auditable events of signal updates in Measure and threshold value changes in Process (row 5).

Table 1: Tailored Values for the Targeted Security Controls

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

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

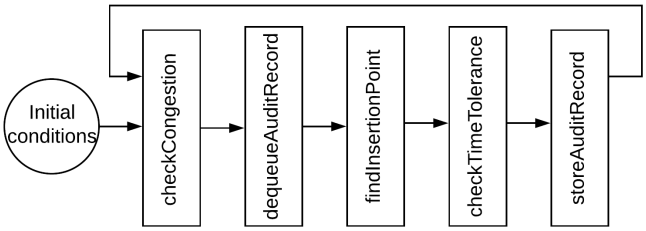

Figure 4: Process Control Flow for the SIMS Audit Component

We focus the requirements proof and self-adaptation assessment on the Audit component. Its process control flow is depicted in Figure 4. The main update loop consists of checkCongestion (logs the amount of buffer capacity being used), dequeueAuditRecord (removes an audit record from the queue), findInsertionPoint (uses a binary search to determine where the audit record's timestamp fits), checkTimeTolerance (logs when the time difference is not within the prescribed tolerance), and storeAuditRecord (stores the audit record at the determined insertion point of the audit trail).

4.2 Potential Adaptation Plans

Assume monitoring finds that too often checkCongestion is logging high buffer capacity. The planning process configures the following potential plans for risk assessment that would decrease the time for audit record insertion into the audit trail. The adaptation would subsequently allow for faster dequeuing, reducing congestion. None of the adaptation plans violate the existence criteria for **R1** and **R3**.

A1: Allow Audit to periodically drop queued audit records.

A2: Add dequeued records to the end of the audit trail (instead of performing binary search) until a predefined number of records have been added.

A3: Increase the performance of the sorting technique by providing a rolling lower timestamp and sorting only records less than it.

Table 2: Targeted Functions for Audit Control Certification of Mechanism Existence

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

We will return to these adaptations and their impact on the verification of the audit security controls in Section 6.

5 FROM PROOFS TO VERIFICATION CONCERNS AND WORKFLOWS

To compare adaptations configured by the planner, we assess their risk of inhibiting the reuse of the original verification process of the security control requirements. In this section, we outline the methodology of extracting the verification concerns (VCs) and the verification workflows (VFlows) using the KIV

theorem prover. We focus on proving the safety property **R5** and the progress property **R4** from Section 3.2. Figure 5 outlines the process to identify the VCs and VFlow associated with a proof. With the Java code restated in KIV's language, KIV can prove LTL expressions. We define a set of lemmas that provide guidance to the KIV proof so that the proof process is made more explicit and the meta-information needed to identify VCs and VFlows per requirement is available.

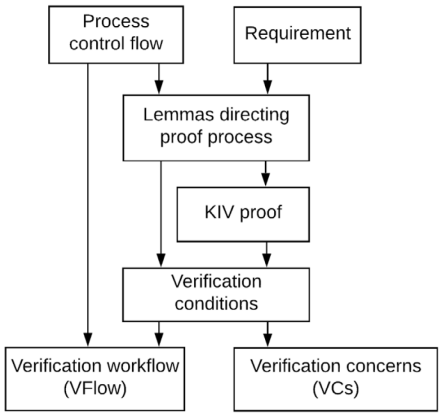

Figure 5: VC and VFlow Identification Process

5.1 Working with KIV

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

Hoare triples [33] are an intuitive way to prove program properties and extract VCs. However, they make no guarantees about the states in between their pre- and postconditions. Thus, they are not suitable for proofs where these states must satisfy a property. To allow KIV to use Hoare triples when proving a property for a single component, we augment them with intermediate state and termination guarantees. Our *temporal contract proposition* (TCP) is similar to KIV's *rely-guarantee* statements for threads but can be used to decompose the different procedure calls in an individual program thread.

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

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

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

5.2 Verification of Safety Property R5

In KIV, invariants to be proven are often represented as

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

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

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

To actually skip over *Code* we need to derive a new proof goal where the program formula is *RestProg*. We apply the following lemma, called *lemma-invariant*, after applying the TCP:

$N = n$,
$\Box$(*After*$_I$ ∧ *InvProp* ∧ $N \leq n \Rightarrow \neg(N = N'' + 1$ **until** $\neg$*InvProp*)),
$\Box$($Mid_I \Rightarrow$ InvProp),
Mid_I **until** (InvProp ∧ $After_I$)
⊢
$\neg(N = N'' + 1$ **until** $\neg$*InvProp*)

We apply *lemma-invariant* using the substitution $After_I$ = *Post* ∧ *RestProg* and Mid_I = Mid. InvProp is substituted with the invariant property to be proven (i.e., to prove $\Box$ p, we use InvProp = p). Given that Post contains InvProp, the fact *Mid* **until** (*Post* ∧ *RestProg*) derived from the TCP implies Mid_I **until** (*InvProp* ∧ $After_I$). N = n can always be established for some n.

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

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

Figure 6 shows a fragment of a KIV proof tree in which a TCP has been applied, followed by an application of *lemma-invariant*. In this case, KIV was able to automatically close the proof goal $Mid_I \Rightarrow InvProp$, resulting in only one new proof goal.

Returning to the audit security controls, we apply the proof strategy described above to requirement R5 for Audit, restated below.

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

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

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

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

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

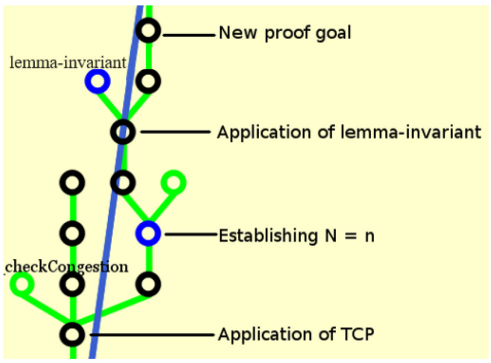

Figure 6: KIV Tree showing Guided Invariant Proof Process

In our proof of R5, we first show that $\Box s$ implies R5, making $\Box s$ an intermediate invariant to be proven. t is an intermediate condition needed in some of our TCPs. Intuitively, t means that there is no audit record being passed to the process in the control flow (see Figure 4) *or* no insertion point has been selected *or* inserting the audit record at the insertion point preserves the sorting. The use of nullOpt and getOpt in this formula arise from our modeling of *optional* sorts, which can have null values.

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

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

Note that the appropriate use of extra conditions such as t must be determined by the human user. Care is needed to ensure that the TCPs are both provable and sufficient to prove the invariant. After manually applying these TCPs in KIV and following each TCP application with *lemma-invariant*, we reach a proof goal where the program formula is:

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

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

5.2.1 Identifying Verification Concerns

Verification concerns (VCs) are state variables with values that are relevant to the proof. VCs are identified by verification conditions (see Figure 5) that are examined within the instantiated *lemma-invariants* (see Section 5.2) associated with the TCPs used to guide the KIV proof. For example, the conditions

auditRecord = nullOpt
insertionPoint ≤ #auditTrail

are a subset of conditions needed to prove R5 within the processes findInsertionPoint, checkTimeTolerance, and storeAuditRecord. The conditions rely on auditRecord, insertionPoint, and auditTrail, identifying them as VCs for R5, as well as the processes where the conditions are checked, which will be used in the construction of the Colored Petri Net (CPN) [34], [35] that describes the associated verification workflow [6].

5.2.2 Classifying the Conditions on Verification Concern Changes

Once the VCs are identified, the conditions used for the proof indicate the range of potential changes to the variables that may increase risk. While it is plausible that finding the VCs could be automated given the resulting KIV proof, the flexibility with which the VCs can be changed by self-adaptation is still a manual procedure. Table 3 shows how a human analyst might classify the impact of certain changes to a VC given the conditions used in the proof of R5. We separate them into fuzzy sets of Devastating, Worrisome, and Unconcerned.

Table 3: R5 VC Conditions for Change Impact

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

Removing auditRecord from auditTrail causes record loss, making both changes potentially devastating to maintaining R5. Setting insertionPoint to null and eliminating sorting is

problematic but decreasing the performance of the sorting algorithm may not cause problems. A change in the condition table that is part of an adaptation will be reflected in the risk impact factor, M_{VC}, for the targeted VC.

5.2.3 Identifying the Verification Workflow

In Figure 5, the process control flow (such as in Figure 4) and the verification conditions (such as those discussed in Section 5.2.1) that result from the KIV proof contribute to the definition of the verification workflow (VFlow) per requirement. We rely on prior work for the construction of a CPN as the VFlow representation [6]. The objective of the VFlow is to inform the planning process of the comparative risk values associated with potential adaptation on the proof reuse for that requirement.

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

Figure 7 represents the VFlow crafted by the meta-data taken from the proof of R5. Three processes, findInsertionPoint, checkTimeTolerance, and storeAuditRecord, are represented as transitions. Because the processes are equally involved in the proof, a change to them has equal impact on proof reuse. This will be reflected in the process impact multiplier, M_{PL}. For adaptation A2, the pink token indicates that the change will occur within findInsertionPoint. The planning process assigns values representing its assessment of the plan quality, which is the risk impact multiplier $\hat{p}$. The blue token carries the information presented by the pink token to each transition for assessment.

Red tokens are produced when a VC has an impact value above 0. They carry the accumulated impact values, M_{VC}, M_{PL}, and $\hat{p}$ to the end state. Multiple red tokens can be produced if multiple VCs are affected and if multiple processes contribute to the potential risk. For R5, adaptation A2 causes three red tokens to be generated, one from each transition, meaning that a VC at the transition was impacted. This VC is insertionPoint for each red token because A2 invokes the condition "Continuous set to #auditTrail" which is devastating. We will discuss how the impact values are used in Section 6.

5.3 Verification of the Progress Property R4

The original requirement R4 from the description of the security controls is restated below.

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

$\square(recordGenerated(e) \Rightarrow \diamondsuit(e \in auditTrail))$

For Audit, we decompose R4 into two progress properties.

R4.1: $\square(recordGenerated(v) \Rightarrow \diamondsuit(v \in InQ))$

R4.2: $\square((v \in InQ) \Rightarrow \diamondsuit(v \in \text{Ext}.auditTrail))$

In this section, we outline the methodology used to guide KIV to prove R4.2. We use e and s as defined in Section 5.2, along with the following abbreviations:

$e_1 \equiv \square(\text{messageVar} \in \text{InQ} \wedge \text{InQ}' = \text{InQ} \Rightarrow \text{messageVar} \in \text{InQ}'')$

$s_1 \equiv \text{messageVar} \in \text{InQ} \Rightarrow \text{messageVar} \in \text{InQ}''$

$s_2 \equiv \text{Int.auditRecord} \neq \text{nullOpt} \wedge$

$\text{Int.auditRecord.getOpt} = \text{messageVar.auditRecord}$

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

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

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

$\square(q \Rightarrow \diamondsuit p)$, where

$q \equiv \text{messageVar} \in \text{InQ} \wedge \text{isRecordMessage(messageVar)}$

$p \equiv \text{messageVar.auditRecord} \in \text{Ext.auditTrail}$

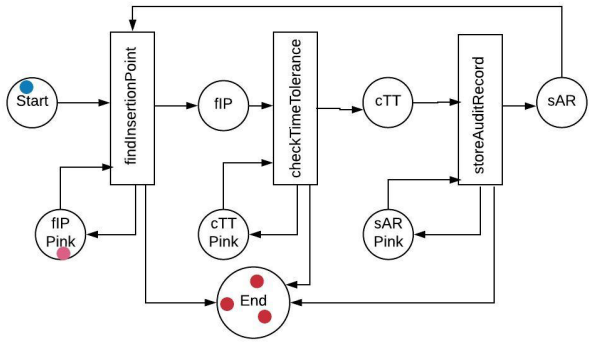

Figure 7: VFlow for Safety Property R5 and Adaptation A2

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

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

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

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

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

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

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

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

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

Pre, [: V_{in}, V_{inout}| Code(V_{in}; V_{inout}); [PL RestProg]]
⊢
(Mid **until** (Post1 ∧ RestProg)) ∨ ◇(Post2 ∧ RestProg)

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

N = n,
□(After$_{PS}$ ∧ N ≤ n ⇒
¬(N = N" + 1 until (PreProgress ∧ ¬◇PostProgress))),
□(After$_{PS}$ ⇒ ◇PostProgress),
◇After$_{PS}$
⊢
¬(N = N" + 1 **until** (PreProgress ∧ ¬◇PostProgress))

After applying a STCP, we apply *lemma-progress-split* with the substitution $After_{PS}$ = $Post_2$ ∧ *RestProg*, PreProgress = *q*, PostProgress = *p*. We get two new proof goals:

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

Figure 8 shows a fragment of a KIV proof tree where a STCP has been applied followed by an application of *lemma-progress* in one branch and *lemma-progress-split* in the other branch.

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

tcp(*e* ∧ e_1 ∧ □*s*, checkCongestion(; Int, Ext, InQ), s_1, *true*)
stcp(*e* ∧ e_1 ∧ □*s*, dequeueAuditRecord(; Int, Ext, InQ), s_1, *true*, s_2)
tcp(*e* ∧ e_1 ∧ □*s*,
findInsertionPoint(; Int, Ext, InQ);
checkTimeTolerance(; Int, Ext, InQ);
storeAuditRecord(; Int, Ext, InQ),s_1, *true*)

5.3.1 Identifying Verification Concerns

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

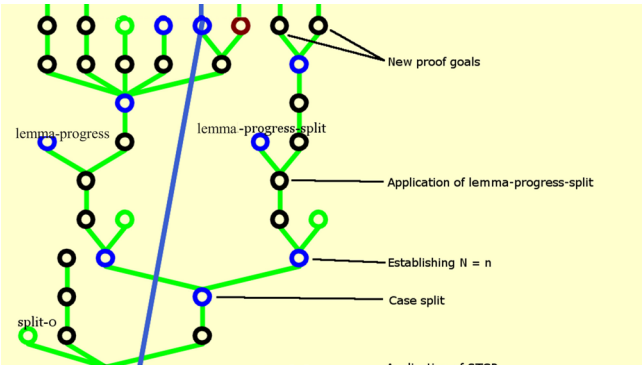

Figure 8: KIV Tree Displaying Progress Property Proof Process

5.3.2 Classifying the Conditions on VC Changes

Table 4 denotes the potential impacts of certain changes to the identified VCs from Section 5.3.1 for **R4.2**. It can be seen that the same changes in Table 3 are viewed as impacting this requirement differently. For example, sorting changes are of less concern because the requirement focuses on ensuring the audit records are stored in the audit trail.

Table 4: R4.2 VC Conditions for Change Impact

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

5.3.3 Identifying the Verification Workflow

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

The VFlow representing the meta-data from the proof process for **R4.2** involves all five processes in the SIMS case study, which will be equally impacted by changes affecting their VCs. Figure 9 shows the CPN representing **R4.2**'s VFlow. Notice that the pink token is associated with findInsertionPoint as it was in Figure 7. The same VC, insertionPoint, is affected, but the effect is less (unconcerned) than for **R5**.

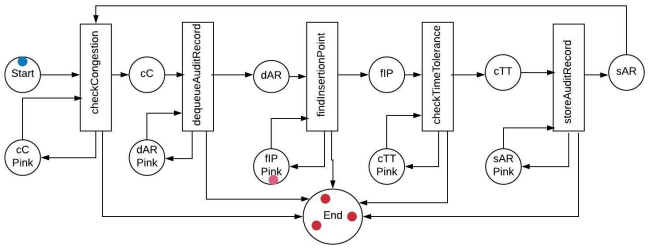

Figure 9: VFlow for R4.2 and Adaptation A2

6 COMPARING ADAPTATION RISK

We use a previously derived utility function [6] to compare adaptation plans A1-A3 according to the assessed risk they pose to inhibiting the reuse of the verification processes for R4 and R5. The success probability of an adaptation plan a with respect to a requirement r is estimated as $\prod_{t \in T(r,a)} p(t)$, where $T(r,a)$ is the set of red tokens generated from r's VFlow for adaptation plan a and $p(t)$ is an estimate of the probability that token t does *not* represent an actual reuse violation of a verification process. We consider probability scaling, where

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

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

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

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

6.1 Risk Valuation

Comparing the potential risk of adaptations on reusing a proof process means assigning values to the impact factors for $p(t)$. For M_{VC}, we assign VC conditions that match the plan changes and are devastating 0.2, worrisome 0.5, or unconcerned 0.9 (higher is less impact). The lowest impact factor is chosen if more than one condition matches. If there is no match, then $M_{VC} = 1$.

Adaptation A1 matches auditRecord's devastating condition "Set to null" for **R4.2** (Table 4, $M_{VC} = 0.2$), but is only worrisome for **R5** (Table 3, $M_{VC} = 0.5$), since time-correlating the auditTrail is unaffected, but dropping is problematic overall. Adaptation A2 matches insertionPoint's devastating condition "Continuous set to #auditTrail" for **R5** ($M_{VC} = 0.2$). For **R4.2**, this same condition is unconcerned ($M_{VC} = 0.9$) because the requirement focuses on auditTrail having all of the audit records produced. Adaptation A3 matches insertionPoint's unconcerned condition of "Alter sorting performance" ($M_{VC} = 0.9$) for **R4.2** and **R5**. Since the impact on the processes is the same, $M_{PL} = 0.5$. The AU-12 requirements are representative of the baseline security control and valued similarly. Therefore, we set $w(r) = 1$ for all requirements r. The planning process rates the adaptation quality. For A1: $\hat{p} = 0.25$, A2: $\hat{p} = 0.6$, A3: $\hat{p} = 0.75$.

6.2 Assessment Results

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

$E[U(A1)] = (0.25*0.2*0.5)^3 + (0.25*0.5*0.5)^3 = 2.60\times10^{-4}$
$E[U(A2)] = (0.6*0.9*0.5)^3 + (0.6*0.2*0.5)^3 = 1.99\times10^{-2}$
$E[U(A3)] = (0.75*0.9*0.5)^3 + (0.75*0.9*0.5)^3 = 7.69\times10^{-2}$

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

7 DISCUSSION AND CONCLUSION

We use an Audit security control from the NIST SP800-53 [1] to show that security certification involves non-functional requirements that mechanisms exist for security control compliance and functional requirements to guarantee the security control effectiveness of the mechanism. For self-adaptive systems, the existence requirement of the mechanism and its mapping to the security control(s) can be provided to the planning phase in the MAPE-K loop so that no adaptation plans include deleting the mechanism without replacement. For functional requirements, we show how KIV can be used for security control compliance to produce the metadata needed for our adaptive plan risk assessment technique. In terms of scalability, controls from the other technical families in the NIST SP800-53 have a similar statement structure, making them amenable to the approach. Additional research is needed to manage any negative interactions of adaptive plans that affect multiple security controls.

ACKNOWLEDGEMENT

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] NIST. 2013. Security and Privacy Controls for Federal Information Systems. NIST Special Publication 800-53, Revision 4.

[2] NIST. 2014. Assessing Security and Privacy Controls in Federal Information Systems and Organizations. NIST Special Publication 800-53A Revision 4.

[3] Q. Lichtenstein and A. Pnueli. 1985. Checking that Finite State Concurrent Programs Satisfy their Linear Specification. *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ACM.

[4] IBM. 2005. "An Architectural Blueprint for Autonomic Computing", Autonomic Computing White Paper, 3rd Edition, IBM, USA, June 2005, available at http://www-03.ibm.com/autonomic/pdfs/AC%20Blueprint%20White%20Paper%20V7.pdf.

[5] G. Ernst, J. Pfähler, G. Schellhorn, D. Haneberg, and W. Reif. 2015. KIV: Overview and VerifyThis competition. *International Journal on Software Tools for Technology Transfer.* vol. 17, no. 6, pp. 677–694.

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

[7] D. Weyns, N. Bencomo, R. Calinescu, J. Camara, C. Ghezzi, V. Grassi, L. Gurnske, P. Inverardi, J.-M. Jezequel, S. Malek, R. Mirandola, M. Mori, and G. Tamburrelli. 2017. Perpetual Assurances for Self-Adaptive Systems. In R. de Lemos, D. Garlan, C. Ghezzi, H. Giese (eds) *Software Engineering for Self-Adaptive Systems III: Assurances*. Lecture Notes in Computer Science, Springer, vol. 9640, pp. 31-63.

[8] N. Esfahani, and S. Malek. 2013. Uncertainty in Self-Adaptive Software Systems. In R. de Lemos, H. Giese, H. Müller, and M. Shaw (eds) *Software Engineering for Self-Adaptive Systems II*. Lecture Notes in Computer Science, Springer, vol. 7475, pp. 214-238.

[9] J. Zhang and B.H.C. Cheng. 2006. Model-Based Development of Dynamically Adaptive Software. *Proceedings of the 28th International Conference on Software Engineering*, pp. 371-380.

[10] B.H.C. Cheng, K.I. Eder, M. Gogolla, L. Grunske, M. Litoiu, H.A. Müller, P. Pelliccione, A. Perini, N.A. Qureshi, B. Rumpe, D. Schneider, F. Trollmann, and N.M. Villegas. 2014. Using Models at Runtime to Address Assurance for Self-adaptive Systems. In N. Bencomo, R. France, B.H.C. Cheng, U. Aßmann (eds.) *Models@run.time*. LNCS, vol. 8378, Springer, pp. 101–136.

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

[12] A. Bennaceur, R. France, G. Tamburrelli, T. Vogel, P.J. Mosterman, W. Cazzola, F.M. Costa, A. Pierantonio, M. Tichy, M, Aksit, P. Emmanuelson, and H. Gang. 2014. Mechanisms for Leveraging Models at Runtime in Self-adaptive Software. In N. Bencomo, R. France, B.H.C. Cheng, U. Aßmann (eds.) *Models@run.time*. LNCS, vol. 8378, Springer, pp. 101–136.

[13] E. Daubert, F. Fouquet, O. Barais, G. Nain, G. Sunye, J.M. Jezequel, J.L. Pazat, and B. Morin, 2012. A Models@runtime Framework for Designing and Managing Service-based Applications. *Proceedings of the 2012 Workshop on European Software Services and Systems Research—Results and Challenges (S-Cube)*, pp. 10–11.

[14] H. Loulou, S. Saudrais, H. Soubra, and C. Larouci. 2016. Adapting Security Policy at Runtime for Connected Autonomous Vehicles. *Proceedings of the 25th IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises*.

[15] A.M. Sharifloo, and A. Metzger. 2017. MCaaS: Model Checking in the Cloud for Assurances of Adaptive Systems. In R. de Lemos, D. Garlan, C. Ghezzi, H. Giese (eds) *Software Engineering for Self-Adaptive Systems III: Assurances*. Lecture Notes in Computer Science, Springer, vol. 9640, pp. 137-153.

[16] M. Trapp, and D. Schneider. 2014. Safety Assurance of Open Adaptive Systems – A Survey. In N. Bencomo, R. France, B.H.C. Cheng, U. Aßmann (eds) *Models@run.time*. Lecture Notes in Computer Science, Springer, vol. 8378, pp. 279-318.

[17] J. Zhang and B. H. C. Cheng. 2006. Using Temporal Logic to Specify Adaptive Program Semantics. *Journal of Systems and Software*, vol. 79, no. 10, pp.1361-1369.

[18] Z. Coker, D. Garlan, and C.L. Goues. 2015. SASS: Self-adaptation using Stochastic Search. *Proceedings 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems.*

[19] J. Cámara, D. Garlan, G. A. Moreno, and B. Schmerl. 2017. Analyzing Self-Adaptation Via Model Checking of Stochastic Games. In R. de Lemos, D. Garlan, C. Ghezzi, H. Giese (eds) *Software Engineering for Self-Adaptive Systems III: Assurances*. Lecture Notes in Computer Science, Springer, vol. 9640, pp. 154-187.

[20] A. Pandey, G. A. Moreno, J. Cámara, and D. Garlan. 2016. Hybrid Planning for Decision Making in Self-Adaptive Systems. In *Proceedings of the 10th IEEE International Conference on Self-Adaptive and Self-Organizing Systems*.

[21] S.W. Cheng, D. Garlan, and B. Schmerl. 2009. Evaluating the Effectiveness of the Rainbow Self-Adaptive System. In I*CSE 2009 Workshop on Software Engineering for Adaptive and Self-Managing Systems*.

[22] B. Schmerl, J. Anderson, T. Vogel, M.B. Cohen, C.M.F. Rubrica, Y. Brun, A. Gorla, F. Zambonelli, and L. Baresi. 2017. Challenges in Composing and Decomposing Assurances for Self-Adaptive Systems. In R. de Lemos, D. Garlan, C. Ghezzi, H. Giese (eds) *Software Engineering for Self-Adaptive Systems III: Assurances*. Lecture Notes in Computer Science, Springer, vol. 9640, pp. 64-89.

[23] M. Anisetti, C.A. Ardagna, and E. Damiani. 2015. A Test-Based Incremental Security Certification Scheme for Cloud-Based Systems. *Proceedings of the 2015 IEEE International Conference on Services Computing*, pp. 736-741.

[24] M. Anisetti, C.A. Ardagna, E. Damiani, and F. Saonara. 2013. A Test-based Security Certification Scheme for Web Services. A*CM Transactions on the Web*, vol. 7, no. 2, pp.1-41.

[25] M. Anisetti, C.A. Ardagna, F. Gaudenzi, and E. Damiani. 2016. A Certification Framework for Cloud-based Services. *Proceedings of the 31st Annual ACM Symposium on Applied Computing*.

[26] P. Stephanow, G. Srivastava, and J. Schütte. 2016. Test-based Cloud Service Certification of Opportunistic Providers. *Proceedings of the 8th IEEE International Conference on Cloud Computing*.

[27] P. Stephanow, and C. Banse. 2017. Evaluating the Performance of Continuous Test-based Cloud Service Certification. *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pp. 1117-1126.

[28] C.A. Ardagna, R. Asal, E. Damiani, N.E. Ioini, C. Pahl, and T. Dimitrakos. 2016. A Certification Technique for Cloud Security Adaptation. *Proceedings of the IEEE International Conference on Services Computing (SCC)*, pp. 324-331.

[29] C. Tsigkanos, L. Pasquale, C. Menghi, C. Ghezzi, and B. Nuseibeh. 2014. Engineering Topology Aware Adaptive Security: Preventing Requirements Violations at Runtime. *Proceedings of the 22nd International Requirements Engineering Conference*, pp. 203-212.

[30] E. Yuan, S. Malek, B. Schmerl, D. Garlan, and J. Gennari. 2013. Architecture-Based Self-Protecting Software Systems. Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures, pp. 33-42.

[31] B. Schmerl, J. Camara, J. Gennari, D. Garlan, P. Casanova, G.A. Moreno, T.J. Glazier, and J.M. Barnes. 2014. Architecture-based Self-

protection: Composing and Reasoning about Denial-of-Service Mitigations. *HotSoS*.

[32] M. Hale and R. Gamble. 2017. Semantic Hierarchies for Extracting, Modeling, and Connecting Compliance Requirements in Information Security Control Standards. *Requirements Engineering*, Dec., pp. 1-38.

[33] C.A.R. Hoare. 1985. *Communicating Sequential Processes*. Prentice Hall.

[34] K. Jensen, and L.M. Kristensen. 2009. *Colored Petri Nets: Modelling and Validation of Concurrent Systems*. Springer-Verlag.

[35] K. Jensen, and L.M. Kristensen. 2015. Colored Petri Nets: A graphical language for formal modeling and validation of concurrent systems. *Communications of the ACM*, vol. 58, no. 6, pp. 61-70.