100 Years of Software - Adapting Cyber-Physical Systems to the Changing World

Hayley Borck[1], Paul Kline[2], Hazel Shackleton[1], John Gohde[1], Steven Johnston[1], Perry Alexander[2], and Todd Carpenter[1]

[1] Adventium Labs,
111 3rd Ave S, Minneapolis, MN 55401
{first.last}@adventiumlabs.com
[2] The University of Kansas,
Information and Telecommunication Technology Center
2335 Irving Hill Rd, Lawrence, KS 66045
{paulkline,palexand}@ittc.ku.edu

Abstract. Cyber-Physical Systems (CPS) are software and hardware systems that interact with the physical environment. Many CPSs have useful lifetimes measured in decades. This leads to unique concerns regarding security and longevity of software designed for CPSs which are exacerbated by the need for CPSs to adapt to ecosystem changes if they are to remain functional over extended periods. In particular, the software in long-lifetime CPSs must adapt to unanticipated trends in environmental conditions, aging effects on mechanical systems, and component upgrades and modifications. This paper presents the Toolkit for Evolving Ecosystem Envelopes (TEEE) system created to help address these challenges in CPSs. TEEE is able to detect environmental changes which have caused errors within the CPS without directly sensing the environmental change. TEEE uses dynamic profiling to detect the errors within the CPS, determine the root cause of the error, alert the user, and suggest a possible adaption.

Keywords: Cyber-Physical Systems, Resilient Systems, Requirements-based testing

1 Introduction

Cyber-Physical Systems can interact with the physical environment by sensing external state, transferring kinetic and potential energy, computing solutions to affect desired outcomes, and driving electrical, optical, and mechanical actuators to achieve those outcomes. Unlike software applications, CPSs sense, depend upon, and actuate physical phenomena. The software in long-lifetime CPS must adapt to unanticipated changes in environments, mechanical, or use. The CPS, however, might not directly sense all aspects of its environment, especially those aspects of the environment which were not considered significant during original development. For example our System Under Test (SUT) is a specific patient controlled analgesia (PCA) pump which requires medical tubing with an inner

diameter of 0.054". However, residents of less developed countries are often forced to use whatever equipment is available to them, often without standard safety procedures or support resources. These users may have access to tubing with a smaller 0.0033" inner diameter which will affect the rate of flow of medication.

This paper presents the Toolkit for Evolving Ecosystem Envelops (TEEE) system to detect changes in the environment that are not directly sensible and semiautomatically adapt to them. Neches et al. [18] described resilient systems as: "trusted and effective out of the box in a wide range of contexts, easily adapted to many others through reconfiguration or replacement, with graceful and detectable degradation of function." TEEE aims to add this sort of resiliency to CPSs. Further, TEEE adds root cause analysis and adaption to errors, whether they are expected (i.e., degradation due to longevity in the field) or unexpected. TEEE uses dynamic profiling tools and techniques to explore CPS performance envelopes, subject to its evolving environment, that will ultimately allow software to adapt as internal and external conditions change. TEEE leverages model-based development techniques for requirements, design, architecture, configuration, and automated measurement and stimulus to identify root causes of anomalies. In contrast, the state of the practice development processes still largely use trial-and-error test-based software coding.

The remainder of this paper is structured as follows. Section 2 presents the TEEE system design and architecture. Section 3 describes the background and related research.f Section 4 describes how TEEE models the CPS system. Section 5 and Section 6 go into detail on the Synthesis of Stimulus and Measurements algorithms respectively. Section 7 presents a real world use case and the results of running it through TEEE. Finally, Section 8 concludes the paper.

2 TEEE Overview

When an error in the system is detected, currently by the user, the TEEE system uses CPS models and design to create and inject profiling code to identify the root cause of the error. The aim of the Synthesize Stimulus Algorithm (SSA) and Dynamic Measurements component is to infer the root cause of the error, especially in cases which the error is not directly sensible by the CPS. When a root cause is determined alternative system hardware or software components (i.e., motor or motor controller software) are suggested to the user.

The primary components of the TEEE architecture are AADL models of the SUT and dynamic profiling components to synthesize measurements and stimulus of the SUT. The current prototype, developed in JAVA and Coq [4], has all of the components built with manual data transfers. TEEE interfaces with the developer (or trained user) before the SUT is deployed. During this step (shown by the circled 1, in Figure 1) the developer indicates which AADL model will be deployed as the SUT. For example the user would indicate the specific implementation of the system motor controller. The SUT is constantly monitored by the user for errors, a process we intend to automate in the future. If a variable in the system has different values than the requirements specify (for example flow

rate on the medical tubing does not fall within a specified range in the requirement) the user indicates to TEEE that an error occurred. Dynamic profiling code is injected into the system using the TEEE CPS Synthesized Stimulus and Dynamic Measurement synthesis tools (circled 2). TEEE generates synthesized stimuli, driven by requirements in the model of the SUT, using the Synthesize Stimulus component. The stimuli drives exploration of the overall operational envelopes of the SUT. Operational envelopes are regions in which the CPS is intended to correctly operate as per its requirements. For the PCA pump SUT, an example envelope might include a space defined by flow rate, environmental temperature, and fluid viscosity. The stimuli can also be used to focus on specific cyberphysical characteristics to evaluate, with input from the user. For example, the user may specify prioritization of stimuli on a certain component (i.e., tubing, motor, sensors etc...). The Synthesize Stimulus component explores operational envelopes by creating a test case suite from requirements. A potential drawback of the current TEEE implementation is the manual process of creating requirements for the SUT; If a requirement is missing in the model there will be no test case created. The user of the SUT is tasked to test the SUT according to the test cases within the suite. Information on the operational envelopes is sent to the Dynamic Measurement component. The Dynamic Measurement component synthesizes measurements, consists of properties about the SUT, and reasons with the architecture models to infer system behaviors. The results of the Dynamic Measurement system is a set of components from which the error may have originated. In Section 7 we will dive into an example of TEEE doing exactly this in a real world scenario.

3 Related Work

Typical design-for-test and unit-test approaches evaluate the SUT against requirements, but these methods only address a small fraction of issues, with the majority of defects actually arising from requirements [16]. As such, several approaches use a SUT model and/or requirements to detect errors and prioritize test cases.

Rodriguez et. al. [22], model the security and specifically the resilience of systems in Unified Modeling Language (UML) models. Their analysis and modeling of security requirements exposes the underlying relationship between security and dependability. Similarly, TEEE uses the dynamic profiling components (Sections 5 and 6) to uncover constraints in the system including security requirements. Rugina et. al. [23], present a framework for modeling dependability using the Architecture Analysis and Design Language (AADL) [8], [7] and Generalized Stochastic Petri Nets (GSPNs). In their framework an error model is added to the AADL architecture model to present a full picture of the dependability for the user. Their framework is used to determine the reliability, availability, and safety prior to system deployment. TEEE focuses on determining if the requirements, including these dependability properties, are satisfied in the event of an environment change or off-specification use when the system has been deployed.

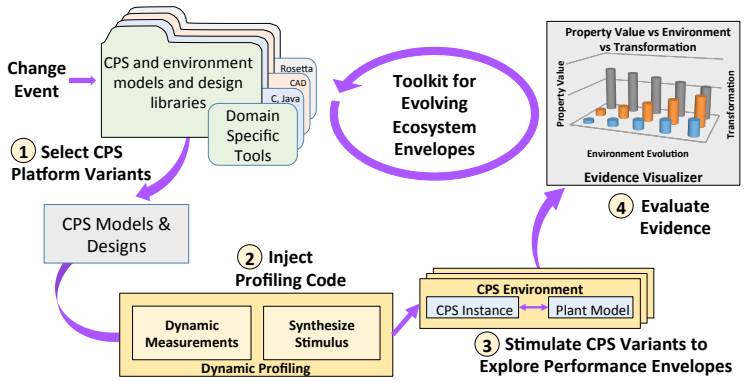

Fig. 1. The TEEE system architecture.

Arafeen and Do [2] use requirements to prioritize test cases and more quickly determine faults. Their prioritization scheme clusters the requirements and prioritizes the cluster based on the priority of the requirements within. TEEE's test case prioritization scheme (Section 5) also takes uses system requirements to create and prioritize test cases. However, TEEE also takes into account whether the test case (and subsequently requirement) has previously exposed an error. The merging of these prioritization techniques may prove interesting and will be explored in further work. Dreossi et. el. [6] detect errors in machine learning components of CPS systems, such as in Lane Keeping Assist Systems in cars, by formulating it as a falsification problem for the model. TEEE similarly uses the model requirements to create test cases and determine errors within the CPS.

Adaption in systems (CPS or software) research is focused primarily on automatically creating patches for software. The GenProg system, Le Goues et. al., [14], uses genetic programming to automatically repair software defects given a set of test cases. The ClearView system [19] automatically patches errors in deployed software without access to source code or debugging info. ClearView learns normal execution, detects failures while monitoring execution, and generates a patch. While ClearView works on deployed systems, as TEEE does, it discovers errors by learning 'normal' execution and would be unable to discover error if the 'normal' execution changes (such as a system use case change). Converse to these software only approaches TEEE is able to find and repair issues stemming from the underlying architecture (with a human in the loop) as well as software errors. TEEE models alternate components in the CPS architec-

ture and, when an issue arises, is able to suggest possible alternate architecture configurations.

The TEEE project is a seedling effort to augment the DARPA Building Resource Adaptive Software Systems (BRASS) program [10], which is tasked with creating resilient systems that have robust and functional 100+ year software. This program has roots in autonomic computing [12] in which systems manage themselves given high-level objectives. TEEE only tries to monitor the system for errors in order to determine error causes and possible adaptations however, rather than the larger task of managing goals and objectives of the system administrator. Part of ensuring resilient long lifetime software includes accounting for unanticipated uses of systems as well as unintended environmental changes. The TEEE approach uses dynamic profiling components to determine whether environmental changes and/or changes to the SUT use cases are the cause of current errors. Stoicescu et. al. [24] expanded upon Neches description of resilient systems to be "expected to continuously provide trustworthy services despite changes in the environment or in the requirements they must comply with." The authors outlined an overall approach to defining fault tolerant applications that automatically adapt during the systems lifetime. Their approach monitors the system and analyzes the observations to determine if adaptation is necessary. Stoicescu et. al. and TEEE share the goal of adapting to changes in requirements and/or the environment. Adjepon-Yamoah [1] modeled fault tolerant methods via petri nets in systems interfacing with unpredictable environments (i.e., the cloud). Similarly, TEEE interfaces with the highly unpredictable physical world to evaluate the cause of errors in the SUT.

4 Modeling Cyber-Physical Systems

The SUT used with the TEEE prototype is a PCA pump. The PCA pump's components and requirements are modeled in AADL. While our current SUT is a PCA pump there is no reason TEEE cannot be generalized to other CPSs, as long as the models are given to the system.AADL was chosen due to it's focus on *architecture* rather than the functional/behavior emphasis that underlies other modeling languages. In particular, it better enables modeling and trading-off *what* components comprise a system and the relationships between the components, rather than *how* the system works. AADL has been shown beneficial to risk management activities using medical devices [13]. One of the salient features of AADL is the ability to model design alternatives coherently within a single AADL model. AADL defines component types that include all externally visible features, separately from implementations, which model component internals. Component `implementations`, an instantiation of a component, may have subcomponents which themselves may be component types or implementations. A component type may have any number of implementations, all of which look identical from outside. By having multiple implementations for a component, different design alternatives can be modeled. This allows many alternatives for fault management to be captured in a single model so they may be evaluated and compared. We anticipate over the lifetime of the CPS additional alternative

implementations and components will be added to the model as technology advances. Lastly AADL is a rich enough language and does not require extensions to model CPSs. Requirements are scraped from the AADL model of the CPS system by a custom OSATE [11] plugin. The requirements are consumed (via XML) by the Stimulus Synthesis Algorithm (SSA) (Section 5) and Dynamic Measurement algorithm (Section 6). Listing 1.1 shows a snippet of one implementation of the motor component in the PCA pump. In this snippet the specific motor modeled is called 'motor', its parents are defined under the `<Parents>` tag. The criticality of the component is defined by the user and annotated with the `<Criticality>` tag. Lastly the requirements of the component are defined using the `<Variable>` tag. Each variable may define an allowable and test range as well as the actual value. Often the actual functioning range of a variable will be larger than the allowed range indicates, which is why we include the option of a test range. The requirement on the motor component in Listing 1.1 defines the variable *Operating Temperature* as having an allowed range of -10 to 40 degrees Celsius.

Listing 1.1. A XML requirement on the motor component of the PCA pump that has been extracted from the AADL model.

```
<Component type="device" implementation="motor">
  <Parents>
    <SystemRef type="system" implementation="motorSystem"/>
    <SystemRef type="system" implementation="pump"/>
    <SystemRef type="system" implementation="Full_sys_inst"/>
  </Parents>
  <Criticality>0</Criticality>
  <Variable name="OperatingTemperature" units="c">
     <allowed>
          <real min="-10.0" max="40.0"/>
     </allowed>
  </Variable>
</Component>
```

5 Stimulus Synthesis Algorithm

The Stimulus Synthesis Algorithm (SSA) probes the SUT operating envelope by creating a set of test cases from the model requirements. The SSA is a combination of state of the art approaches which are described further in this section. The SSA consists of two sub-algorithms 1) Create all test cases from the system specifications and requirements, 2) Reduce test cases to N-wise subsets where possible, and prioritize the test cases. The results are sent to the Dynamic Measurements component.

5.1 Create test cases from requirements

For each component in the model, the SSA creates a test case that corresponds to each variable's allowable range and test range. Our algorithm to create test cases

from requirements is derived from Ranganathan's [21] work using the Rosetta modeling language. A *test case* is defined in our work as a *test scenario*, a boolean condition to be applied to a variable; and a *test vector*, a set of inputs to be substituted for the variable in the boolean condition. The system requirements for the motor component (Listing 1.1) only define one variable with an allowable range, therefore, one test case will be created. The test scenario is the boolean condition: $-10 \leq temp \ \wedge \ temp \leq 40$. This example test case will test if a particular component in the CPSs, the motor, is operating under the temperature range it for which it was designed. The test vector for each test case is created using the step value in the requirement. If there is no step value present in the model, a step value of the nearest 1 at the lowest non-zero decimal place is used (i.e., 200 has a step = 100, 0.34 has a step = 0.01). We expect the AADL model to be hand created by system designers and therefore, have all of the necessary information such as step value. However, in the case of a legacy model or if a designer does not know the step value we have implemented a rest step creation algorithm. A test vector is created by the SSA for the operating temperature variable by enumerating each value between -10 and 40 with a step of 10 $(-10, 0...30, 40)$. Boundary values have been implicated in faults within the SUT [17], therefore an additional n, where $n = 2$ in the current prototype, vector values are added on each boundary. The SSA also adds test vector values for the actual variable value, if available. The resulting test case suite has sufficient coverage over the specified requirements.

5.2 Combine and Prioritize

To reduce the number of test cases and subsequently the time it takes to test the SUT, the SSA combines test cases using the method by Lott et. al. [15]. As previously mentioned a test case is created for each variables allowable range and test range. The large number of test cases is not scalable to large CPSs which is why we combine the test cases. The combination algorithm is a simple greedy algorithm described by Cohen et. el. [5], which combines test cases into pairwise randomly until there are none (or only one) left to combine. The SSA does not pair test cases which test the same variable (i.e., temperature) in the test scenario. We found, as Lott et. al. did, that a higher order combination yields greater test pattern savings. Though currently the SSA algorithm uses pairwise combination to reduce the risk of combining differently named variables which are actually the same (i.e., operating temperature vs temperature). With pairwise combination, assuming independence, growth of the test space increases $log_2(x)$ where x is the number of independent requirements. Increasing the order of combination of test cases, changes it to $log_n(x)$.

The test suite is prioritized to find failure quickly using a the fault-recorded test prioritization (FRTP) technique [20]. At this stage the user may request prioritization on a specific component. Each time a test case is marked as failed, its Failure Detection Number (FDN) is incremented. This indicates a fault has been found at the component(s) being tested within the test case. The FRTP method iteratively extracts information from the testing process and does not need to be

bootstrapped with information from prior test executions. The FRTP method prioritizes test cases based on previously found faults (FDN). Some components are necessarily 'more important' than others. For example, if the motor in the PCA pump fails then the PCA pump will not work. If instead a sensor on the motor fails then the PCA pump will error but may continue to work. To encode this we added the criticality of the component to the prioritization algorithm by using an equation derived from the *Risk Exposure* metric [3] to prioritize the test suite.

$$RiskExposure(TS) = \frac{\sum_{tc \in TC} P(f) * C(f)}{|f|} \tag{1}$$

Chen et. al., defines the risk exposure metric (Eq. 1) as the probability of failure ($P(f)$) of a component in the current test case tc multiplied by the cost of failure of the components in the current test case ($C(f)$) and then divided by the total number of components in the current test case. In place of determining the probability of failure for each component in the test case we redefined $P(f)$ in TEEE to represent the number of times the components in the current test case previously failed any test case. Equation 2 shows the TEEE definition of $P(f)$ which is a novel extension of the Chen Risk Exposure metric. In TEEE the $P(f)$ is defined by the sum of the FDN for each component in the current test case over the entire test suite (denoted by TS). The cost of failure ($C(f)$), or criticality of a component, is annotated by the user in the AADL model (`<Criticality>` tag). The default criticality is to zero, which means not critical. In future iterations we plan to explore ways of automatically inferring criticality to give the SSA more meaningful and complete information to reason on.

$$P(f) = \sum_{tc \in TS} \left(\sum_{c \in tc} FDN(c) \right) \tag{2}$$

Finally Grindal et. al. [9] looked at the effectiveness of test case combination and found better results when pair-wise test cases are combined with a single variable test strategy. The SSA's final step is to randomly add k one-wise test cases to the test suite from the pre-combined list of one-wise test cases for the SUT. We choose a random k between 25% and 75% of the test suite size to test the prototype.

Figure 5.2 shows the prototype GUI for the SSA algorithm. The requirements file for the PCA pump has been loaded and the SSA algorithm has been run in the figure. The left side of the GUI shows statistics on the number of test cases created and the number of *test patterns* (the test scenarios from the test case and one test vector value from each test case) before and after combination. It is worth noting that combining the test cases into pair-wise test cases creates a test pattern savings of 24% for the PCA pump. The right side of the GUI shows a pairwise test case. The test scenarios test the tube component (top) and power system component (bottom). The user is requested to test the test vector values highlighted in red by substituting the vector values for their respective variable. In this example the variables are length for the tube component and power for

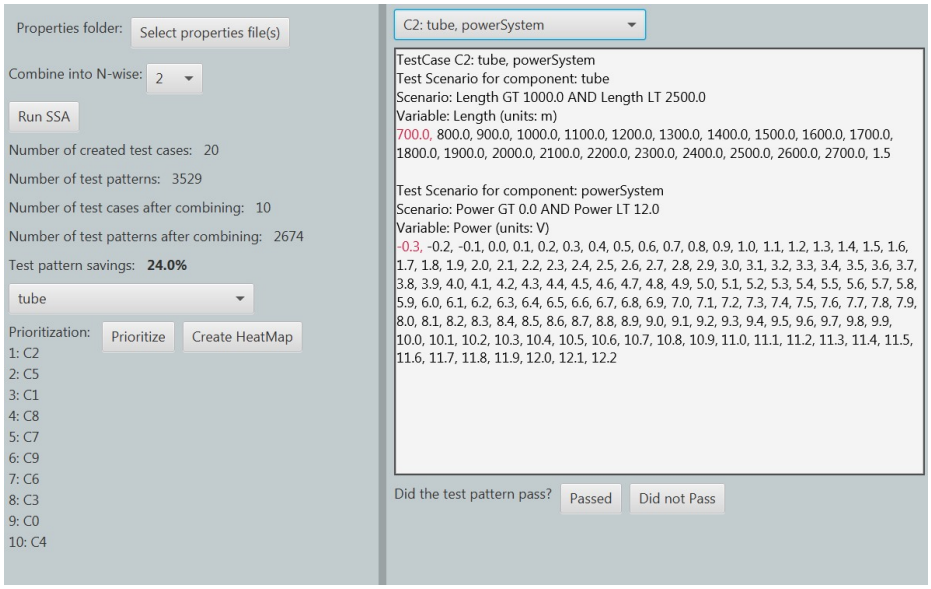

Fig. 2. A GUI of the SSA algorithm showing a combined test case testing a tube and motor sensor

the power system. The user is expected to record the results using the 'passed' or 'did not pass' buttons.

6 Dynamic Measurements

Some of the properties in CPSs necessary for engineering design decisions or operational decisions are not directly sensed by the SUT. To calculate the measurements of these properties TEEE uses dynamic measurements of properties which can be sensed. By synthesizing these measurements TEEE alleviates the need to measure properties directly in the environment (i.e., the user may not need to buy new sensors for the SUT in order to determine newly encountered errors). For example, flow rate is a critical property both the user and designer need for their respective tasks. The most obvious way to determine flow rate is to sense it. However, in our working example (PCA pump) flow rate sensors are not usually built into the system. This forces us to calculate flow rate from other known quantities. Currently the PCA pump calculates flow rate from system parameters input by the user. While this has some utility, it is an indication of what the flow rate *should* be and not what it actually is. For both engineering and use case scenarios determining the actual delivered flow rate is critical.

TEEE constructs measurements which are not directly sensible by the SUT using dynamic, physical measurements from other properties. The measurement calculation is performed much as it currently is, but using physical measurements

rather than exclusively user input. For example, flow rate can be calculated by taking the following measurements: motor speed, distance (meters) plunger traveled per motor rotation, and vial diameter and then calculating a flow rate value:

$$\texttt{flowRate} := \texttt{metersPerRevolution} * \texttt{motorSpeed} * (\pi * (\texttt{tubeDiameter}/2)^2)$$

Additionally, we would like mathematical evidence to provide further confidence in the calculated value. While flow rate is a simple calculation, when we begin to explore more complex properties mathematical assurance is essential. To achieve these goals we construct and verify a formal model of the calculation and measurements and synthesize a protocol from the calculation. Verification assures correctness properties hold and synthesis assures resulting code faithfully implements the calculation. We have chosen the proof assistant Coq [4] for our modeling, verification and synthesis tasks. Coq's design as a verification and synthesis language for software and its proof programming capabilities make it ideal for our purposes.

Coq provides a dependent type checking capability that can establish a diverse set of properties well beyond what is traditionally viewed as type checking. To provide a degree of assurance in our high-level property specifications we used Coq's dependent type system to implement units analysis. Similar to techniques taught in basic math and science classes, this technique ensures that units involved in calculations are compatible. When they are not, expressions will not type check and thus cannot be used in any computation. Thus, units analysis provides a simple static analysis that predicts errors prior to processing.

Every measurable quantity in our engineering domain is expressible by some combination of the seven base units (Ampere, Candela, Kelvin, Kilogram, Meter, Mole, and Second). For example, a Newton is $\frac{Kgm}{s^2}$, and a Volt is $\frac{Kgm^2}{s^3A}$. To our surprise, we could find no existing Coq library for keeping track of units. Therefore, we created a Units library and a dependently typed expression language implementing Units. With these libraries, we can create a *typed expression* where the simplification of the subterms are guaranteed to evaluate to the stated units of the expression itself. If the units do not match, the statement cannot be constructed.

We know that the end result of our flow rate calculation has units or type $\frac{m^3}{s}$. Thus, any calculation of flow rate must result in that type. The following Coq pseudo-code calculates flow rate using the previous equation with units:

```
var flowrate :: m^3/s := (metersPerRevolution :: m/Void)
                       * (motorSpeed :: Void/s)
                       * (3.14 * (tubeDiameter :: m / 2)^2)
```

During type checking Coq examines the types of various quantities with associated units and determines compatibility. The type of the tube cross section area is m^2 and is calculated by squaring the `tubeDiameter` variable of type m. The Meters Per Revolution (MPR) type is $\frac{m}{Void}$, meters divided by a unit-less number. When multiplying the MPR by motor speed of type $\frac{Void}{s}$ the $Void$

values cancel giving $\frac{m}{s}$. Finally when multiplying the result by tube cross section results in $\frac{m^3}{s}$, the unit associated with flow rate. One cannot make a correctness assertion of the tube diameter based on this result, but it is evidence that the formula is correct.

In addition to properties which are not directly sensible, but may be calculated using other other properties measurements, some properties may not be calculated using properties, or may not be directly measurable from the operational environment. For example, the distance traveled by the plunger per motor revolution is not easily measurable in our SUT because the gear train is sealed preventing counting teeth or relying on them all being the same. The value is also not likely to change without severe modification and abnormal use of the system. However, the value may be derivable if we are able to determine flow rate from more than one method. The differing values are detected, and we can deduce what environmental factors may have changed to explain the discrepancy. Therefore it may be possible to adjust predefined assumption values as needed. The assumed or given value for distance traveled is identified in the SUT AADL model.

To reason about the measurement process we must have a model of the pump's operational `Environment`. To model this environment in Coq we create a class containing measurable quantities. The instance of this class must have every possible measurement enumerated and defined as either an assumption or a measurable value. Assumptions and their assumed values are provided in the AADL model. Measurable values must define how the value is measured. Additionally, a proof must be provided to confirm every measurement is present in exactly one of these two categories. When the measurement code is synthesized from the Coq model, the environment model falls away and is replaced by the actual environment.

7 Scenario Walkthrough

CPSs developed for first world countries are retired to developing countries after their service life expires in the first world countries. In these situations resources are not always available to run these systems in the environment they are designed for. We will validate two scenarios which came from real world observations of PCA pumps being used in developing countries. Then we will walk through one of the scenarios, showing the output of each of the TEEE components, and demonstrate TEEE is able to determine possible root causes and suggestion an adaption. As this is a unique system a full system evaluation was not able to be run, however, we show through the scenario walkthrough the validity of the system. First we will look at the viscosity of the material being pumped. Untrained or overworked users may put the wrong medication into the pump. While there is a bar code reader on the PCA pump, it is easily bypassed. Additionally, temperature has an affect on the viscosity of liquids. Egg whites are similar in viscosity to blood plasma, which is commonly used to treat patients with shock. If, for example, the PCA pump is used in an area which is very hot

and without air conditioning the medication could be more viscous. In the walk through we will show that TEEE is able to determine the root cause of this error is the viscosity of the medication in the pump. A second common issue found is how a brown out may affect the SUT. Brown outs can cause the motor to run slower and subsequently the amount of medication expelled is less. To continue regular functionality, many medical devices contain a battery. However, very few working batteries last or even arrive in the developing world, and black/brown outs can have a significant effect on the device that the programmers thought would never occur[3]. The motor, in this particular PCA pump, does not have a sensor to determine the motor rate nor does it have a sensor to determine the flow of the material being pumped. If a brown out occurs there could be no way for the CPS software to determine the cause of the problem.

We ran experiments to confirm the validity of these scenarios. Water and egg whites were run through the PCA pump for 5 minutes and varied the speed of the motor (100Hz, 50Hz, and 25Hz). We ran 5 experiments for each variant. The experiments showed a significant difference, using a paired t Test with $p < .005$, between uL expelled per tick of the motor between 100Hz and 25Hz as well as 50Hz and 25Hz when using egg whites. Water showed a significant difference between 100Hz and 50Hz as well as 100Hz and 25Hz. The t Test resulted in a value of $p = 0.007$ when comparing 50Hz and 25Hz using water. The results of this experiment can be seen in Figure 3. The test results also indicated a significant difference in uL expelled per tick of water versus egg whites at 100Hz and 25Hz ($p < .005$). These experiments confirm the validity of the brownout scenario by showing the rate of the motor affects the amount of material dispensed. They also confirm the validity of the viscosity scenario showing materials at different viscosities affect the amount of material dispensed.

Confirmation of adaptation within TEEE The data from the PCA pump experiments shows that there is a difference in amount of material expelled when using materials of difference viscosities. Viscosity of the medication in the PCA pump, however, is a change within the environment that cannot be known via it's sensors. To confirm that TEEE is able to determine the root cause of this scenario (material is of a different viscosity than is expected) and adapt to such changes we will dive into the output of each component. The first step is to model the PCA pump in AADL.

Listing 1.2. A snippet showing requirements in the viscosity scenario.

```
<Component type="device" implementation="tube">
 <Variable name="FlowRate" units="ulps" varType="real">
```

[3] In one of the authors person experience, we once came across some donated defibrillators none of which had batteries. While the defibrillators are designed to still function without a battery (slightly slower charge build up), they were clearly never intended to be used this way as one of steps in the daily self test *required* the presence of a battery despite the battery itself not being present in the test. Luckily, we were able to find an alternate method of ensuring proper functionality.

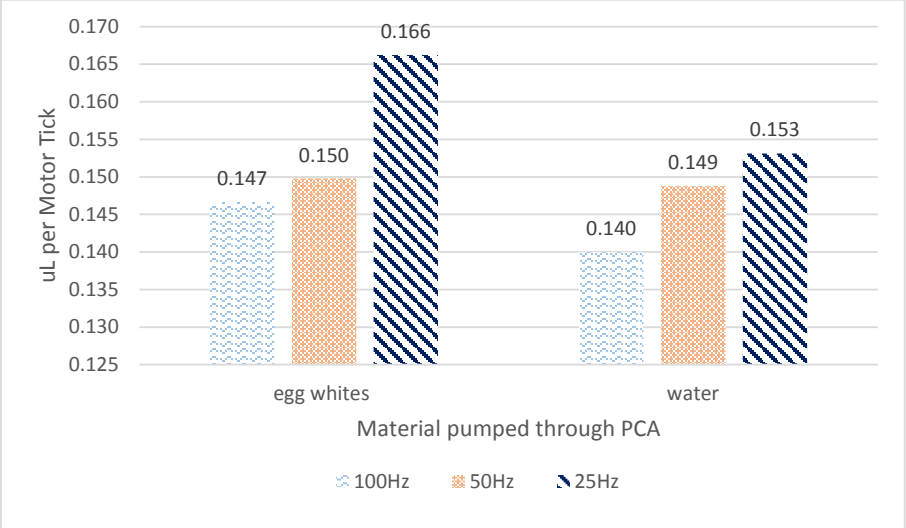

Fig. 3. Comparison of the uL of material (egg whites or water) expelled from the PCA pump when the motor was running at 100Hz, 50Hz, and 25Hz.

```
    <allowed> <real min="0.141" max="0.147"/> </allowed>
 </Variable>
<\Component>

<Component type="device" implementation="medication">
 <Variable name="DynamicViscosity" units="cP" varType="real">
    <allowed> <real min="1" max="1.5"/> </allowed>
 </Variable>
<\Component>
```

A requirement is put on the tube component of the model that the flow rate of the medication must be between 0.141 and 0.147 and viscosity of the medication must be between 1 and 1.5. (Listing 1.2). The SSA algorithm created 20 test cases from the requirements within the model which enumerated 3529 test patterns (the test scenarios of the test case and one value from each of the test vectors). After the SSA pair-wise combination step is run the test case suite size is reduced to 10 cases and 2674 test patterns, yielding a test pattern savings of 24% (results shown in the SSA GUI in Figure 2. The test cases in Table 1 corresponds to the requirements.

A randomized user was simulated testing the PCA pump, i.e., running through the test cases and marking them passed or failed. Each test pattern had a 50% chance to mark its parent test case as failed, except the test case for the Tube component, shown in Table 1, which was marked failed each time. The test case

Component	Test Scenario	Test Vector	Actual Value
Tube	$0.141 < FlowRate < 0.147$	0.139, 0.140, 0.141, 0.142, 0.143, 0.144, 0.145, 0.146, 0.147, 0.148, 0.149, 0.166	0.166
Medication	$1.0 < Viscosity < 1.5$	0.8, 0.9, 0.94, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7	0.94

Table 1. Test cases created for requirements on tube flow rate and medication viscosity

suite was then prioritized on the tube component. The resulting prioritization along with failure detection number and risk exposure score is found in Table 2.

Case Id	**Component A**	**Component B**	**FDN**	**Risk Exposure**
C5	tube	medication	2070	.60
C2	tube	power system	144	.35
C8	interface logic system	tube	44	.17
C1	motor	power system	114	.09
C7	pump	power system	120	.08
C9	environment	power system	110	.06
C4	pump	pump	12	.04
C0	pump	interface logic system	46	.02
C3	motor sensor	motor controller	18	.01
C6	motor controller	motor sensor	30	.01

Table 2. The prioritization of test cases for the tube component based on randomized user data.

The information on the test case failures was sent to the Dynamic Measurement component to provide more information concerning the cause of the error. The Dynamic Measurement component models the calculation of mass flow rate as described previously. Working from measured values back to flow rate provides an alternative perspective on the failure. The mass flow rate equation is defined using Coq and verified using units analysis and using an execution semantics for the protocol description. Using information from testing and measurement, the user is able to determine the failure is likely that the medication is the incorrect viscosity rather than the alternative of improper tube diameter. With the root cause of the failure found a recommendation is presented to the user to change the viscosity of the medication based on evidence from the SSA and Dynamic Measurement system. A new test suite is set up and tested to confirm the issue was solved.

8 Conclusion

In this paper we presented the Toolkit for Evolving Ecosystem (TEEE) system, to address challenges in CPSs due to changing environment or use over time. We presented a real world example of environmental changes affecting the use of a PCA pump. The scenario was verified valid by a series of experiments using a Hospira PCA pump. We showed the TEEE prototype is able to determine the root cause of the issue in the scenario using the Stimulus Synthesis and Dynamic Measurements algorithms. Further work will focus on automating the components of TEEE. The SSA creates a bottleneck by requiring a human in the loop to manually mark test cases as passed/failed. In the future we plan to create tools using OSATE-based analysis to determine if a test case will pass/fail. Future work on the Dynamic Measurement algorithm will focus on deducing more complex or obscured environmental changes, such as vial diameter or faulty sensors. To do this we will create a number of verified measuring programs for each property within the AADL model. This will allow the algorithm to dynamically answer requests like "measure flow rate every possible way and compare the results". We are also aiming to create a Dynamic Measurement algorithm which is able to determine the property measurement without using assumptions in the current environment.

Acknowledgments. This material is based upon work supported by the United States Air Force and DARPA under Contract No. FA8750-16-C-0273. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Air Force or DARPA.

References

1. Adjepon-Yamoah, D.E.: cloud-atam: Method for analysing resilient attributes of cloud-based architectures. In: International Workshop on Software Engineering for Resilient Systems. pp. 105–114. Springer (2016)
2. Arafeen, M.J., Do, H.: Test case prioritization using requirements-based clustering. In: Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on. pp. 312–321. IEEE (2013)
3. Chen, Y., Probert, R.L., Sims, D.P.: Specification-based regression test selection with risk analysis. In: Proceedings of the 2002 conference of the Centre for Advanced Studies on Collaborative research. p. 1. IBM Press (2002)
4. Chlipala, A.: Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant. MIT Press (2013)
5. Cohen, D.M., Dalal, S.R., Fredman, M.L., Patton, G.C.: The aetg system: An approach to testing based on combinatorial design. IEEE Transactions on Software Engineering 23(7), 437–444 (1997)
6. Dreossi, T., Donzé, A., Seshia, S.A.: Compositional falsification of cyber-physical systems with machine learning components. In: NASA Formal Methods Symposium. pp. 357–372. Springer (2017)

7. Feiler, P., Lewis, B., Vestal, S.: The SAE Avionics Architecture Description Language (AADL) Standard: A Basis for Model-Based Architecture-Driven Embedded Systems. In: Real-Time Applications Symposium Workshop on Model-Driven Embedded Systems (2003)
8. Feiler, P.H., Gluch, D.P., Hudak, J.J.: The architecture analysis & design language (aadl): An introduction. Tech. rep., DTIC Document (2006)
9. Grindal, M., Lindström, B., Offutt, J., Andler, S.F.: An evaluation of combination strategies for test case selection. Empirical Software Engineering 11(4), 583–611 (2006)
10. Hughes, J., Sparks, C., Stoughton, A., Parikh, R., Reuther, A., Jagannathan, S.: Building resource adaptive software systems (brass): Objectives and system evaluation. ACM SIGSOFT Software Engineering Notes 41(1), 1–2 (2016)
11. Institute, S.E.: Open source aadl tool environment (osate). http://la.sei.cmu.edu/aadlinfosite/OpenSourceAADLToolEnvironment.html
12. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer 36(1), 41–50 (2003)
13. Larson, B., Hatcliff, J., Fowler, K., Delange, J.: Illustrating the aadl error modeling annex (v. 2) using a simple safety-critical medical device. ACM SIGAda Ada Letters 33(3), 65–84 (2013)
14. Le Goues, C., Nguyen, T., Forrest, S., Weimer, W.: Genprog: A generic method for automatic software repair. IEEE Transactions on Software Engineering 38(1), 54–72 (2012)
15. Lott, C., Jain, A., Dalal, S.: Modeling requirements for combinatorial software testing. In: ACM SIGSOFT Software Engineering Notes. vol. 30, pp. 1–7. ACM (2005)
16. Mogyorodi, G.: What is requirements-based testing? Tech. rep., Crosstalk (2003)
17. Myers, G.J., Sandler, C., Badgett, T.: The art of software testing. John Wiley & Sons (2011)
18. Neches, R.: Engineered resilient systems (ers) s&t priority description and roadmap (2011)
19. Perkins, J.H., Kim, S., Larsen, S., Amarasinghe, S., Bachrach, J., Carbin, M., Pacheco, C., Sherwood, F., Sidiroglou, S., Sullivan, G., et al.: Automatically patching errors in deployed software. In: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. pp. 87–102. ACM (2009)
20. Qi, Y., Mao, X., Lei, Y.: Efficient automated program repair through fault-recorded testing prioritization. In: Software Maintenance (ICSM), 2013 29th IEEE International Conference on. pp. 180–189. IEEE (2013)
21. Ranganathan, K., Rangarajan, M., Alexander, P., Regan, T.: Automated test vector generation from rosetta requirements. In: VHDL International Users Forum Fall Workshop, 2000. Proceedings. pp. 51–58. IEEE (2000)
22. Rodríguez, R.J., Merseguer, J., Bernardi, S.: Modelling and analysing resilience as a security issue within uml. In: Proceedings of the 2nd international workshop on software engineering for resilient systems. pp. 42–51. ACM (2010)
23. Rugina, A.E., Kanoun, K., Kaâniche, M.: A system dependability modeling framework using aadl and gspns. In: Architecting Dependable Systems IV, pp. 14–38. Springer (2007)
24. Stoicescu, M., Fabre, J.C., Roy, M.: Architecting resilient computing systems: overall approach and open issues. In: International Workshop on Software Engineering for Resilient Systems. pp. 48–62. Springer (2011)