A Platform for Run-time Health Verification of Elastic Cyber-physical Systems

Daniel Moldovan, Hong-Linh Truong
Distributed Systems Group, TU Wien, Vienna, Austria
E-mail: {d.moldovan,truong}@dsg.tuwien.ac.at

***Abstract*—Cyber-physical Systems (CPS) have components deployed both in the physical world, and in computing environments, such as smart buildings or factories. *Elastic Cyber-physical Systems (eCPS)* are adaptable CPS capable of aligning their resources, cost, and quality to varying demand. eCPS have started to generate interest in various domains due to their adaptability, such as *Industrie 4.0*. In Industrie 4.0 they can link manufacturing processes with private or public cloud services, and adapt to varying usage patterns and requirements. However, industrial eCPS are mission critical systems designed with strict requirements. Using cloud services increases the complexity of eCPS and introduces particular challenges in ensuring their run-time health. Failures can appear at appear at cloud provider from the virtualization middleware, physical resources, or software configuration. eCPS failures can also occur due to management operations, software bugs, or resources congestion. While static verification methods can determine failure sources, they are less applicable to eCPS infrastructures. eCPS can have complex hardware and software stacks, and might use third-party black-box components(e.g., sensors, cloud services). To this end, a new approach to run-time health verification is required to ensure eCPS continue to fulfill their operating requirements during their lifetime. In this paper we introduce an approach and supporting platform for verifying at run-time if the components of elastic cyber-physical systems are: (i) deployed and running, (ii) correctly configured, and (iii) provide expected performance. We evaluate our platform on an eCPS for analysis of streaming data from smart environments.**

***Keywords*-elastic system, run-time verification, cyber-physical, health**

I. Introduction

A *Cyber-physical System (CPS)* is a system which has components deployed both in the physical world (e.g., industrial machines, smart buildings), and in computing environments (e.g., data centers, cloud infrastructures)[1]. For example, a smart factory could be considered as a CPS having components: (i) inside assembly robots, (ii) inside sensor gateways deployed in the factory to collect environmental conditions, and (iii) deployed in a private data-center to analyze data collected from robots and sensor gateways. An *Elastic Cyber-physical Systems (eCPS)* can further add/remove components at run-time, from computing resources to physical devices. Elasticity enables eCPS to align their costs, quality, and resource usage to load and owner requirements.

eCPS have started to generate interest in various domains due to their adaptability. Such a domain is *Industrie 4.0*[1], in which industrial enterprises can build complex elastic cyber-physical systems linking their manufacturing processes with private or public cloud services. Industrial eCPS would enable manufacturing processes to adapt to varying usage patterns and requirements. However, industrial systems are usually mission critical, designed with strict requirements. Combining them with cloud and elasticity introduces particular challenges and problems which need to be addressed for realizing industry-level eCPS. In general, managing systems using cloud services requires a lot of effort[2]. First, there is almost certain that failures at the cloud provider end will occur during the system lifetime [3], [4], [5]. Failures can originate in the cloud hardware layer, from physical resources such as servers, storage, or network elements [6], [7]. A second source of failures is the virtualization middleware used in the eCPS, which can either fail itself due to internal bugs, or can cause the failure of the software running on top of it [8]. Determining virtualization failures is ever more important as CPS functionality is increasingly virtualized. Today we can find virtualized sensors, gateways, or communication networks[9], which increase the complexity of managing eCPS. Another cause of failures is system management operations, such as upgrades, which can generate failures due to incorrect configurations or interfering operations [10]. Failures can also occur due to software bugs or resource congestion.

To this end, run-time health verification is required to ensure eCPS fulfill their operating requirements, especially after scaling actions adding/removing components at run-time. Most of the existing run-time verification approaches focus on the design of formal methods for the specification of properties that must be verified at run-time [11], [12], or simulate the system behavior in order to verify it [13], [14]. Further, approaches which deal with running systems do not consider their elasticity, such as anomaly detection [15], [16], [6], or complex event processing [17]. eCPS run-time verification requires a mechanism for executing verification tests designed with system elasticity in mind. The platform should be customizable to cope with heterogeneity between components of cyber-physical systems, and enable their run-time verification. Due to the novelty of elastic cyber-physical systems, health analysis features should be provided to be used both by humans and software controllers in determining

This work was partially supported by the European Commission in terms of U-Test H2020 project (H2020-ICT-2014-1 #645463)

[1]http://www.plattform-i40.de/

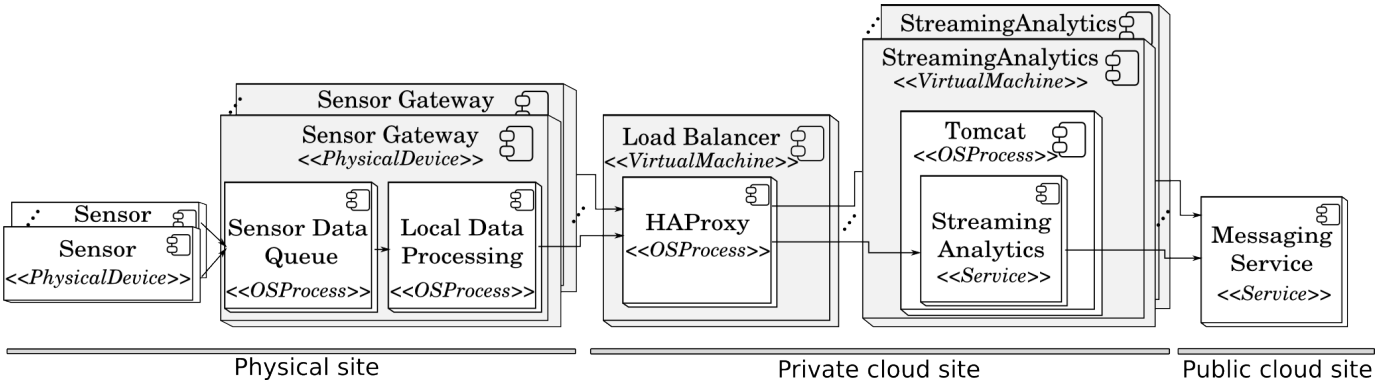

Fig. 1: Architecture and deployment stack of elastic CPS for analysis of streaming data

unwanted behavior.

In this paper we address the problem of ensuring complex elastic cyber-physical systems operate according to their requirements. To this end, we introduce an approach and supporting platform for verifying at run-time if system's components: (i) are deployed and running, (ii) are correctly configured, and (iii) provide expected performance.

To achieve our objective, in the rest of this paper we identify and answer the following research questions :

1) How to capture and manage the structure and deployment stack of Elastic Cyber-physical Systems?: by introducing a model for describing their deployment stack and communication dependencies, and conceptual method for building run-time verifiable eCPS.

2) How to describe run-time verification strategies for Elastic Cyber-physical Systems with varying structure and deployment stack complexity?: by defining a domain-specific language for expressing health verification strategies at various levels of complexity.

3) How to verify Elastic Cyber-physical Systems at run-time considering their particular verification capabilities, structure, and deployment stack?: by defining and implementing a mechanism for enforcing both direct and indirect verification tests, and an integrated run-time health verification platform.

The rest of this paper is structured as follows. Section II presents the motivation behind this work. Section III details our approach for run-time verification of elastic cyber-physical systems. Section IV introduces our run-time verification platform prototype evaluated on an elastic cyber-physical system for analysis of streaming data coming from smart environments. Section V compares and contrasts related work. Section VI concludes the paper and outlines future work.

II. Motivation

The owner of a smart factory builds an elastic cyber-physical system (eCPS) for analysis of streaming data coming from the factory's industrial robots, machines, and environmental sensors (Fig. 1). The system can scale to adapt to changes in load or factory requirements by adding and removing both physical and cyber components. Factory sensors robots send data to physical devices called *Sensor Gateways*. The gateways perform local data processing and sends the data through a HAProxy[2] HTTP *Load Balancer* to *Streaming Analytics* services hosted in virtual machines in a *Private Cloud*. The Streaming Analytics service is deployed as a software artifact in a Tomcat[3] web server. Selected analytics results are published to interested parties through a third-party *Messaging Service* offered as it is by a *Public Cloud* provider.

The smart factory owner wants to ensure that the system is healthy and operates within specified parameters, especially after scaling actions which add/remove components. I.e., the system is correctly configured, its components are deployed and running, and provides expected performance.

To better understand the health issues affecting elastic cyber-physical system, in the following we discuss in general their scaling and failure possibilities. In Fig. 3 we exemplify the possibilities of scaling system components according to their deployment stack. First, one might be able to change software properties at `Software Level`. Considering that each instance of a system component runs as a standalone process, at the `Process Level` one is able to create more component instances, and destroy them processes when no longer needed. This level can benefit situations in which the component does not suffer from resource congestion, and there are still enough computing resources unused. The next elasticity level is the `Virtualization Container Level`, in which processes belonging to different component instances are executed in isolation, due to specific concerns. At this level one can create multiple such containers (e.g. Docker[4]) inside the same virtual or physical machine. Similar to the container level, at the `Virtual Machine` Level one is able to allocate/deallocate virtual machines to run instances of a system component or virtual containers, providing complete OS isolation between each VM, and thus, potentially increased security. Finally, at the `Cloud` level, one is able to change the cloud provider used by the system, while in the `Physical World` a user can chose where to deploy and run the physical devices and machines. As each of these

[2]http://www.haproxy.org/
[3]http://tomcat.apache.org/
[4]https://www.docker.com/

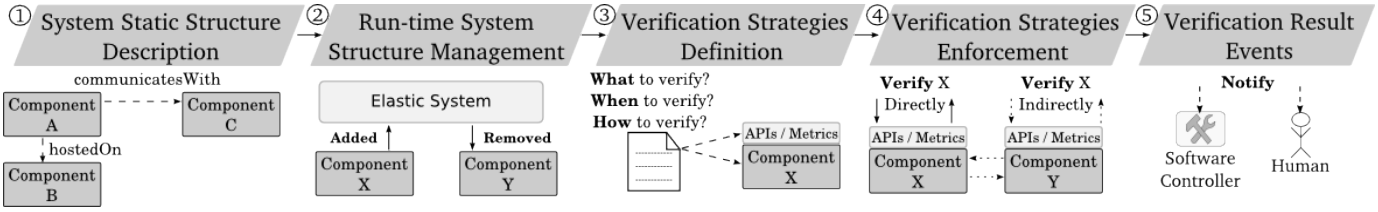

Fig. 2: Approach for run-time verification of elastic cyber-physical systems

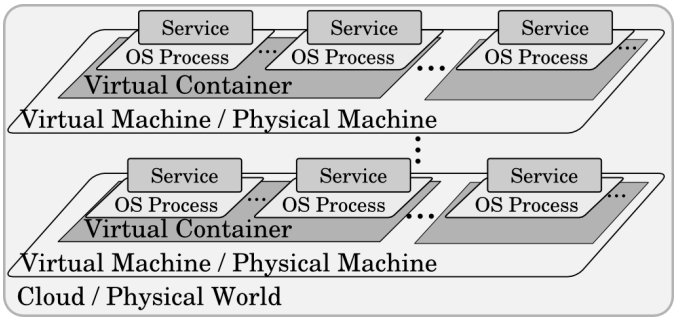

Fig. 3: Possibilities to horizontally scale a system component

Stack Level	Potential failure causes
Service	Incorrect configuration. Software bugs. User behavior.
OS Process	Incorrect configuration. Resources congestion.
Virtual Container	Incorrect configuration. Container middleware failure. Resources congestion.
Virtual Machine	Incorrect configuration. Virtualization middleware failure. Resources congestion.
Physical Device/Machine	Physical device/machine hardware failure. Network failure. Power failure.

TABLE I: Example of potential failures at system different deployment stack levels

levels provides elasticity capabilities, very complex eCPS can be created by combining them.

eCPS failures might occur during the enforcement of scaling actions, or during the normal operation of the system. We capture main failure possibilities and example of potential failure causes in Table I. At the software level common sources of failure are software bugs, incorrect software configuration, or resources congestion. Virtual machines and containers can also exhibit failures originating in configuration errors, virtualization middleware, resources congestion or failure of underlying hardware. In public cloud providers errors can also appear from interaction errors, cost issues, or natural phenomena. Finally, physical devices can exhibit failures generated by external sources such as failure in power, network communication, or device hardware.

For maintaining elastic cyber-physical systems within their operation parameters, their owners need to determine and pinpoint the cause of system health problems. To this end in this paper we introduce a new approach and supporting platform for run-time health verification of eCPS.

III. Run-time Health Verification Approach

We introduce a platform for run-time health verification of elastic cyber-physical systems (eCPS) addressing the needs expressed in the previous section (Fig. 2), providing functionality for:

1) *Specifying the logical structure of elastic cyber-physical systems*, introducing a model capturing their deployment stack and communication dependencies.
2) *Managing the run-time structure of elastic cyber-physical systems*, introducing a decentralized notification-based system for managing addition/removal of system components.
3) *Specifying verification strategies*, introducing a domain-specific language for defining periodic and event-driven execution of direct and indirect verification tests on different system components.
4) *Executing verification strategies*, introducing a distributed mechanism based on remote code execution for execution of verification tests and collection of verification results.
5) *Notifying interested parties about the verification result*, introducing mechanisms for notifying users about changes in the result of verification tests.

A. Health verification tests

We consider verification as *enforcement of verification tests considered black boxes*. This enables us to manage verification tests customized for specific systems, increasing the applicability of our approach. To this end we conceptually define a verification test as a function `Test` in Eq.1:

$$Test : D \rightarrow R \in [0,100] | \begin{cases} 0 = complete\ failure \\ 100 = complete\ success \end{cases} \quad (1)$$

The function applies a set of custom operations having as domain D system specific parameters, and as output a real non-negative number in the [0..100] domain. The output indicates the degree with which the system passed the test, 0 meaning complete test failure and 100 complete success. Using a range for test results enables the specification of also intermediary states, such as "degraded system behavior", as values between 0 an 100. It is the responsibility of specific users to define custom tests and translate their results in the [0,100] range according to particular system requirements and their beliefs over system health [18].

B. Modeling elastic cyber-physical systems

To realize the functionality for *specifying the logical structure of elastic cyber-physical systems* we need a model for capturing the deployment stack and dependencies of system components. As our goal is run-time verification of real eCPS,

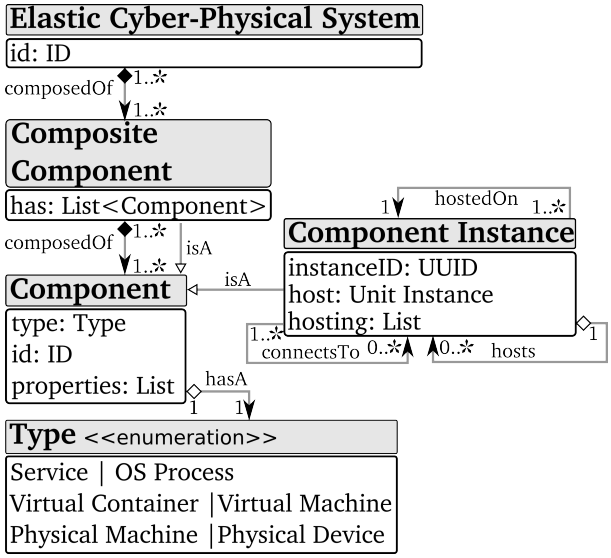

Fig. 4: High level eCPS model

the model must capture the state of the run-time infrastructure. The model must also be applicable to heterogeneous eCPS, and easy to extend with additional types of components depending on particular systems. To this end we introduce an abstract model (Fig. 4) for representing eCPS components and their run-time instances. Our model targets only the infrastructure of eCPS and is designed with simplicity and generality in mind. These properties allow the model to be applied to a wide range of systems without requiring a large amount of domain-specific knowledge.

We first capture *Physical Machine*, *Physical Device*, and *Virtual Machine* (VM) components, crucial in describing systems which run both in the cloud and in the physical world. We capture *Virtual Container* components to describe and verify virtualization containers such as Docker. Increasing the verification detail, we capture *OS Process*, and *Service* components. Capturing components from different stack levels enables hierarchical testing, in which we can verify the lower level (e.g., VM), and if that succeeds, verify the higher levels (e.g., OS Process running inside a VM). Additional component types can be defined by extending the *Type* enumeration.

A system *Component* can have one or more *Component Instances* according to the system's run-time structure. E.g., multiple instances of the Streaming Analytics component from Section II. A component instance can be *hostedOn* another component, e.g., an OS Process running inside a Virtual Machine. The reverse relationship of *hostedOn* is *hosts*, enabling model navigation in the opposite direction. Instances can also communicate with other instances, captured with a *connectsTo* relationships. Further, components can be combined to achieve functionality. We use the term *Composite Component* to describe combinations of multiple system components working towards the same functionality goal. For example, the Streaming Analytics component using a VM hosting a Web Server hosting in turn a RESTful Service.

Literal	Description
Type	Represents a component type according to elastic system representation model captured in Fig. 4
ID	Represents a custom component ID used to identify a component (e.g., a system component) in the system's design-time structure
UUID	Represents the unique ID of a deployed system component instance. An elastic system component (e.g., web server) can have multiple instances running at one time.
Event	Represents a custom defined system event identified by its name or ID (e.g., scale-out)

TABLE II: Literals in verification strategy grammar

Keyword	Description
Description	Marks the test description section
name	Marks the name of the test to be executed
description	Human-readable description of the test to be executed
timeout	Defines a timeout in which test result must be received before considering the test failed
Triggers	Marks the test triggers section defining when the test is executed
event	Specifies that the test should be executed when certain events are encountered
on	Used to specify on which system component the event must be detected to trigger test execution
every	Used to specify periodical test execution
Execution	Marks the section describing what component executes the test
executor	Defines for which components the test is executed, and which components will execute it
for	Used to define what component executes a test defined for the same or another component

TABLE III: Keywords in verification strategy grammar

C. Preparing eCPS for health verification

To be verifiable, systems must expose the necessary verification capabilities to determine health problems deemed important. To this end, a user of our platform must first answer the next questions:

1) What characterizes a system and its components as healthy?: The system developer must decide what does it mean to be healthy for each system component and deployment stack level.

2) When and how can the system and its components fail?: Depending on particular systems, failures can appear anytime, or can be induced by certain control processes.

3) When and how can the system and its components encounter health issues?: Depending on particular systems, unhealthy behaviors can appear anytime, or could be induced by certain control processes.

4) What verification capabilities provide information about system health?: It is important to understand what are the verification capabilities provided by the system, and which must be implemented.

Answering these questions enables system owners or developers to define appropriate verification strategies, for which we introduce in the next section a domain specific language.

D. Defining verification strategies

We realize the functionality for *specifying system verification strategies* by introducing a domain-specific language. The language identifies a set of concepts required to identify the system component to be verified, the verification tests to be enforced, and events defining when the verification tests should be executed. We capture these concepts in Table II, and use them in the language as literals. The keywords used in the language are defined and explained in Table III.

In the following we describe in Extended Backus-Naur Form (EBNF) our grammar for specifying verification strategies. Non-terminals are marked using <>, optional specifications with [], and groupings with (). | should be interpreted as logical OR, and ::= as "is defined as". Enumerations of zero or more elements are marked as $*\{element\}$, and of one or more elements as $+\{element\}$.

Using Production 2 we allow for maximum flexibility the identification of the system component to verify by its ID, instance ID, or Type (according to types defined in Fig. 4). We further capture custom system events used to trigger test executions by event name/id. For flexibility, we enable the enumeration of one or more component or event identifiers using Production 3.

$$\begin{aligned} < id >::= \ & (Type \text{ "."} \ < type >) \mid \\ & (ID \mid UUID \mid Event) \text{ "."} \ < string > \end{aligned} \tag{2}$$

$$< idExpr >::= \ < id > \ *\{\text{","} \ < id >\} \tag{3}$$

We write one verification strategy for each verification test, structured in three parts: (i) test properties `Description`, (ii) specification of test execution `Triggers` , and (iii) test `Execution` information. The test properties can be defined using Production 4, specifying for each test a name, a human-readable description, and optional timeout. The name is used to identify the test. A `timeout` is used to mark as failed tests which do not return results in the specified interval of time.

$$\begin{aligned} < dExpr >::= \ & Description \ (name \text{ ":"} \ < string >) \\ & (description \text{ ":"} \ < string >) \\ & [(timeout \text{ ":"} \ < integer > \ < timeUnit >)] \end{aligned} \tag{4}$$

We use triggers to specify when a particular test should be executed using Production 5. A trigger can be an event, or a periodic timer.

$$\begin{aligned} < tExpr >::= \ & Triggers \\ & (+\{event \text{ ":"} \ < string > \ on \ < idExpr >\}) \\ & (every \text{ ":"} \ < integer > \ < timeUnit >) \end{aligned} \tag{5}$$

We support both direct and indirect tests, as detailed in the next section. Thus, in the last strategy section we specify using Production 5 which component will execute the test. One or more `executor` specifications can be defined, describing which specific executor to execute the test for which specific component identifier. A `distinct` keyword states that the test executor must be other than the test target, useful in executing indirect tests from components with the same identifier (e.g., pinging a VM from another VM).

$$\begin{aligned} < eExpr >::= \ & Execution + \{executor \text{ ":"} \\ & < idExpr > \ for \ + \{< idExpr >\}[distinct]\} \end{aligned} \tag{6}$$

E. Verification strategies enforcement process

For enforcing verification tests we define two core entities. The first entity is a centralized run-time *Verification Orchestrator*. It is responsible for managing the system structure, dispatching tests, and collecting and analyzing results. The second core entity is a *Test Executor*. One Test Executor can be deployed along each system component, and it is responsible for executing tests received from the Verification Orchestrator. It further sends events notifying the Verification Orchestrator when a component instance was added or removed.

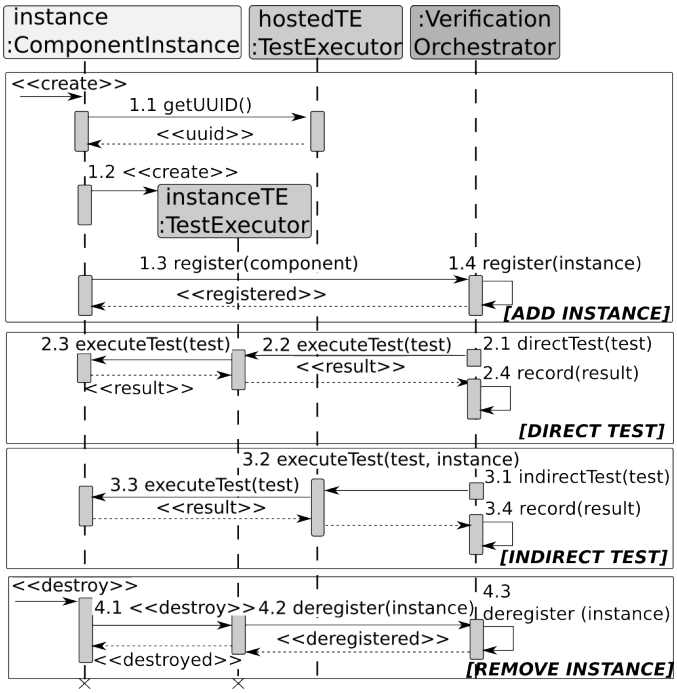

Fig. 5: Run-time verification process and interactions

We determine two types of verification tests to support: *direct* and *indirect*. Direct tests are executed by the test executor of the tested component. E.g., verifying the CPU usage of a VM can be done from inside the VM. However, many tests must be executed indirectly. Thus, an *indirect* test is executed by a third party executor, either the test executor of the component hosting the tested component, or an unrelated executor. E.g., verifying if a VM is running can be done by pinging it from another VM using an indirect test.

The core verification entities and defined interactions are depicted in (Fig. 5). To verify eCPS at run-time, we must maintain an accurate view over their run-time structure, i.e., their components' instances. eCPS can be controlled and managed using both centralized and decentralized mechanisms [19]. A centralized controller could inform about any changes to the system's structure, such as addition/removal of components. However, in autonomous distributed control, each system component might be its own controller. To cover both scenarios, we design a mechanism in which each system component is responsible for notifying about any changes concerned to itself. E.g., each component instance notifies that it has been added to the system, or before being removed. We consider that any action that changes the structure of an elastic system can be mapped to two fundamental actions: {*add*, *remove*} component. We represent the steps and interactions in our approach as a sequence diagram in Fig. 5. When a

new component instance is added in the system (e.g., scaling-out), it will query (step 1.1) the unique identifier (UUID) of the component hosting it (if any). It will then use the UUID to instantiate a Test Executor (step 1.2), which notifies the Verification Orchestrator (step 1.3) that a new component instance was added. The Verification Orchestrator dispatches verification tests. If an indirect test is dispatched (step 2.1), it will be executed by the test executor of the targeted component (i.e., hostedTE:TestExecutor). Indirect tests (step 3.1) are executed by the test executor receiving the test command from the Verification Orchestrator (e.g., hostingTE:TestExecutor). Finally, when a system component is removed from the system (e.g., scaling-in), the component notifies its test executor (step 4.1), which in turn notifies the Verification Orchestrator (step 4.2), which removes the component from its internal system representation. The Verification Orchestrator also generates events to which third parties can subscribe for component addition/removal and test execution results, enabling controllers to reach system behavior.

IV. Evaluation

A. Verification platform prototype

We implement our run-time verification platform prototype[5] (Fig. 6) in Python due to its reduced complexity in deploying and operating the platform. Our platform implements the two entities described in Section III-E: a centralized `Verification Orchestrator` providing most of the platform's functionality, and a `Test Executor` component deployed along system components to enforce verification tests. We expect custom test executors to be implemented for particular target systems, and provide a `Messaging Queue`. The queue acts as a communication broker between the Verification Orchestrator and Test Executors, hiding their particular implementation details from each other. We use RabbitMQ[6] for the queuing middleware, as it supports both AMQP and MQTT protocols, providing a queuing solution applicable to a wide range of systems and components. The platform's functionality is divided between: (i) a `System Structure Manager` handling any structure-related operation; (ii) an `Events Manager` handling the processing of events received from the test executors due to verification results or addition/removal of system component instances; (iii) a `Tests Execution Manager` dispatching verification tests; (iv) a `Persistence Manager` using SQLite[7] to persist system and verification information; and (v) a *UI Manager* handling interactions with the platform's web user interface. For ease of use and integration with third party software components, we implement the interactions with our run-time verification platform as RESTful services using Flask[8] and JSON[9]. We also implement a web-based interface

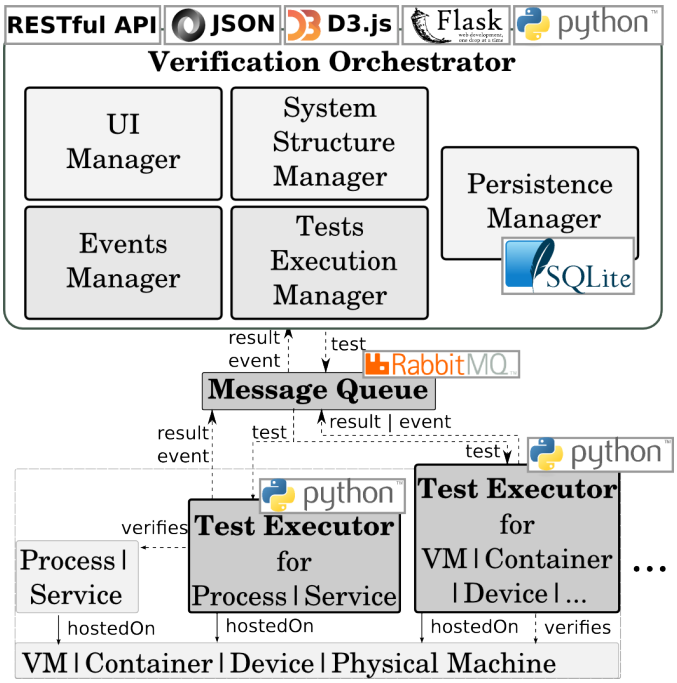

Fig. 6: Run-time verification platform prototype

relying on HTML, Javascript and and D3.js[10] enabling human users to interact with our platform. A verification test is a self-contained sequence of Python code and we provide a library to report the results of particular test executions. We also provide a contextualization mechanism that injects in each python test variables denoting the ids and uuids of the test target and executor to be used in the test.

B. Defining What?, When?, and How? to verify

In the rest of this section we highlight the capabilities of our platform using the system described in Section II. The system owner deploys `Sensor Gateway` components on RaspberryPi[11]. A private OpenStack cloud is used to deploy and run instances of `Streaming Analytics` composite component. For each instance a VM is deployed, running a Tomcat process hosting a Streaming Analytics web service. Finally, a the `Messaging Service` uses a third party message queue software as a service from CloudAMQP[12].

To verify the health of an eCPS, the user first needs to determine *What?*, *When?*, and *How?* to verify. In the following we focus on the `Streaming Analytics` composite component and detail how our platform supports its verification. The health indicators determined from answering the above questions are captured in TABLE IV. The system owner first determines *What?* to verify considering the component structure, and determines the following health indicators:

- The `VM` component is healthy if it is network accessible (TABLE IV row 1)

[5]http://tuwiendsg.github.io/RuntimeVerification/
[6]https://www.rabbitmq.com/
[7]https://www.sqlite.org/
[8]http://flask.pocoo.org/
[9]http://www.json.org/
[10]https://d3js.org/
[11]https://www.raspberrypi.org/
[12]https://www.cloudamqp.com/

- The `Tomcat` component is healthy if its Java process runs and it receives requests from the `Load Balancer` (TABLE IV row 2)
- The `Service` component is healthy if its response time is $<$ 1s (TABLE IV row 3)

Next, the user determines *When* to verify each health indicator, and defines one or more *verification descriptions* for each indicator using our domain specific language introduced in Section III-D. The strategy for verifying if the `VM` component is healthy is depicted in Listing 1. As the Streaming Analytics is elastic, network accessibility should be verified when a new VM is created. A test `Trigger` entry is added (Line 5) for the `event: "Added"` for *ID."VM.StreamingAnalytics"* representing the Streaming Analytics VMs, detected by our verification platform. VMs can also fail at run-time due to various factors. Network accessibility should be also verified periodically during the system's run-time. To this end a `every: 30 s` periodic test trigger is defined in the strategy (Line 6). The executor of the test must also be specified. VM network accessibility should be verified from outside the VM. Thus, a `distinct` executor is requested (Line 9), having the type `VirtualMachine`. Finally, a `timeout` specifies how long to wait for the test result before considering that it has failed (Line 2). This is useful if something happened to the test executor component, e.g., it has also failed.

The user must further decide *How* each health indicator can be verified depending system capabilities. The VM network accessibility indicator can be verified using the `ping` command available in each VM operating system. Using our platform, the test is defined as a standalone Python script depicted in Listing 2. The script can use `contextualized variables` injected at test execution by our platform, such as `targetID`, which for VMs is their IP (Line 6). It is the responsibility of the test designer to use domain-specific knowledge in implementing the test logic and deciding when a test is successful or not (Lines 8-11). Each test result is be returned using the type defined by our platform (Line 13).

Component to verify?	What to verify?	When to verify?	Verification test implementation
1. VM	`VM` network accessible	After event: `VM ADDED`	Linux `ping` command
		Periodically: `every 30 seconds`	
2. Tomcat	`Tomcat Java` process runs.	After event: `VM ADDED`	`Linux-specific` commands: `ps aux | grep tomcat`
	`Tomcat` receives requests from the `Load Balancer`	After event: `VM ADDED`	`Custom system capability` to verify if IP of VM hosting `Tomcat` processes is in `Load Balancer` configuration file
3. Service	`Service` response time is $<$ 1s	Periodically: `every 30 seconds`	`Custom service API` exposing response time

TABLE IV: Health indicators for Streaming Analytics composite component

Listing 1: VM network accessible: verification strategy

```
1  Description
2  timeout: 30
3
4  Triggers
5  every: 30 s
6  event: "Added" on ID."VM.StreamingAnalytics"
7
8  Execution
9  executor: distinct Type.VirtualMachine for
10                              Type.VirtualMachine
```

Listing 2: VM network accessible: test implementation

```
1 #test implemented as standalone python code
2 # all imports must be local
3 os = __import__('os')
4 #contextualized "targetID" variable
5 #executing custom OS command
6 response = os.system("ping -c 1" + targetID)
7 #construct result
8 if response == 0: #if ping fails response is 256
9   success = 100
10 else:
11   success = 0
12 #TestResult type provided by our verification platform
13 return TestResult(success, response)
```

Using our language a user can easily specify what, when, and how to verify. Enabling users to define their verification test as self-contained Python ensures our approach is applicable to a wide range of systems, Python enabling the implementation of both simple and complex tests.

C. Managing structure of elastic cyber-physical systems

To verify a system, its static structure description is submitted to our platform as JSON. An excerpt is shown in Listing 3, detailing the Streaming Analytics composite component. The system is described as a recursive composition of components according to the model introduced in Section III-B. Each component has a `name`, `type`, and potential `containedComponents`. A component can also be hosted on another component, indicated by `hostedOn` property.

In the following we evaluate if our platform detects when the system structure changes due to additions and removal

Listing 3: Static system structure JSON description

```
{ 'name': 'System',
  'containedComponents': [
    { 'name': 'StreamingAnalytics',
      'type': 'Composite',
      'containedComponents': [
        { 'name': 'VM.StreamingAnalytics',
          'type': 'VirtualMachine'
        },
        { 'name': 'Process.Tomcat',
          'type': 'Process',
          'hostedOn': 'VM.StreamingAnalytics'
        },
        { 'name': 'Service.StreamingAnalytics',
          'type': 'Service',
          'hostedOn': 'Process.Tomcat'
        }
      ]
    },
    {'name': 'LoadBalancer',
    ...
    ,{'name': 'MessagingService',
    ...
```

No.	Component Information		
	Type	ID	UUID
1	VirtualMachine	VM.StreamingAnalytics	10.99.0.68
2	Process	Process.Tomcat	10.99.0.68-Tomcat
3	Service	Service.StreamingAnalytics	10.99.0.68-Tomcat-StreamingAnalytics

TABLE V: Events information for added/removed Streaming

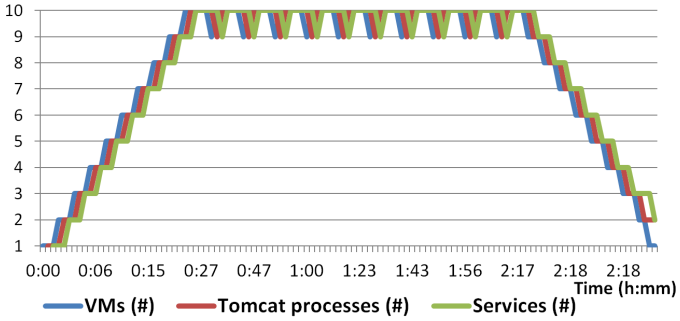

Fig. 7: Number of component instances determined from added/removed events

of Streaming Analytics components. To simulate the behavior of elastic systems, we implement a cloud controller which scales out the system by adding 10 Streaming Analytics component instances, one instance every 2 minutes. After the initial additions, the system goes through a set of 10 scale in/out operations adding and removing one Streaming Analytics component instance every 10 minutes. Finally, the system scales down by removing one Streaming Analytics instance every 2 minutes. In the current evaluation setup the test executors are deployed as OS services inside each VM.

Adding a component instance implies allocating a new VM, deploying and starting a Tomcat process on it, hosting a Streaming Analytics service. Additionally, the IP of the new VM is added in the system's `Load Balancer`, enabling the added component to handle requests and data. One `Test Executor` is deployed for each VM, Process, and Service components. The executor is implemented in our prototype to send messages to the run-time verification platform when the executor service is started and stopped.

Table V shows the events generated by the test executors of one Streaming Analytics instance. Each event contains information about the type, ID, and UUID (unique instance id) of each added component, along with more information not shown here, such as id of the component's system. Based on these events we depict in Fig. 7 the number of VM, Tomcat, and Service instances over the evaluation time.

This evaluation shows that *our platform can be applied on elastic cyber-physical systems, as it can be used to detect when component instances are added or removed.*

D. Determining system health problems due to scaling

Next we focus on evaluating how our platform aids users in determining if particular system components are not healthy. To this end we inject failures when scaling the Streaming Analytics component. We prepare three VM images to be used in scaling. One image contains a correct component configuration. In the second image the component's Tomcat process

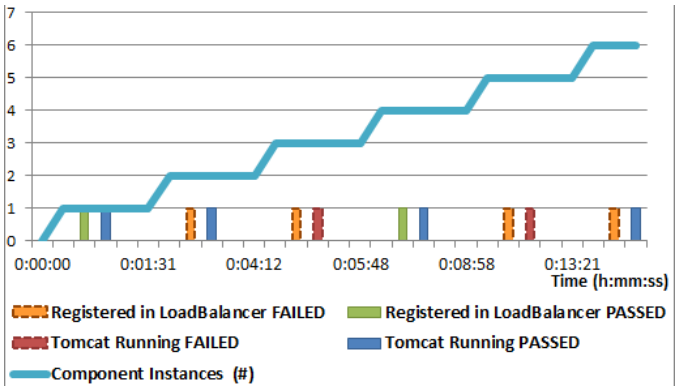

Fig. 8: Verification results for added Streaming Analytics instances

does not register itself in the Load Balancer after starting. In the third configuration the Tomcat process fails to start. Our platform is used to verify the system and generate events if any of the defined tests for Tomcat and Service components fail (row 2 in Table IV). We use the previously implemented cloud controller and iteratively scale out the Streaming Analytics component by adding one VM at a time, iterating through the three configurations. For testing if the added Streaming Analytics instances are healthy we define 2 tests: (i) a `Tomcat Running` test verifying if the Tomcat process is running, and (ii) a `Registered in Load Balancer` test verifying if the IP of the VM hosting the Tomcat process appears in the system's `Load Balancer` configuration.

In Fig. 8 we depict with columns for each test `Passed` and `Failed` events generated by our platform for the first 6 scale out actions. We further depict with a line the number of Streaming Analytics instances, to easily identify that the test results belong to a newly added component instance. In the 6 scaling actions, 3 instances are created for each configuration. From the figure we see that the first instance using the correct configuration passes all tests. The second instance fails the second test, due to configuration 2 not registering the instance in the Load Balancer. The third instance fails both tests.

Thus, *using our platform, users can define fine-grained verification strategies and test their systems at multiple levels.*

E. Determining system health problems at run-time

In the following we highlight how our approach can detect virtual infrastructure failures occurring at run-time. We focus on the health indicator from row 1 in Table IV, and use our platform to define a verification strategy for periodically testing if each VM is network accessible. We use the previously implemented cloud controller, deploy 10 Streaming Analytics instances (i.e., VMs), and introduce iteratively 20 infrastructure failures by `suspending` one random VM at a time. Fig. 9 depicts the test failures determined by our platform and the associated VM IP which has been determined not to function anymore by not responding to ping.

This evaluation scenario highlighted that *our platform can be used to determine health problems emerging during system run-time, identifying the failed component.*

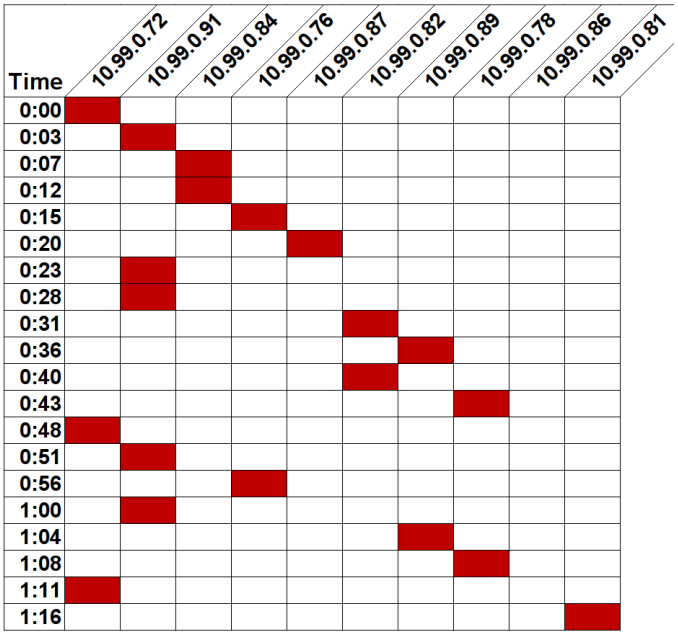

Fig. 9: Determined virtual infrastructure failures

F. Verifying third-party system components

In the following we verify black-box components which permit interaction only through well-defined APIs and do not allow installation of test executors. Verifying such components is crucial for eCPS deployed in industrial scenarios (e.g., Industrie 4.0) where manufacturing machines and robots are considered system components. In the following we focus on the third-party black-box `Messaging Service`. The service is offered by CloudAMQP, exposing over Internet the API of a standalone RabbitMQ deployment. The system owner answers the What? and When? to verify that the component is alive, i.e., its provider has not encountered failures. How? to verify is answered by checking if the RabbitMQ API "/api/overview" is online and accessible. Then, a developer implements the verification test as a Python sequence of code issuing a HTTP GET with his CloudAMQP credentials to the service's API. The system owner or developer further defines a verification strategy to execute the test every 30 seconds from any running VM, describing the test executor as `executor: Type.VirtualMachine for ID."MessagingService"`. Finally, the developer can send an alive message to our platform (Table V) notifying that the component is running and should be tested.

Using our platform, a system owner can verify directly and indirectly both white and black-box components. This is crucial in verifying eCPS operating in industrial scenarios, which can be composed from both white-box software components and black-box physical devices provided by third parties.

V. Related Work

Most of existing verification approaches can be classified from three perspectives: (i) approaches relying on formal methods for the specification of properties that must be verified at run-time, (ii) approaches verifying systems by simulating run-time behaviors and verifying state transitions, and (iii) approaches verifying running systems deployed in real scenarios.

Belonging to the first category, García-Valls et al.[20] introduce an approach for formally verifying at run-time configuration changes in adaptive cyber-physical systems resulting from adaptation processes. Baldellon et al.[11] monitor system properties at run-time, using historical monitoring data to trigger transitions in a Petri net that describes behavioral and temporal properties of the system. Camilli et al.[12] employ a similar approach, using Time-Basic Petri nets to specify and verify the behavior of self-adaptive systems. Compared with formal approaches we tackle problems related verifying running systems. We consider that formal methods provide the techniques for designing verification strategies, and our approach provides the necessary mechanisms for enforcing them at system run-time.

Verifying systems using simulations, Cardozo et al.[13] introduce an approach using symbolic code execution to maintain the system state and verify the system behavior with respect to a set of defined behavioral requirements. The authors also argue that as systems grows in complexity, it becomes increasingly difficult to verify about every possible runtime adaptation in a static context, verifying the system's run-time behavior becoming crucial in determining potential failures. Torjusen et al.[21] argue that properties and objectives of self-adaptive systems must be verified at run-time to cope with changing environmental conditions and the self-adaption itself. To this end the authors introduce run-time verification enablers in a feedback adaptation loop of to guarantee the achievement of security properties in eHealth systems. Ferrante et al.[14] introduce a pattern-based mechanism for describing system behavioral requirements as contracts, explicitly capturing the conditions and assumptions over the behavior and interaction of all system components. Mdhaffar et al. [22] define an adaptive complex event processing architecture for analysis of cloud systems, switching between centralized and distributed analysis depending on requirements. In our approach we do not simulate system behavior. Instead, we introduce a mechanism for verifying real systems during their run-time.

Verifying running systems, Doelitzscher et al.[15] deal with security intrusions in cloud-based systems. The authors introduce a behavioral learning solution which detects behavioral anomalies. Wang et al.[16] introduce a flexible monitoring and analysis middleware for troubleshooting large-scale multi-tier applications used for on-line processing of live data. The middleware collects metrics and determines based on a set of rules when the metric values exceed allowed boundaries. Chen et al.[6] introduce a machine learning approach for predicting job-level and task-level failures in clouds based on historical resource usage metrics. Saleh et al.[17] define a framework for complex event processing which collects cloud infrastructure utilization metrics as data streams. The data streams are further analyzed to determine patterns and relationships providing behavioral indicators about cloud systems. Bonakdarpour et al.[23] introduce a time-triggered approach to run-time verification, in which a monitor takes samples from

the system with a constant frequency, in order to analyze its health. Nelissen et al.[24] highlight that run-time verification cannot be achieved without appropriate monitoring and test enforcement mechanisms, and introduce an approach relying on code introspection for building a run-time verification platform. Todman et al.[25] argue that static testing cannot determine all health problems that can occur at run-time for large systems, and introduce an approach for embedding verification tests as asserts in the system hardware.

Most verification approaches require detailed knowledge about the eCPS software, or do not consider its elasticity. We differ as we introduce a customizable mechanism relying on verification capabilities exposed by each system component. We further tailor our approach for systems which change their structure at run-time, automatically managing their structure.

VI. Conclusions

In this paper we have introduced an approach and supporting platform for run-time verification of elastic cyber-physical systems (eCPS). We have highlighted the importance, challenges, and problems in verifying such systems at run-time. We have defined a model for representing from simple to complex system structures and deployment stacks, based on which we have introduced an approach for verifying them at run-time. We have defined a domain-specific language enabling the specification of verification strategies with varying levels of complexity, supporting both direct and indirect execution of verification tests. We have implemented our approach in a platform for run-time verification of eCPS. The platform provides functionality for automatically managing the changing structure of eCPS, and generates events for each change in system structure and verification test results. We have evaluated our approach on an eCPS having both white and black-box components for analysis of streaming data coming from smart environments. We have demonstrated that using our platform, users can successfully verify elastic cyber-physical systems with complex deployment stack, manage their changing structure, and determine health problems.

We further plan to study and develop classification and analysis techniques on the events received from the verification platform, towards creating a controller to enforce actions addressing determined eCPS health problems.

References

[1] E. A. Lee, "The past, present and future of cyber-physical systems: A focus on models," *Sensors*, vol. 15, no. 3, p. 4837, 2015.

[2] "5th annual trends in cloud computing," CompTIA, Tech. Rep., October 2014. [Online]. Available: https://www.comptia.org/resources/5th-annual-trends-in-cloud-computing

[3] Y. Watanabe, H. Otsuka, M. Sonoda, S. Kikuchi, and Y. Matsumoto, "Online failure prediction in cloud datacenters by real-time message pattern learning," in *International Conference on Cloud Computing Technology and Science (CloudCom)*, Dec 2012, pp. 504–511.

[4] B. Javadi, J. Abawajy, and R. Sinnott, "Hybrid cloud resource provisioning policy in the presence of resource failures," in *International Conference on Cloud Computing Technology and Science (CloudCom)*, Dec 2012, pp. 10–17.

[5] A. Sampaio and J. Barbosa, "Dynamic power- and failure-aware cloud resources allocation for sets of independent tasks," in *International Conference on Cloud Engineering (IC2E)*, March 2013, pp. 1–10.

[6] X. Chen, C.-D. Lu, and K. Pattabiraman, "Failure prediction of jobs in compute clouds: A google cluster case study," in *International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Nov 2014, pp. 341–346.

[7] R. Potharaju and N. Jain, "When the network crumbles: An empirical study of cloud network failures and their impact on services," in *Annual Symposium on Cloud Computing (SOCC)*. New York, NY, USA: ACM, 2013, pp. 15:1–15:17.

[8] J. Navas-Molina and S. Mishra, "Cudswap: Tolerating memory exhaustion failures in cloud computing," in *International Conference on Cloud and Autonomic Computing (ICCAC)*, Sept 2014, pp. 15–24.

[9] H. L. Truong and S. Dustdar, "Principles for engineering iot cloud systems," *IEEE Cloud Computing*, vol. 2, no. 2, pp. 68–76, 2015.

[10] M. Fu, L. Zhu, L. Bass, and A. Liu, "Recovery for failures in rolling upgrade on clouds," in *International Conference on Dependable Systems and Networks (DSN)*, June 2014, pp. 642–647.

[11] O. Baldellon, J. C. Fabre, and M. Roy, "Minotor: Monitoring timing and behavioral properties for dependable distributed systems," in *Pacific Rim International Symposium on Dependable Computing (PRDC)*, Dec 2013, pp. 206–215.

[12] M. Camilli, A. Gargantini, and P. Scandurra, "Specifying and verifying real-time self-adaptive systems," in *International Symposium on Software Reliability Engineering (ISSRE)*, Nov 2015, pp. 303–313.

[13] N. Cardozo, L. Christophe, C. De Roover, and W. De Meuter, "Runtime validation of behavioral adaptations," in *International Workshop on Context-Oriented Programming (COP)*. New York, NY, USA: ACM, 2014, pp. 5:1–5:6.

[14] O. Ferrante, R. Passerone, A. Ferrari, L. Mangeruca, C. Sofronis, and M. DAngelo, "Monitor-based run-time contract verification of distributed systems," in *International Symposium on Industrial Embedded Systems (SIES)*, June 2014, pp. 1–4.

[15] F. Doelitzscher, M. Knahl, C. Reich, and N. Clarke, "Anomaly detection in iaas clouds," in *IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, vol. 1, Dec 2013, pp. 387–394.

[16] C. Wang, I. A. Rayan, G. Eisenhauer, K. Schwan, V. Talwar, M. Wolf, and C. Huneycutt, "Vscope: Middleware for troubleshooting time-sensitive data center applications," in *International Middleware Conference (Middleware)*, 2012, pp. 121–141.

[17] O. Saleh, F. Gropengiesser, H. Betz, W. Mandarawi, and K.-U. Sattler, "Monitoring and autoscaling iaas clouds: A case for complex event processing on data streams," in *International Conference on Utility and Cloud Computing (UCC)*, Dec 2013, pp. 387–392.

[18] M. Zhang, B. Selic, S. Ali, T. Yue, O. Okariz, and R. Norgren, "Understanding uncertainty in cyber-physical systems: A conceptual model," Tech. Rep., Nov 2015. [Online]. Available: https://www.simula.no/publications/understanding-uncertainty-cyber-physical-systems-conceptual-model

[19] A. De Paola, M. Ortolani, G. Lo Re, G. Anastasi, and S. K. Das, "Intelligent management systems for energy efficiency in buildings: A survey," *ACM Computing Surveys*, vol. 47, no. 1, Jun. 2014.

[20] M. García-Valls, D. Perez-Palacin, and R. Mirandola, "Time-sensitive adaptation in cps through run-time configuration generation and verification," in *Computer Software and Applications Conference (COMPSAC)*, July 2014, pp. 332–337.

[21] A. B. Torjusen, H. Abie, E. Paintsil, D. Trcek, and A. Skomedal, "Towards run-time verification of adaptive security for iot in ehealth," in *European Conference on Software Architecture Workshops (ECSAW)*. New York, NY, USA: ACM, 2014, pp. 4:1–4:8.

[22] A. Mdhaffar, R. Ben Halima, M. Jmaiel, and B. Freisleben, "A dynamic complex event processing architecture for cloud monitoring and analysis," in *IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, vol. 2, Dec 2013, pp. 270–275.

[23] B. Bonakdarpour, S. Navabpour, and S. Fischmeister, "Time-triggered runtime verification," *Formal Methods in System Design*, vol. 43, no. 1, pp. 29–60, 2013.

[24] G. Nelissen, D. Pereira, and L. M. Pinho, *Ada-Europe International Conference on Reliable Software Technologies*. Cham: Springer International Publishing, 2015, ch. A Novel Run-Time Monitoring Architecture for Safe and Efficient Inline Monitoring, pp. 66–82.

[25] T. Todman and W. Luk, "Runtime assertions and exceptions for streaming systems," in *International Conference on Field programmable Logic and Applications*, Sept 2013, pp. 1–4.