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. However, failures can appear at run-time in the physical or software resources used by the eCPS. Failures can have different origins, from hardware failure, to management operations, software bugs, or resource congestion. While static verification methods can determine failure sources, they are less applicable to eCPS with complex hardware and software stacks. To this end, in this paper we introduce an approach and supporting platform for verifying at run-time eCPS health, and evaluate it on an eCPS for analysis of streaming data from smart environments.**

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

I. Introduction

A *Cyber-physical System (CPS)* 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, aligning 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 as *Industrie 4.0*[1], where they can enable manufacturing processes to adapt to varying usage patterns and requirements. However, industrial systems are usually mission critical, designed with strict requirements. Combining such systems with elasticity introduces particular challenges and problems. First failures can occur at the cloud provider end [2]. Failures can also originate in the eCPS hardware resources such as servers, storage, or network elements [3], due to various causes such as physical failures, software bugs, or resource congestion. Furthermore, today's eCPS can use cloud or virtualized resources [4], which increases the complexity of managing them.

To this end, run-time health verification is required to ensure eCPS fulfill their operating requirements. Most of existing verification approaches focus on the specification of properties that must be verified at run-time [5], [6], simulate the system behavior in order to verify it [7], [8], or do not consider their elasticity [9], [3]. eCPS require a mechanism for run-time health verification designed with system elasticity in mind. Due to eCPS novelty, the mechanism should be usable both by humans and software controllers.

Let's consider an elastic cyber-physical system (eCPS) for analysis of streaming data coming from sensors (Fig. 1). The system can scale to adapt to changes in load by adding/removing both physical and cyber components. Sensors send data to physical devices called *Sensor Gateways*. The gateways perform a first data processing step, and send it through a HAProxy[2] HTTP *Load Balancer* to *Streaming Analytics* services hosted in virtual machines in a *Private Cloud*. Each Streaming Analytics service is hosted in a Tomcat[3] web server.

In such eCPS failures can occur during scaling, or during normal system operation. At software level, common sources of failure can be software bugs, incorrect configurations, or resources congestion. Virtual machines and containers can also exhibit failures from configuration errors, virtualization middleware errors, resources congestion, or hardware failure. Physical devices can exhibit failures generated by external sources such as power, network, or device hardware.

In this paper we introduce an approach for verifying at run-time if eCPS components are: (i) deployed and running, (ii) correctly configured, and (iii) provide expected performance. To this end in the rest of this paper we identify and answer the following research questions:

- *How to capture and manage the structure and deployment stack of Elastic Cyber-physical Systems?*
- *How to describe run-time verification strategies for Elastic Cyber-physical Systems with varying structure and deployment stack complexity?*
- *How to verify Elastic Cyber-physical Systems at run-time considering their particular verification capabilities, structure, and deployment stack?*

The rest of this paper is structured as follows. Section II details our eCPS run-time verification approach. Section III introduces our verification platform prototype and evaluation. Section IV discusses related work. Section V concludes the paper and outlines future work.

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/

[2]http://www.haproxy.org/

[3]http://tomcat.apache.org/

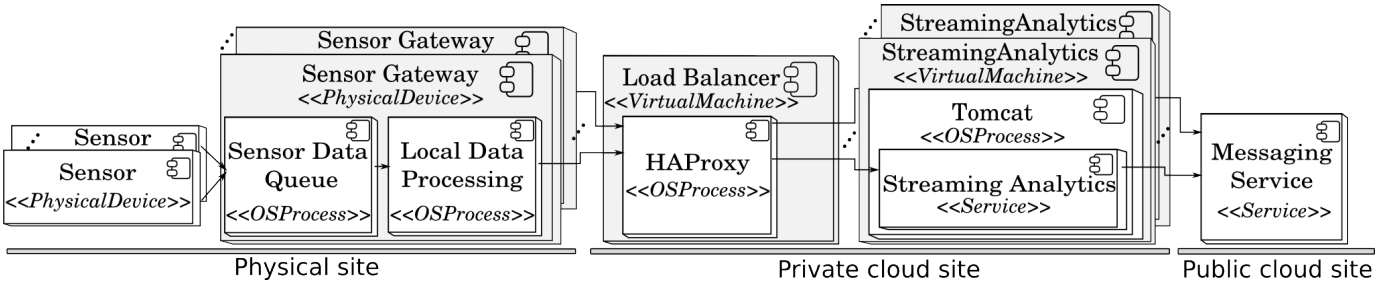

Fig. 1: Architecture, components, and deployment stack of eCPS for analysis of streaming data

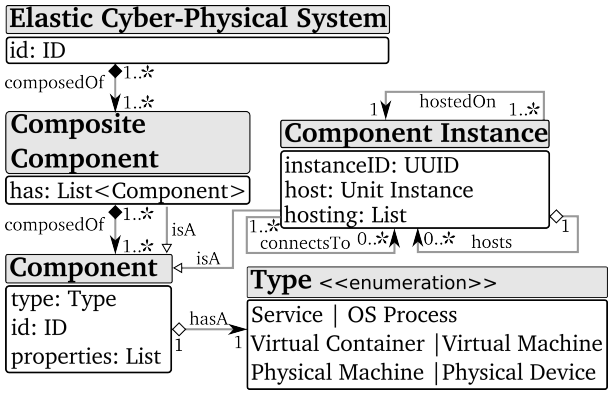

Fig. 2: High level eCPS model

II. Run-time Health Verification Approach

We introduce a platform for run-time health verification of elastic cyber-physical systems (eCPS), providing functionality for: (i) *specifying the logical structure of eCPS*, (ii) *managing the run-time structure of eCPS*, (iii) *specifying verification strategies*, (iv) *executing verification strategies*, and (v) *notifying third-parties about the verification result*.

A. Health verification tests

We consider verification as *enforcement of verification tests considered black box*. This enables us to manage tests customized for specific systems, increasing the applicability of our approach. To this end we conceptually define a verification test as a function $Test : D \rightarrow R \in [0, 100]$. 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 indicating test failure and 100 complete success, according to particular system requirements and beliefs over its health [10].

B. Modeling elastic cyber-physical systems

For *specifying the logical structure of elastic cyber-physical systems* we define a model for capturing the deployment stack and dependencies of system components (Fig. 2). Our model targets only the infrastructure of eCPS and is designed with simplicity and generality in mind. While the model's generality conceptually allows the specification of non-realistic system structures, such as an OS Process hosting a Physical Machine, this can be easily restricted in practice, and it allows the model to be applied to a wide range of systems.

We first capture *Physical Machine*, *Physical Device*, and *Virtual Machine* (VM) components, used in systems which run both in the cloud and in the physical world. We capture *Virtual Container* components to describe virtualization containers such as Docker. Increasing the detail, we capture *OS Process*, and *Service* components. Capturing 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 at run-time one or more *Component Instances*. E.g., multiple instances of the Streaming Analytics component from Section I. A component instance can be *hostedOn* another component. The reverse relationship of *hostedOn* is *hosts*. Instances can also communicate with other instances, captured with a *connectsTo* relationships. Further, we use the *Composite Component* to describe combinations of system components. For example, the Streaming Analytics component using a VM hosting a Web Server hosting in turn a RESTful Service.

C. Preparing eCPS for health verification

To verify cyber-physical systems, the user of our platform must first answer the next questions:

1) What characterizes a system and its components as healthy?: Decide what does healthy means for each system component and deployment stack level.

2) When and how can the system and its components encounter health issues?: Decide if unhealthy behaviors can appear anytime, or after certain events.

3) What verification capabilities provide information about system health?: Understand what are the verification capabilities provided by the system, and which must be implemented.

Answering these questions enables the definition of appropriate verification strategies, for which we introduce in the next section a domain specific language.

D. Defining verification strategies

For *specifying system verification strategies* we introduce a domain-specific language. The language uses a set of concepts required to define the system component to be verified, the verification tests to be enforced, and the events specifying when the verification tests should be executed (Table I). The language keywords are defined and explained in Table II.

Literal	Description
Type	Defines a component type according to elastic system representation model captured in Fig. 2
ID	Defines the component ID from the system's static structure
UUID	Defines the unique ID of a deployed system component instance.
Event	Defines a custom defined system event identified by its ID

TABLE I: Literals in verification strategy grammar

Keyword	Description
Description	Identifies the test description section
name	Identifies the name of the test to be executed
description	Human-readable description of the test to be executed
timeout	Time to wait for result before considering the test failed
Triggers	Identifies 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 the test execution
every	Used to specify periodical test execution
Execution	Identifies 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 for which component the test is executed

TABLE II: Keywords in verification strategy grammar

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".

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

$$< dExpr >::= \; Description\ (name\ "\!:\!"\ < string >)\ (description\ "\!:\!"\ < string >)\ [(timeout\ "\!:\!"\ < integer >\ \ < timeUnit >)] \quad (1)$$

We support direct and indirect tests (detailed in Section II-E). We specify using Production 2 which component will execute the test. A `distinct` executor than the test target can be specified, useful in indirect tests from similar components (e.g., pinging a VM from another VM).

$$< eExpr >::= \; Execution + \{executor\ "\!:\!"\ < idExpr >\ \ for\ \ + \{< idExpr >\}[distinct]\} \quad (2)$$

E. Verification strategies enforcement process

For enforcing verification tests we use two components: (i) a centralized run-time *Verification Orchestrator* responsible for managing the system structure, dispatching tests, and collecting results, and (ii) a *Test Executor*. One Test Executor can be deployed for each component, executing tests received from the orchestrator, and sending events to the orchestrator when a component instance is added/removed. 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 CPU usage from inside the VM. An *indirect* test is executed by a third party executor. E.g., verifying if a VM is running by pinging it from another VM.

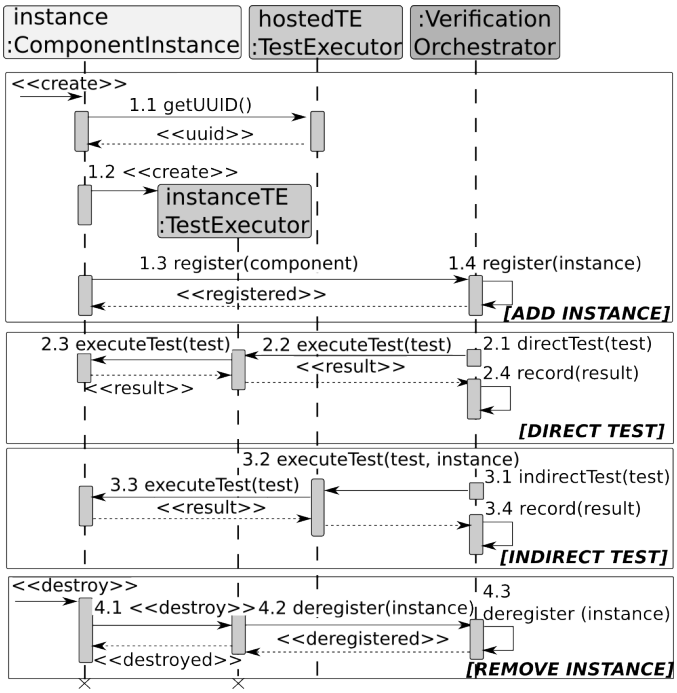

Fig. 3: Run-time verification process and interactions

eCPS can be controlled using centralized or decentralized mechanisms. A centralized controller could inform about changes to the system's structure. In distributed control each system component might be its own controller. To cover both scenarios, we design a mechanism in which the Verification Orchestrator receives events about system changes. We represent the steps and interactions in our approach in Fig. 3. When a new component instance is added it queries (step 1.1) the unique identifier (UUID) of the component hosting it (if any). It then uses it to instantiate a Test Executor (step 1.2), which notifies the Verification Orchestrator (step 1.3) that a new component instance was added. Direct tests (step 2.1) are 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 (e.g., hostingTE:TestExecutor). Finally, when a component is removed from the system, it notifies its test executor (step 4.1), which in turn notifies the orchestrator (step 4.2).

III. Evaluation

A. Verification platform prototype

We implement our run-time verification platform prototype[4] (Fig. 4) in Python. We expect custom test executors to be implemented for particular systems, and provide a `Messaging Queue` using RabbitMQ[5] acting as communication broker between the Verification Orchestrator and Test Executors. The platform's functionality is divided

[4]http://tuwiendsg.github.io/RuntimeVerification/

[5]https://www.rabbitmq.com/

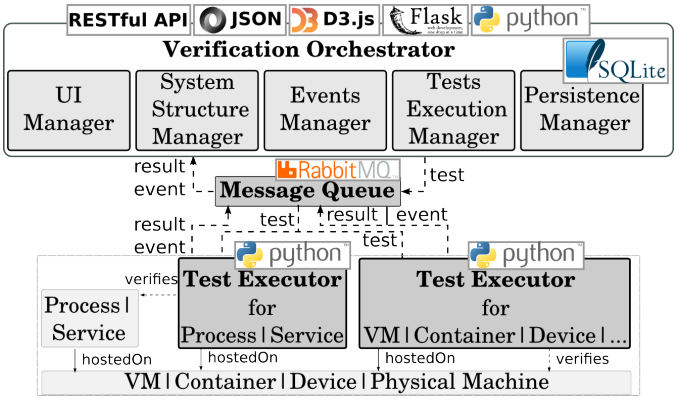

Fig. 4: Run-time verification platform prototype

between: (i) a `System Structure Manager` handling structure-related operations; (ii) an `Events Manager` processing events received from test executors; (iii) a `Tests Execution Manager` dispatching verification tests; (iv) a `Persistence Manager` using SQLite[6] to persist system and verification information; and (v) a `UI Manager` handling interactions with platform users. We implement our run-time verification platform with RESTful services using Flask[7] and JSON[8]. We implement a web-based interface relying on HTML and Javascript for human users. A verification test is a self-contained sequence of Python code, and we provide a library to report the test result. We contextualize each test with information about test target and executor.

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

We highlight the capabilities of our platform using the system described in Section I. `Sensor Gateways` are deployed on RaspberryPi[9]. A private OpenStack cloud is used to deploy instances of `Streaming Analytics` composite component. For each component 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[10].

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 capture in TABLE III the health indicators determined from answering the above questions. Focusing on *What*, the user determines the following health indicators:

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

[6]https://www.sqlite.org/
[7]http://flask.pocoo.org/
[8]http://www.json.org/
[9]https://www.raspberrypi.org/
[10]https://www.cloudamqp.com/

Component to verify?	What to verify?	When to verify?	Verification test implementation
1. VM	`VM` network accessible	After event: `VM ADDED` Periodically: `every 30 seconds`	Linux `ping` command
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 III: Health indicators for Streaming Analytics composite component

Listing 1: VM network accessible: verification strategy

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

Answering *When* to verify, the user defines one or more *verification descriptions* for each health indicator. 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 VM is created. A test `Trigger` entry is added (Line 5) for the `event: "Added"` for *ID."VM.StreamingAnalytics"* representing the Streaming Analytics VMs. VMs can also fail at run-time due to various factors, meaning the network accessibility should also be verified periodically. To this end a `every: 30 s` periodic test trigger is defined (Line 6). VM network accessibility should be verified from outside the VM. Thus, a `distinct` executor is specified (Line 9), having the type `VirtualMachine`. Finally, a `timeout` specifies to wait 30 seconds for the test result before considering that it has failed (Line 2). This is useful if something happened to the test executor component.

The user then decides *How* each health indicator can be verified. The VM network accessibility indicator can be verified by `pinging` it. The test is defined as a standalone Python script (Listing 2), and uses `contextualized variables` injected at test execution by our platform, such as `targetID`, which for VMs is their IP (Line 3). Domain-specific knowledge is used in implementing the test logic (Lines 5-8). Each test result returns the type defined by our platform (Line 9).

Using our language a user can easily specify what, when, and how to verify.

C. Managing structure of elastic cyber-physical systems

The system static structure is submitted to our platform as JSON according to the model introduced in Section II-B.

Listing 2: VM network accessible: test implementation

```
os = __import__('os') #standalone code with local imports
#contextualized "targetID" variable
response = os.system("ping -c 1" + targetID)
#TestResult type provided by our platform
if response == 0: #if ping fails response is 256
  success = 100
else:
  success = 0
return TestResult(success, response)
```

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'
        }]}, ...
```

An excerpt is shown in Listing 3, detailing the Streaming Analytics composite component. Each component has a `name`, `type`, and potential `containedComponents`. A component can also be `hostedOn` another component.

In the following we apply our platform to detect when the system structure changes due to addition/removal of Streaming Analytics components. 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 10 scale-in/out operations adding/removing one Streaming Analytics component instance every 10 minutes. Finally, the system scales-in 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, and deploying and starting a Tomcat process on it. One `Test Executor` is deployed for each VM, Process, and Service components. Our executor sends events to the verification platform when it is started (on VM addition) and stopped (on VM removal). Table IV shows the events from the Streaming Analytics test executors. Each event defines the type, ID, and UUID (unique instance id) of the added component, along with information not shown here, such as the system ID. Based on these events we depict in Fig. 5 the number of VM, Tomcat, and Service instances over 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 evaluate our platform on determining unhealthy components by injecting failures in scaling the Streaming Analytics component. We use three VM images in scaling: (i) one correctly configured, (ii) one in which the Tomcat process does not register itself in the Load Balancer, and (iii) one in which the Tomcat process fails to start. We use the

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 IV: Events information for added/removed Streaming Analytics instance

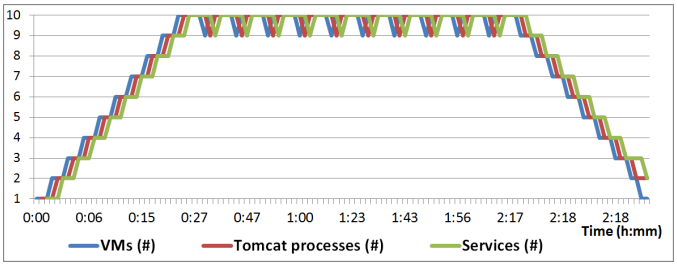

Fig. 5: Number of component instances determined from added/removed events

previously implemented cloud controller and iteratively scale-out the Streaming Analytics component by adding one VM, iterating through the three images. We define 2 tests: (i) a `Tomcat Running` direct test verifying if the Tomcat process is running, and (ii) a `Registered in Load Balancer` indirect test verifying if the `Load Balancer` configuration contains the IP of VM hosting the Tomcat process.

In Fig. 6 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 highlight 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 detect virtual infrastructure failures occurring at run-time. We focus on the health indicator from

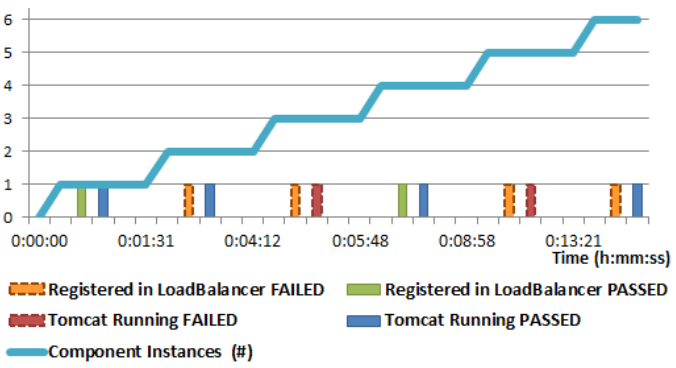

Fig. 6: Verification results for Streaming Analytics instances

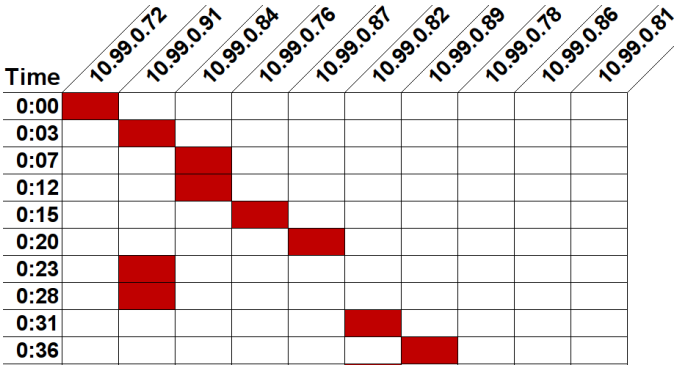

Fig. 7: Determined virtual infrastructure failures

row 1 in Table III, and periodically test if each VM is network accessible. We use the previously implemented cloud controller, deploy 10 Streaming Analytics VMs, and introduce iteratively 10 infrastructure failures by `suspending` one random VM at a time. Fig. 7 depicts the test failures determined by our platform and the associated VM IP.

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

IV. Related Work

Relying on formal specification of properties that must be verified at run-time, [11] formally verifies configuration changes in adaptive cyber-physical systems, [5] uses historical monitoring data to trigger transitions in a Petri net that describes behavioral and temporal properties of the system, while [6] employs Time-Basic Petri nets to specify and verify the behavior of self-adaptive systems. Simulating run-time behavior and verifying state transitions, [7] uses symbolic code execution to maintain the system state and verify its behavior. Security properties in eHealth systems are verified by [12] through run-time verification enablers inserted in feedback adaptation loops. In [8] the authors define a pattern-based mechanism for describing system behavioral requirements as contracts, while [13] defines an adaptive complex event processing architecture for analysis of cloud systems. Verifying running systems, [9] detects behavioral anomalies in cloud-based systems. The authors of [3] present a machine learning approach for predicting job-level and task-level failures in clouds based on historical resource usage metrics, [14] propose a time-triggered approach to run-time verification, while [15] relies on code introspection for run-time verification. Most verification approaches require detailed knowledge about the eCPS, or do not consider its elasticity. We differ as we view system components as black boxes. Our approach further relies on verification capabilities exposed by each system component, and is tailored for systems which change their structure at run-time.

V. 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. 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 and have evaluated our approach on an eCPS for analysis of streaming data coming from smart environments. We further plan to study and develop techniques to classify and analyze the events received from the eCPS.

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] 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.

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.