Conceptually Understanding Uncertainty in Self-Healing Cyber-Physical Systems

Tao Ma
Simula Research Laboratory
Oslo, Norway
+47 482 49 020
taoma@simula.no

Shaukat Ali
Simula Research Laboratory
Oslo, Norway
+47 474 66 831
shaukat@simula.no

Tao Yue
Simula Research Laboratory and University of Oslo
Oslo, Norway
+47 404 33 125
tao@simula.no

ABSTRACT
Smart Cyber-Physical Systems (CPSs) are autonomic systems capable of making decisions at runtime by themselves. These systems are complex in nature and typically operate in highly unpredictable and uncertain environments. One key autonomic capability of such systems is to recover from failures in an autonomic manner, referred to as *self-healing*. Due to the wide range of applications of smart CPSs in our daily life, self-healing behaviors of smart CPSs must be reliable even under uncertainty. Uncertainty and Self-Healing in CPS, in general, are understudied areas of research and thus as a first step towards understanding self-healing and uncertainty, we propose a conceptual model to understand key concepts including self-healing behaviors, uncertainties and their relationships. Our ultimate objective is to provide a unified understanding of these concepts, which forms a foundation for additional analyses in the future such as testing. We validated the conceptual model with six case studies having self-healing behaviors from the literature and industry.

CCS Concepts
• Software and its engineering □ Software creation and management • Software development techniques □ Error handling and recovery.

Keywords
Smart Cyber-Physical Systems; Self-healing; Uncertainty.

1. INTRODUCTION
Smart Cyber-Physical Systems (CPSs) are becoming prevalent in our daily life and their dependability is thus of utmost importance [2]. Such systems are very complex in nature and impose novel challenges for their design, development and testing as compared to traditional embedded systems. First, smart CPSs are autonomic, i.e., capable of making their own decisions during their real operation that are difficult to predict at the design time. Second, smart CPSs typically operate in highly unpredictable environments. Third, uncertainty is inevitable during the operation of smart CPSs due to new types of interactions among control, communication, and computational components.

In this paper, we focus on one autonomic behavior of smart CPS, that is commonly referred to as *self-healing*, i.e., autonomously recovering from failures. Since both testing self-healing behaviors and uncertainty in smart CPSs are relatively understudied areas [6, 7], this paper aims to provide a conceptual model to establish a common understanding of the areas and their relationships. Our ultimate goal is to use this conceptual model as a starting point to develop novel analyses techniques for the design, development, and testing of self-healing behaviors of smart CPSs in the future.

The conceptual model was developed in the following three stages. In the first stage, the conceptual model for self-healing CPSs was developed, based on the literature on CPSs and self-healing systems. Second, we defined an uncertainty taxonomy for self-healing behaviors by extending an existing uncertainty conceptual model of CPSs called *U-Model* reported in [10, 11], existing uncertainty taxonomies from literature, and the self-healing conceptual model developed in the first stage. In the third stage, the conceptual model was validated with six case studies from the literature and industry, in terms of completeness, correctness and redundancy.

The rest of the paper is organized as below. Section 2 gives a running example, Section 1 shows the conceptual model and Section 4 presents the evaluation, followed by the related work (Section 5) and conclusion (Section 6).

2. RUNNING EXAMPLE
An Automatic Power Restoration System (APRS) [13] is the running example that we will use throughout the paper to explain the conceptual model. The key self-healing functionality of APRS is to timely detect electricity failures and restore power supply to affected normal segments without electricity overloading. As shown in Figure 1, in APRS, the power distribution grid is divided into different "teams" bounded by the IntelliDevices. Every IntelliDevice is a compound electricity device, which is controlled by an embedded controller. Via wireless network, the controller periodically reports current, voltage and phasor angle

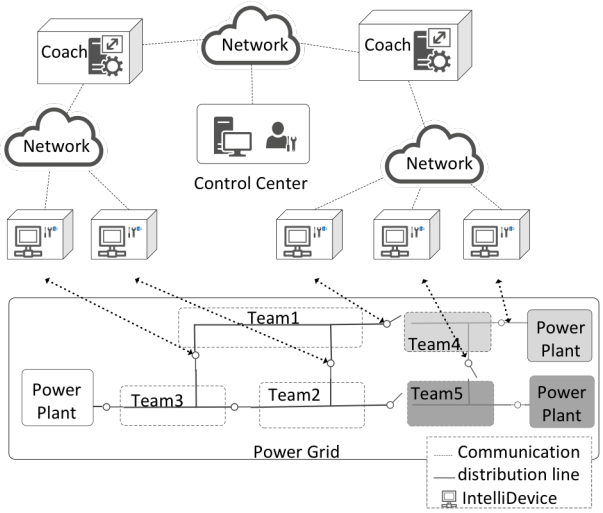

Figure 1. Running Example - Automatic Power Restoration System (APRS)

measurements to the team's "coach", which is a regional controller in the APRS. Based on the collected data, the coach captures the state of power distribution within the team and shares it with other coaches through public or dedicated networks. By this approach, every coach can calculate the topology of the grid and infer the excess power capacity of its neighbor team. This information forms the foundation for power restoration. Whenever a coach detects a line outage, it utilizes the real-time data to locate the fault. Then it calculates which IntelliDevices should interrupt to isolate the fault and sends instructions to them. After fault isolation, several not faulted teams may suffer power loss due to the interruptions. For each of them, its coach will search for an alternative power source that has sufficient excess power capability to compensate its team's power loss and transfer the power load to the first found one.

During this process, some more sophisticated model-based [15] and process history based methods [16] can be used by coaches to detect faults earlier to prevent catastrophic failures. For example, once a loss of power distribution synchronization is detected based on phasor angle measurement, instead of waiting for a blackout, coaches can cooperate to rebalance the power and eliminate the causes of desynchronization.

3. THE CONCEPTUAL MODEL

In this section, we present the conceptual model in three parts: the CPS conceptual model, the self-healing conceptual model and the uncertainty conceptual model, which are discussed in separate subsections.

3.1 The *CPS* Conceptual Model

A conceptual model of CPSs is shown in Figure 2 as a class diagram, whereas details of each concept are presented in Table 1. A *Self-HealingCPS* can be seen as a collection of heterogeneous, distributed and networked *PhysicalUnits* put together to control or monitor *PhysicalProcesses*, e.g., power distribution process in the APRS running example (Figure 1). Such a CPS often has its architecture being centralized, decentralized or hybrid. *Controller*s are the core elements of a *PhysicalUnit*, providing the control logic and computation capabilities to the *PhysicalUnit* and communicate with other *Controller*s owned by other *PhysicalUnit*s via the *Network*. Meanwhile, a *Controller* monitors and controls the *PhysicalProcesse*s optionally via *Sensor*s and *Actuator*s in the same *PhysicalUnit* or through that of other *PhysicalUnit*s. Due to the stochastic nature of the *Environment*, random events may occur and affect the *PhysicalProcesse*s. For example, in the APRS, both power generation and consumption are changing, which affects the power distribution process. Changes in the power generation and consumption are abstracted as *Situation*s in the conceptual model.

3.2 The *Self-Healing* Conceptual Model

The conceptual model is presented in Figure 3 as a class diagram, whereas details of each concept are presented in Table 2 for reference.

In a CPS, both hardware and software may have fault tolerant capabilities. For hardware, fault tolerance is typically achieved via introducing redundant hardware that has limited adaptive capabilities at the runtime. In contrast, software

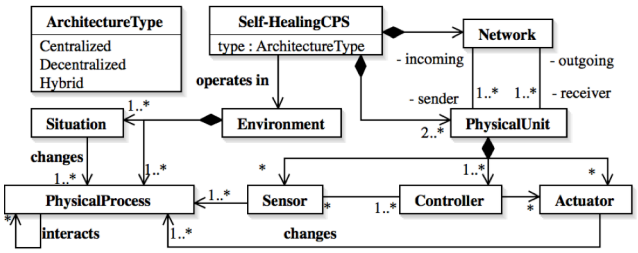

Figure 2. Self-Healing CPS

can be reconfigured and modified, thus in the context of Self-Healing CPSs, *Controller*s provide such self-healing capabilities, as shown in Figure 3.

Self-healing systems are defined by Debanjan Ghosh in [17] as "*a self-healing system should recover from the abnormal (or " unhealthy") state and return to the normative (" healthy") state, and function as it was prior to disruption*". This requires a *Controller* to detect the occurrence of *Error*s in a timely fashion (via *Self-Detection* capabilities of its *HealingBehavior*s), and react to the *Error*s to possibly restore its normal operation. Hence, a *Controller* is equipped with *Probe*s and *Effector*s to make itself self-aware and adaptable. *Probe* and *Effector* are two types of interfaces that are used to inquire *Controller*'s States and adjust *Controller*'s *Behavior*s, respectively.

As shown in Figure 3, a *Controller* has two types of *Behavior*s: 1) *FunctionalBehavior*s implementing business requirements of the *Controller*; 2) *HealingBehavior*s that use *Probe*s and *Effector*s to monitor and maintain the correctness of *FunctionalBehavior*s or *HealingBehavior*s. *HealingBehavior*s are classified as static if they are fixed and defined at the design time, else they are classified as dynamic. Besides, *HealingBehavior*s are implemented at different hierarchical levels, e.g., healing only one function of the *Controller*, (*ControllerLevel*), several functions of a *PhysicalUnit* (*PhysicalUnitLevel*), or the whole *Self-healingCPS* (*SystemLevel*).

One prerequisite of realizing *HealingBehavior*s is the accurate specification of each component's *Goal*s in terms of its performance and/or functional requirements. Moreover, *Goal*s at the system level can be decomposed into several sub-goals at the *PhysicalUnitLevel* and further at the *ControllerLevel*. Eliciting and specifying goals, which have been broadly studied in requirements engineering community, are however out of the scope of this paper.

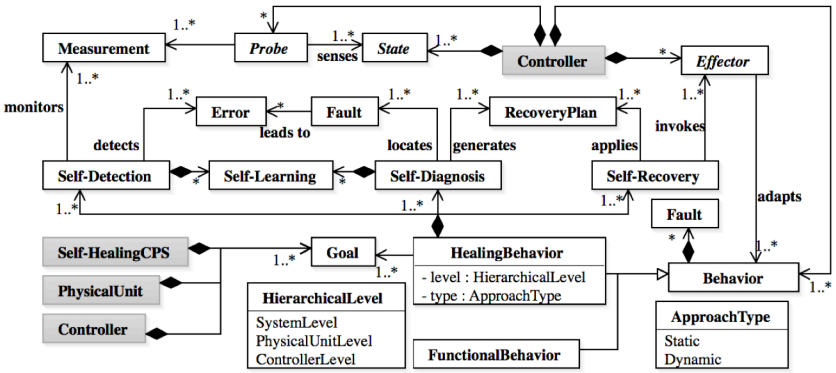

Figure 3. Self-Healing Conceptual Model (Overview)

Table 1. Concepts about CPS

Concept	Definition	Example
C1. Self-HealingCPS	A CPS, which can autonomously detect, diagnose and recover from *Errors* (C18)	APRS is a *Self-HealingCPS*, since it is a distributed control system and it can detect, diagnose, recover or partially repair electricity failures.
C2. PhysicalProcess	A sequence of chemical, physical, or biological activities for the conversion, transport, or storage of material or energy	In the context of APRS, the *PhysicalProcess* it needs to monitor and control is power distribution.
C3. PhysicalUnit	A physical device that can communicate with the others, optionally having the computation and control capabilities	"Coach" and IntelliDevice are the two major kinds of *PhysicalUnit* in APRS. IntelliDevices have sensing and actuation capabilities, but they only have limited computation ability and only have local observations. Whereas, coaches have more powerful computation ability. They collect data from IntelliDevices and share this data with each other to build a global view of power distribution. Based on this, coaches direct each IntelliDevice's actuation behavior to achieve an optimal power distribution.
C4. Network	The medium used as the communication channel among *PhysicalUnit*s (C3)	Usually the *PhysicalUnit*s in APRS use wireless network to communicate with others.
C5. Sensor	A device that measures the physical variables of a *PhysicalProcess* (C2)	Typically IntelliDevices are instrumented with current, voltage and phasor angle sensors to monitor the power distribution states.
C6. Actuator	A device able to change physical quantities of a *PhysicalProcess* (C2)	The *Actuators* used by IntelliDevices are different kinds of switch, which can change the topology of power grid.
C7. Controller	A software deployed on the *PhysicalUnit* (C3), controlling *Sensors* (C5) and *Effectors* (C23) either directly or indirectly, communicating with other *Controllers* and providing computational capability	Both coach and IntelliDevice have *Controller*s, which are in charge of task execution and communication. To differentiate *PhysicalUnit*s and its *Controller*s, the controllers of coach and IntelliDevice are called coach controller and IntelliDevice controller respectively.

*HealingBehavior*s have three capabilities: *Self-Detection*, *Self-Diagnosis* and *Self-Recovery*. First, a *Self-Detection* behavior evaluates the state of a *Controller* according to *Measurement*s collected via *Probe*s. If an *Error* is detected, it means that the state of the *Controller* has deviated from its correct state. Afterwards, a *Self-Diagnosis* behavior is alerted to isolate the cause of the *Error* followed by calculating a *RecoveryPlan* at one time for recovery. Finally, a *Self-Recovery* behavior executes the *RecoveryPlan* via *Effector*s at a given point of time.

The following sections explain the *Self-Detection*, *Self-Diagnosis* and *Self-Recovery* capabilities in details.

3.2.1 The Self-Detection Conceptual Model

Figure 4 presents the *Self-Detection* model, whereas Table 3 provides details of each concept. If *Self-Detection* cannot detect *Error*s in a timely manner, it would be too late to respond to the *Error*s and therefore will fail to recover back to the normal state. The timely detection of *Error*s is even more critical due to the hybrid nature of CPSs. A *PhysicalProcess* is continuous; whereas the computation performed by a *Controller* is discrete. A *controller* must discretize the continuous process in appropriate intervals to avoid missing the detection of *Error*s. If the interval is too long, the detection of an *Error* may be missed; otherwise if the interval is too small, the senor may not be able to obtain *Measurement*s and will miss the *Error*.

*Probe*s can be classified into *PerformanceProbe*s, *EventProbe*s and *PhysicalProcessProbe*s. *PerformanceProbe*s are responsible for monitoring system's performance, such as response time, throughput and availability. These performance metrics are used to verify system's performance requirements. While, an *EventProbe* is in charge of monitoring a *Controller*'s behavior described as a trace of events such as function calls and exceptions. This trace can be checked against system's properties, which are usually in two forms: 1) constraints based on formalisms such as temporal logic [18], state machine [19]. 2) rule based specification using Domain Specific Language (DSL) [20]. In contrast, a *PhysicalProcessProbe* directly uses sensor data to monitor the *PhysicalProcess*es such as the *Self-Detection* behavior can decide if the *PhysicalProcess* is proceeding as expected.

A detection process starts with *Probe*s, which collect their corresponding *Measurement*s periodically or triggered by events. The *Measurement*s include performance metrics, event occurrence (e.g., function calls, exceptions) and physical quantities values of *PhysicalProcess*es, which reflect the *Controller*'s state from the performance, functional and external aspects. *Self-Detection* then checks the *Measurement*s against the *ErrorCriteria*, which is typically in the form of constraints or threshold values. In the field of online monitoring, constraints specifying system's properties (e.g., invariants and behavior's pre/post-conditions) have been broadly used to detect inconsistent behaviors [18]. For example,

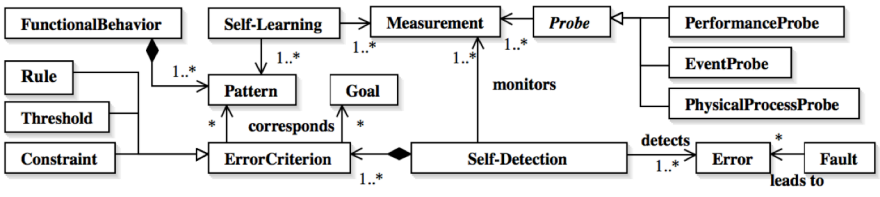

Figure 4. Self-Detection Conceptual Model

Table 2. Concepts about Self-Healing

Concept	Definition	Example
C8. Behavior	Describing a sequence of actions executed by a *Controller* (C7)	The IntelliDevice controller's *Behavior*s are reading current, voltage and phasor angle measurements and invoking switch operation.
C9. FunctionalBehavior	The business logic of a *Controller* (C7) [1]	The *FunctionalBehavior* of coach controllers are scheduled power grid manipulation, which is defined manually for normal distribution conditions.
C10. HealingBehavior	A sequence of *Self-Detection* (C17), *Self-Diagnosis* (C20) and *Self-Recovery* (C22) actions aiming at the recovery of a *Controller* (C7), a *PhysicalUnit* (C3) or the whole *Self-HealingCPS* (C1) from *Errors* (C18)	The *HealingBehavior* of coach controllers are started from electricity failure detection to fault isolation and ended with completion of recovery executions.
C11. HierarchicalLevel	The subordination level within the structure of the *Self-HealingCPS* (C1) [3]	The *HierarchicalLevel* of coach controllers' self-healing behaviors is system level, which aims at enhance power supply reliability of the whole distribution network.
C12. ApproachType	Indicating if a *HealingBehavior* (C10) is static or dynamic.	Since coach controllers detect, isolate and recover electricity distribution errors based on real time data instead of execute predefined process, the *ApproachType* of coach controller is dynamic.
C13. Goal	"... a non-operational objective to be achieved by the composite system" [4]	The *Goal* of APRS is to enhance power distribution reliability, which can be decomposed into two sub-goals: reducing blackout time and power restoration delay.
C14. State	A particular combination of the attribute values of a *Controller* (C7) [8]	The states of a coach controller are defined by the tasks they are currently executing and phases of tasks they are staying.
C15. Probe	A system measurement mechanism, which observes and measures the *States* (C14) of a *Controller* (C7) [9].	Coach controllers use physical process probes to directly monitor the state of power distribution, by which to evaluate the correctness of their instructions to IntelliDevices.
C16. Measurement	A value of a *State* variable, such as performance metrics	Three types of *Sensor* correspond to three *Measurement*s: current, voltage and phasor angle.
C17. Self-Detection	The action that detects *Errors* (C18) from monitored *Measurement*s (C16) and reports the *Errors* (C18) to *Self*-Diagnosis [12]	A coach controller's Self-Detection action is that it utilizes model based or process history based fault detection algorithms to analyze real time power distribution data.
C18. Error	Difference between monitored *Measurement*s (C16) and specified ones [3]	For power distribution network, one of the most common *Error*s is line outage.
C19. Fault	The cause of an *Error* (C18) [14]	The *Fault*s lead to line outage may be power imbalance, short circuit or overloading, etc.
C20. Self-Diagnosis	The action that locates *Fault*s (C19), which lead to detected *Error*s (C18), and calculates an appropriate *RecoveryPlan* (C21) together with *RecoveryPolicy* (C39) [12]	The *Self-Diagnosis* of a coach controller is achieved by using model based reasoning or data driven learning algorithms based on historic and real time *Measurement*s.
C21. RecoveryPlan	A sequence of *AdaptationAction*s (C33), aiming to recover the CPS from *Error*s (C18)	For a coach controller, a *RecoveryPlan* can be a set of IntelliDevice instructions for power rebalance or a sequence of actions for power restoration.
C22. Self-Recovery	The action that applies a *RecoveryPlan* (C21) on the CPS [12]	The *Self-Recovery* of a coach controller is to instruct IntelliDevices to execute the actions according to the *RecoveryPlan*.
C23. Effector	A mechanism carrying out *AdaptationActions* (C33) [9]	The *Effector*s used by coach controllers are control effectors, which can trigger coach controller to generate a new topology for the power distribution network in order to repair the detected *Error*s.
C24. Self-Learning	A mechanism that can recognize system behavior *Patterns* (C25) and/or infer *RecoveryPlan*s (C21) from historic *ExecutionResult*s (C43)	A coach controller's *Self-Learning* is the data driven learning algorithm, which help coach controller abstract key features of the physical process and use them to detect and diagnose faults.

in the APRS, a line flow invariant is that none of the transition lines in the distribution network are overloaded. When a line outrage happens, the power load on the other lines will increase to compensate it and may exceed their transfer capabilities. In this case, the invariant is violated, which means an *Error* has occurred.

Residual thresholds and performance metric thresholds are another kind of criteria. Comparing a threshold with an actual performance or a residual between an actual output and the expected can directly detect undesired behaviors. However, for some dynamic systems including *Self-healingCPS*s, the variation of metrics is very large due to severe environment changes. Therefore, it is difficult to set thresholds at the design time. Whereas, a *Self-Learning* mechanism can alleviate this problem, under the assumption that normal behaviors appear much more frequently than abnormal ones. A self-learning mechanism can cluster monitored *Measurement*s within the most recent period to infer normal behavior *Pattern*s under the current Situation [21].

3.2.2 The Self-Diagnosis Conceptual Model

Figure 5 shows the *Self-Diagnosis* conceptual model, whereas Table 3 provides details of each concept. *Self-Diagnosis* locates *Fault*s leading to detected *Error*s, determines types and locations of the *Fault*s, and decides which *AdaptationAction*s to take as the *RecoveryPlan* to handle the *Fault*s, directed by *RecoveryPolicy*.

Each *AdaptationAction*, provided by *Effector*s, can be considered as a variable point of the system, which enables reconfiguration and adaptation at runtime. Different *AdaptationAction*s have different *Effect*s on the CPS, and may have different *Overhead*s and *Delay*s. The *Self-Diagnosis* behavior has to trade off between adaptation benefits and cost in terms of time and/or resource consumption (i.e., *TimeOverhead* and *ResourceOverhead*). Which *AdaptationAction*s to take at runtime is determined by recovery policies.

In the literature, there are three well-known policies [5]: *ActionPolicy*, *GoalPolicy* and *UtilityFunctionPolicy*. *ActionPolicy* can be seen as a pair in the form of <condition, action>. If the condition is satisfied, then a corresponding action is executed [22]. *GoalPolicy* specifies a set of desired states, which requires that a sequence of *AdaptationAction*s in a *RecoveryPlan* should be taken to make the system from the current faulty state to a desired state [23]. *UtilityFunctionPolicy* defines an objective function containing multiple objectives of the system to guide the system to move towards a desired state in terms of utility values [24].

Because of the tight integration of a CPS with its environment, and its stochastic nature, developing a complete recovery strategy at the design time is non-trivial if not impossible. Similarly, as for detection, *Self-Diagnosis* can benefit from *Self-Learning*. By analyzing every *ExecutionResult* of a *RecoveryPlan*, *Self-Learning* mechanism can infer which combination of *AdaptationAction*s is more effective to repair a kind of *Fault*s, which can be directly used as the *RecoveryPolicy* by the *Self-Diagnosis* actions [25].

3.2.3 The Self-Recovery Conceptual Model.

Figure 6 shows the conceptual model of *Self-Recovery*. *Self-Recovery* executes *RecoveryPlan*s via *Effector*s. Since *Effector*s adapt system's behaviors at runtime, *Self-Recovery* behaviors need to assure that runtime adaptations do not conflict with neither *FunctionalBehavior*s of a *Controller* nor other adaptations. One way to avoid such conflicts is to explicitly design and implement preadaptation and post-adaptation steps in the recovery process. In the preadaptation step, all functional components affected by the adaptations need to be hanged up and blocked

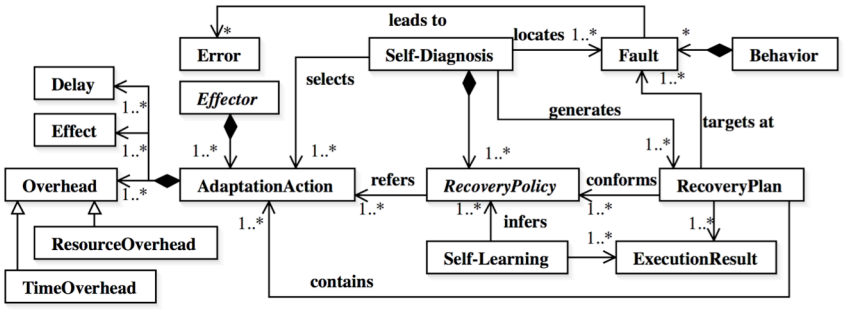

Figure 5. Self-Diagnosis Conceptual Model

from the others. After the completion of applying a *RecoveryPlan*, these components are set back to normal.

*Effector*s used by *Self-Recovery* can be classified into three types. One is *ParameterEffector* that can adjust system components' parameters [26]. The second type is *ArchitectureEffector*, which adds, removes or replaces system components [27]. The last type is *ControlEffector*, which is in charge of changing *FunctionalBehavior* of a *Controller* in response to error conditions. These three types of *Effector*s can be achieved through aspect-oriented programming, reflection or component based design [1].

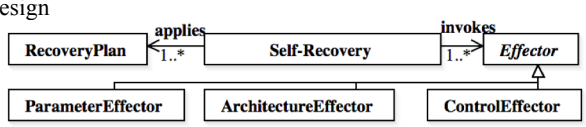

Figure 6. Self-Recovery Conceptual Model

3.3 The Uncertainty Conceptual Model

Uncertainty is intrinsic in the *Self-HealingCPS* due to its tight integration with *PhysicalProcess*es and runtime *HealingBehavior*s. Hence, various types of potential uncertainties should be studied and analyzed in order to establish the confidence that the self-healing CPSs can eventually be able to deal with such uncertainties in a graceful manner during their operation. In this section, we provide a taxonomy of uncertainties specifically for self-healing based on the uncertainty conceptual model for CPS (*U-Model*) defined in [10, 11].

Figure 7 presents the taxonomy of self-healing related uncertainties along with its association with the *Uncertainty* concept from the U-Model [10, 11]. The *U-Model* defines uncertainty (lack of confidence) in a subjective way, i.e., *"uncertainty is modeled as a state (i.e., worldview) of some agent or agency – henceforth referred to as a BeliefAgent – that, for whatever reason, is incapable of possessing complete and fully accurate knowledge about some subject of interest. Since it lacks perfect knowledge, a BeliefAgent possesses a set of subjective Beliefs about the subject."* [10, 11]. As shown in Figure 7, an uncertainty may have an associated *Pattern*, *Locality*, *Effect*, *Measurement*, *Risk*, and *Lifetime* as described in [10] that are once again applicable to specialized classes of uncertainties defined for *Self-HealingCPSs*. Generally, uncertainty is classified into five classes: *Occurrence*, *Content*, *Time*, *GeographicalLocation* and *Environment* [10, 11]. Notice that because of the association between *Self-HealingCPSUncertainty* and *Uncertainty*, each category of self-healing uncertainty can be linked to one or more types of these uncertainties. Below, we provide the taxonomy of uncertainties related to self-healing.

Table 3. Concepts about Healing Behavior

Concept	Definition	Example
C25. Pattern	The mode/style of a system behavior characterized by a combination of *Measurement*s (C16)	In the context of APRS, a *Pattern* can be a trend of phasor angle or voltage change under a specific kind of condition.
C26. ErrorCriterion	The standard, rule, or test, on which a judgment of an *Error* (C18) can be based	For an IntelliDevice, its *ErrorCriteria* are predefined valid current, voltage scopes.
C27. Rule	A pair of a set of preconditions and a set of actions. If all preconditions are satisfied, then the actions are executed	A supervised learning algorithm, e.g., support vector machine (SVM), can use history process data to associate attributes with different types of faults, in the form of rule such as "x1 > a && x2 < b implies A fault"
C28. Threshold	A limit or boundary of *Measurement*s (C16) used to distinguish normal and abnormal values	A predefined minimum and maximum value for current.
C29. Constraint	A specification of system properties that the system must hold during its execution.	A *Constraint* can be a linear temporal logic expression, "no P before Q", where P and Q are both events.
C30. PerformanceProbe	A *Probe* (C15) for monitoring system's performance.	Timely repairing a fault or power restoration is essential for APRS, hence timer is used as a *PerformanceProbe* to evaluate the coach controller's response time.
C31. EventProbe	A *Probe* (C15) for monitoring events occurred in *Self-HealingCPS* (C1)	A fault interruption event publisher is used as a *EventProbe* by IntelliDevices to inform coach controllers that a fault has occurred.
C32. PhysicalProcessProbe	A *Probe* (C15) for monitoring the state of a *PhysicalProcess* (C2)	For coach controllers, *PhysicalProcessProbe*s are the interfaces that enable them get access to IntelliDevices' sensor data.
C33. AdaptationAction	The runtime modification of control data, controlling or affecting a *Controller* (C7)	AdaptationActions that can be used by coach controllers to handle power distribution faults are the switch open and close instructions to the IntelliDevices.
C34. Delay	The time interval between the initiation and completion of an *AdaptationAction* (C33)	The *Delay* of a switch open instruction is the amount of time from emitting it by a coach controller until the completion of the instruction.
C35. Effect	The change of one or more *Behavior*s (C8) caused by an *AdaptationAction* (C33)	The *Effect* of a switch open operation is the current, voltage and phsor angle differences between before and after the operation.
C36. Overhead	Extra resources and/or time used to execute *AdaptationAction*s (C33), in addition to planned resources and/or time at the design time	The *Overhead*s of a switch open operation are the prerequisite operations or post operations of this action, e.g., after the switch is opened, some other switches may need to be closed to restore the power supply.
C37. ResourceOverhead	Extra resources for executing *AdaptationAction*s (C33), as compared to regular CPS resource consumption without adaptation	One of the *ResourceOverhead*s of a switch open operation is the extra power capability used to restore the power.
C38. TimeOverhead	Extra time for executing *AdaptationAction*s (C33), as compared to regular CPS response time without adaptation	The TimeOverhead of a switch open operation is the time that is spent by these post operations.
C39. RecoveryPolicy	A type of formal behavioral guide for *HealingBehavior* (C10) [5]	See below.
C40. ActionPolicy	Specifying which *AdaptationAction*(s) (C33) should be taken for a *Fault* (C19) [5]	For each types of repairable fault, coach controller has a corresponding repair plan.
C41. GoalPolicy	Specifying a set of desirable *States* (C14) of *Self-HealingCPS* (C1), providing the target states for *HealingBehaviors* (C10) [5]	Operator defines a stable condition for the power distribution process. As soon as a coach controller detects the dissatisfactory of that condition, it should take actions to bring the process back.
C42. UtilityFunctionPolicy	Assigning each *State* (C14) of *Self-HealingCPS* (C1) a utility value directing the system moving towards states with a higher utility value [5]	Electricity from different power source has different cost. The cost can serves as the *UtilityFunctionPolicy* and directs coaches to construct a most cost-efficient power distribution network.
C43. ExecutionResult	The consequence and the effect of a *RecoveryPlan* (C21)	The resulting current, voltage and phasor angle values after the recovery plan has been exected.

Based on the various constituting components of *Self-HealingCPS*, i.e., *Sensor*s, *Controller*s, *Actuator*s, *Probe*s, *Effector*s and *Network*s, which, at the same time, are influenced by three components: *PhysicalProcesse*s, *Knowledge*, *Situation*s, we derived nine types of uncertainties, each of which corresponds to one component as shown in Figure 7. Table 4 gives the relations between *Self-HealingCPSUncertainty* and the

uncertainty taxonomy (*U-Model*). The first column lists the self-healing related uncertainties, and the second to the fifth columns show the generic types of uncertainties from the *U-Model*, whereas the last column shows a few examples of the effects of the uncertainties. Notice that the list of effects is by no means complete and just provides examples for each type. In addition, due to space limitation, mapping to the other attributes of uncertainties such as *Pattern* and *Locality* from the *U-Model* is also omitted. It is worth mentioning that we present an initial high level classification of uncertainties in self-healing CPSs that we plan to further extend in the future at more fine-grained level.

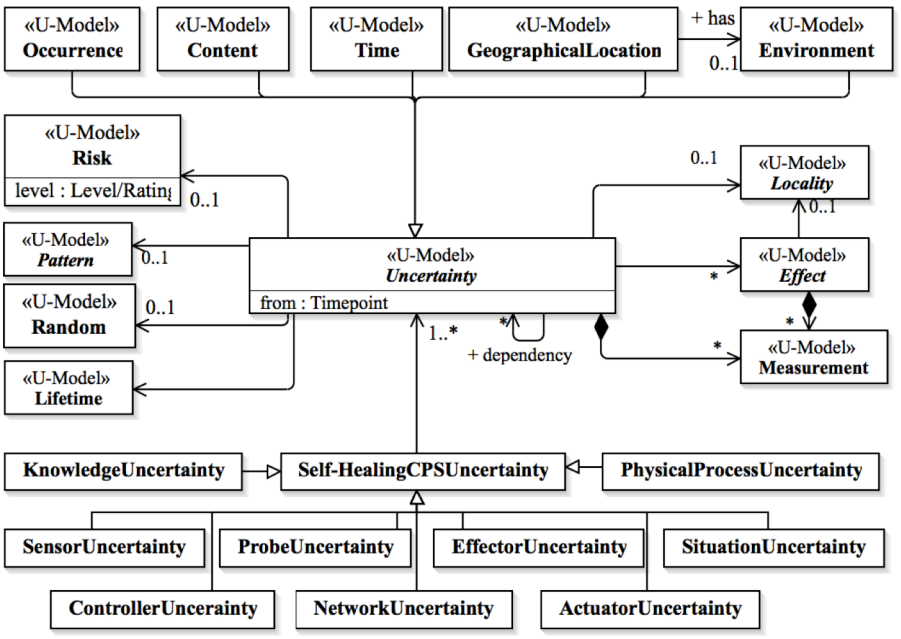

Figure 7. Uncertainty Conceptual Model

The first type is *SituationUncertainty* arising from the environment of a *Self-HealingCPS*. A *SituationUncertainty* is the uncertainty of a belief about *whether*, *when*, *where*, and *in* which context a *Situation* will happen and *which Situation*, corresponding to *Occurrence*, *Time*, *GeographicalLocation*, *Environment* and *Content Uncertainty* respectively. Two effects may be caused by a *SituationUncertainty*. One is uncertain physical environment condition change (e.g., weather, temperature). The other one is unpredictable interactions with other agents or systems (e.g., power plant and power consumers in APRS).

The second type is *PhysicalProcessUncertainty*, which represents the *Uncertainty* about the characteristics of *PhysicalProcess*es. For example, in the APRS, *Uncertainty* exists in the mathematical functions of current and voltage, owing to the unknown resistance of the electricity cable. Due to the dynamic nature of *PhysicalProcess*es, their characteristics are constantly changing. Hence, the occurrence and time of the changes are not relevant. Instead, the content of the changes (*ContentUncertainty*) is more important to be captured by *Self-HealingCPS* to timely adapt its behaviors.

The third one is *SensorUncertainty*, which reflects the uncertain attributes of a sensor, including accuracy (*Content*), lifetime (*Time*), malfunction (*Occurrence*), deployed location and environment (*GeographicalLocation* and *Environment*). A few possible effects of these include measurement errors and sensor failures as shown in Table 4.

Table 4. Mappings to the U-Model concepts

Uncertainty Classes	U-Model Concepts					
	Occurrence	Content	Time	Geographical Location	Environment	Effect
Situation Uncertainty	√	√	√	√	√	Environment change External interaction
PhysicalProcess Uncertainty	×	√	×	×	×	Characteristics of physical process change
Sensor Uncertainty	√	√	√	√	√	Measurement uncertainty Measurement error Sensor failure
Actuator Uncertainty	√	√	√	√	√	Actuation deviation Actuator failure
Probe Uncertainty	√	√	√	×	√	Measurement uncertainty Measurement error Probe failure
Effector Uncertainty	√	√	√	×	√	Effectuation failure
Controller Uncertainty	√	√	√	×	√	Undefined behavior Unexpected behavior Indeterminate behaviors Elusive execution time and resource consumption
Knowledge Uncertainty	√	√	√	×	×	Incorrect awareness Incorrect assumption
Network Uncertainty	√	√	√	×	√	Compromised QoS

√ is type of × is not type of

The fourth type is *ActuatorUncertainty*. Similar with *Sensor*, the *Content* (accuracy), *Occurrence* (malfunction), *Time* (execution time of an actuation), *GeographicalLocation* (instructed position) and *Environment* (operation context) uncertainties are also applicable to *Actuator*s and they may cause the *Actuations* executed by *Actuator*s to deviate from a standard one or lead to an actuator failure.

The fifth and sixth are *Probe* and *Effector* uncertainties, which are similar with *Sensor* and *Actuator* uncertainties from the control perspective. However, *Probes* and *Effectors* are software instead of hardware and thus do not have direct relationship with *GeographicalLocationUncertainty*. *Probe* or *Effector* failures are examples of the effects of the probe and effector related uncertainties.

The seventh type is

ControllerUncertainty, which represents uncertain *Behaviors* of a controller. Due to the missing knowledge about a *Controller*'s requirements, design, implementation, underlying platform and interconnection with other *Controllers*, the actual behavior of a *Controller* (*Content*, *Occurrence*), execution time (Time) and context (*Environment*) of that *Behavior* are uncertain. This leads to undefined, unexpected, and indeterminate behaviors in addition to unpredictable execution time and resource consumption. Undefined behaviors are the ones that are not specified in requirements, unexpected behaviors are the results of incorrect design or implementation, i.e., the actual behaviors do not satisfy controllers' requirements, whereas indeterminate behaviors are the ones that intentionally use some sort of randomized algorithms to make choices such as Genetic algorithms. Time and resource are key factors in CPS, as many control actions have time requirements and resources are limited.

The eighth is *KnowledgeUncertainty*, including uncertain system and context knowledge, such as *States* and *Patterns*. As self-awareness and context-awareness are the prerequisites of self-healing, gaining the knowledge is essential, but also challenging for *Self-HealingCPS*. Because of the changing nature of CPS and environment, the knowledge is also evolving, which further increases the uncertainty. The uncertain knowledge may directly lead to wrong assumptions and incorrect awareness about the system or its environment, due to the wrong decisions based on the uncertain knowledge.

The last one is *NetworkUncertainty*. Suffered from dynamic traffic load (*Environment*), the networks' performance, including network latency, jitter and packet loss rate (*Occurrence*, *Time* and *Content*), keeps on changing, which may dramatically impact the quality of service of a system.

4. EVALUATION

Section 4.1 illustrates the development and validation process of the conceptual model, followed by the evaluation results (Section 4.2).

4.1 Development and Validation Process

Figure 8 shows the development and validation process of the conceptual model, which has four stages. The first stage is the development of the Self-Healing CPS conceptual model, based on the literature of self-healing and CPSs (I1, I2), most of which will be explained in Section 5. The key output is the initial version of the conceptual model (O1). Then, based on the existing uncertainty literature in [10, 11, 28, 29] (I3), and an existing uncertainty conceptual model for CPS, i.e., *U-Model* [10, 11], the second stage extends the initial conceptual model with uncertainty concepts and constructs the complete self-healing CPS model with uncertainties (O2). After that, several CPS and self-healing system architectures or frameworks [9, 30-37] (I4) are utilized by the third stage to refine the derived conceptual model and construct the conceptual model v2.0 (O3). The last step evaluates the conceptual model v2.0 with the six case studies from the industry and literature (I5).

The six systems are Videoconferencing System (VCS) developed by Cisco, Norway and used in our previous research [38], Traffic Monitoring System (TMS) [39], Radio-frequency identification

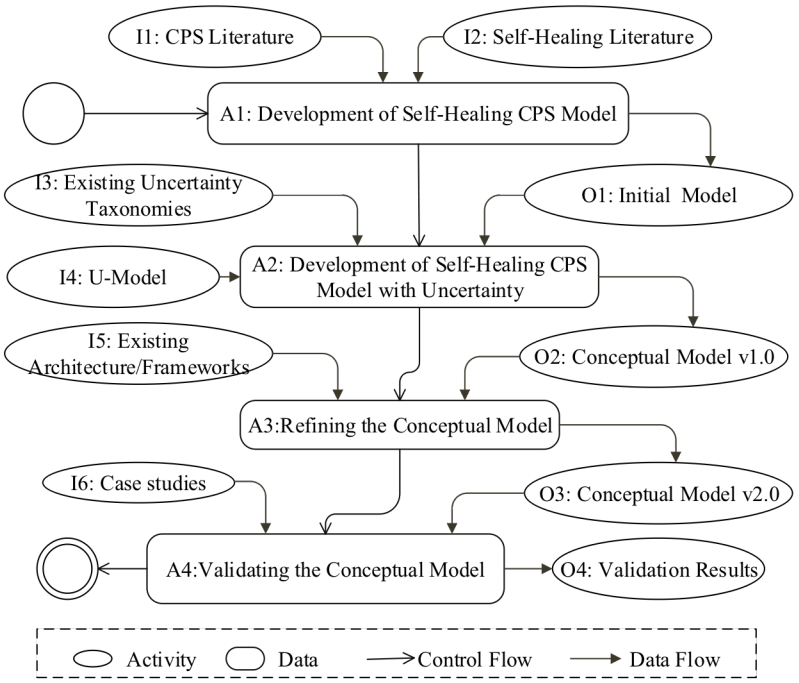

Figure 8. Development and Validation Process

(RFID) supply chain (RFID-SC) [40], Distributed Systems Research Lab (DSRL) [41] Intelligent Service Robot (ISR) [42] and APRS (also used as running example in this paper). We evaluated the conceptual model in terms of its completeness, correctness and redundancy.

First, to check the model's completeness, all constituted elements of the six systems were abstracted from their specifications, followed by mapping them to the concepts of our conceptual model. If there was one element that couldn't be mapped to any concept, it means the model is incomplete. Taking VCS as an example, VCS consists of several videoconferencing terminals (*PhysicalUnit*) and conference conductors (PhysicalUnit) and its goal is to enable high quality videoconferencing (*PhysicalProcess*) among multiple participants. A conference manager uses a voice sensor (*Sensor*) to decide if a speaker's voice needs to be amplified using a voice amplifier (*Actuator*). Each videoconferencing terminal of VCS is equipped with a Real-time Transport Control Protocol (RTCP) reporter (*Probe*), which periodically reports the receiver's packet loss rate (Measurement) to the other terminals. If the receiver's packet loss rate is above a threshold (*ErrorCriterion*), the sender will detect that the packet loss rate is abnormal (*Error*) and assume that it is caused by the network capacity overload (*Fault*). According to the terminal's type, network bandwidth and variation of the packet loss rate, the codec uses a set of static rules (*RecoveryPolicy*) to find out a *RecoveryPlan* corresponding to current State. Codec configurator (*Effector*) provides several *AdaptationAction*s that can be used in the *RecoveryPlan*, including repair P-frames, decoder concealment and down speeding. During the whole process, the occurrence of the packet loss (*NetworkUncertainty*) is the main concern of developing and testing the VCS (TestItem).

Second, the correctness and redundancy of the conceptual model are assessed by investigating if a concept appears in these case studies. If one concept does not appear in any case study or two concepts are mapped to the same element of the case study, then this gave a hint that the concepts may be redundant (Redundancy).

In addition, the concepts' associations in the model are validated against that of actual system elements (Correctness). For example, Figure 2 shows that each *PhysicalUnit* at least contains one *Controller*. If a *PhysicalUnit* does not contains any *Controller*, or a *Controller* is not contained by a *PhysicalUnit* in a case study, this means the multiplicity is wrong.

4.2 Evaluation Results

System elements (e.g., software and hardware components, policies, constraints) were abstracted from specifications that do not contain concrete numbers for each element deployed in the system. We therefore only collected element types and for each type of elements it can have numerous instances. Table 5 reports the number of each concept, contained in the two versions of conceptual models, that is mapped to these element types.

Table 5. Evaluation Results

Concept	VCS		TMS		APRS		RFID-SC		DSRL		ISR		Total V1	Total V2
	V1	V2	V1	V2	V1	V2	V1	V2	V1	V2	V1	V2		
PhysicalUnit	5	5	1	1	4	4	4	4	3	3	1	1	18	18
Sensor	2	2	1	1	3	3	1	1	5	5	4	4	16	16
Controller	5	5	1	1	4	4	4	4	3	3	1	1	18	18
Actuator	2	2	1	1	3	3	0	0	2	2	1	1	9	9
PhysicalProcess	1	1	1	1	1	1	1	1	1	1	1	1	6	6
Probe	2	2	1	1	0	3	5	5	2	5	0	4	10	20
Fault	2	2	3	3	3	3	4	4	3	3	3	3	16	16
ErrorCriterion	2	2	1	1	4	5	3	3	5	6	4	4	19	21
Self-Learning	1	1	0	0	3	3	0	0	1	1	0	0	5	5
RecoveryPolicy	1	1	1	1	1	2	1	1	1	1	1	1	7	7
Effector	2	2	2	2	0	3	2	3	0	4	0	3	6	17
Uncertainty	1	3	2	2	3	5	5	5	2	4	2	2	15	21
Total / Case Study	26	28	15	15	29	38	30	30	28	37	18	25	145	174

It can be seen from the table that every system comprises of one or more types of *PhysicalUnit*, containing several types of *Sensor*s, *Actuator*s and *Controller*s. TMS and RFID-SC are the simplest of the six systems, since they respectively use cameras (*Sensor*s) to monitor traffic [39] and RFID scanners (*Sensor*s) to monitor supply chains [40]. Besides, all the six systems were designed for monitoring or controlling only one kind of *PhysicalProcess*es, such as videoconferencing for VCS and traffic for TMS.

Regarding self-healing, the six systems present great divergence. VCS and RFID-SC rely on *PerformanceProbe*s to monitor performance metrics and use *Threshold*s to detect performance problems [38, 39]. *EventProbe* is used by TMS, RFID-SC and DSRL to detect *Constraint* violations of system behaviors. APRS, DSRL and ISR directly use *PhysicalProcessProbe*s to monitor differences between actual variable values and its expected values computed from empirical reference models.

If *Error*s are detected, most of these systems employ *ActionPolicy*, i.e., finding out *AdaptationAction*s to be executed for the current *Situation* from a set of rules. While, *GoalPolicy* and *UtilityFunctionPolicy* are only applied by APRS and DSRL, respectively. In addition, most systems change their behaviors (*ControlEffector*) to cope with *Fault*s. For example, if a power outage is detected, SM changes the power delivery route to isolate the *Fault* and restore power. In contrast, TMS and RFID-SC dynamically remove or replace a fault component (*ArchitectureEffector*) to eliminate the effect of faults. VCS adjusts the compression ratio and buffer length (*ParameterEffector*) at runtime to repair a performance issue.

One can see from Table 5 that three out of six systems (VCS, APRS and DSLR) have self-learning mechanisms employed in *Self-detection* or *Self-diagnosis* behaviors. VCS and APRS apply a statistical analysis to learn distributions (*Pattern*s) of end-to-end delay, power transfer load and normal phase angle difference. These distributions can be considered as the criteria to detect errors. DSRL uses reinforce learning to learn each *AdaptationAction*'s effectiveness and cost, which is defined in a reward function [25], based on which an optimal sequence of actions can be selected at runtime to handle faults.

Moreover, Table 6 shows the frequency of occurrence of each kind of *Probe*, *RecoveryPolicy*, *Effector* and *Uncertainty*. As shown in Table 6, *PhysicalProcessProbe* and *ControlEffector* are the most common *Probe* and *Effector* used by these self-healing CPSs. This is reasonable since their ultimate goal is to make the physical processes proceed consistently with what users require, *Self-HealingCPS* should assure that the *PhysicalProcess* behave in a valid scope. Thus, monitoring the state of *PhysicalProcess*es is more straightforward than capturing the state of *Controller*s. Besides, due to wear and tear and interaction with hazardous physical environment, hardware is more vulnerable than software. Hence, hardware failure is more common. When it happens, *Controller*s can either replace the faulted component with a

Table 6. Frequency of the Occurrence of Concepts

Concept		VCS	TMS	APRS	RFID-SC	DSRL	ISR	Total	Percentage
Probe	**PerformanceProbe**	2	0	0	3	0	0	5	25%
	EventProbe	0	1	0	2	2	0	5	25%
	PhysicalProcessProbe	0	0	3	0	3	4	10	50%
RecoveryPolicy	**ActionPolicy**	1	1	1	1	0	1	5	71%
	GoalPolicy	0	0	1	0	0	0	1	14%
	UtilityFunctionPolicy	0	0	0	0	1	0	1	14%
Effector	**ParameterEffector**	2	0	0	0	0	0	2	12%
	ArchitectureEffector	0	1	0	2	0	0	3	18%
	ControlEffector	0	1	3	1	4	3	12	71%
Uncertainty	**SituationUncertainty**	0	0	2	1	1	0	4	19%
	PhysicalProcessUncertainty	0	1	1	0	0	0	2	10%
	SensorUncertainty	0	0	1	0	0	1	2	10%
	ActuatorUncertainty	0	0	0	0	0	1	1	5%
	ProbeUncertainty	1	0	0	0	0	0	1	5%
	EffectorUncertainty	0	0	0	0	1	0	1	5%
	ControllerUncertainty	0	0	0	4	1	0	5	24%
	KnowledgeUncertainty	1	0	1	0	1	0	3	14%
	NetworkUncertainty	1	1	0	0	0	0	2	10%

Percentage = n / N n is subclass' appearance num N is class' appearance num

redundant one or change its control logic for manipulation of *PhysicalProcess*es. As redundancy will increase both cost and physical size and it may only be used for key component, the adaptive control is becoming popular [43]. With the respect of *RecoveryPolicy*, *ActionPolicy* is dominating. This is probably due to the simplicity of the case studies. As it can be seen in Table 5, most of them only handle two or three kinds of faults. Although, *ActionPolicy* is easy to implement, it is a static policy and can only work in known situations, which may significantly restrict system's self-healing capability when the operating environment is unknown.

Unsurprisingly, just a few types of uncertainty are explicitly handled in the six case studies. Among them, *SituationUncertainty* and *KnowledgeUncertainty* attract more attention. This may not be typical for all self-healing CPSs, as different systems focus on different aspects. For example, for VCS, the *NetworkUncertainty* has significant impact on the videoconferencing's quality, so VCS applies several healing behaviors to handle it. For APRS, unstable production of green power (*SituationUncertainty*), dynamic demand (*SituationUncertainty*) and occurrence of power outage (*PhysicalProcessUncertainty*) are the main uncertainty concerns.

5. RELATED WORK

After a decade's effort, several key elements of CPSs and self-healing system have been identified by academic and industrial communities and are adopted in our conceptual model. In [6], a CPS was defined as a set of heterogeneous physical units communicating via heterogeneous networks. In this definition, physical units are recognized as the first class objects of CPSs. Besides, the network is also important, as it provides the communication mechanism among the physical units. This definition is consistent with other definitions of CPSs: "engineered systems that are built from, and depend upon, the seamless integration of computational and physical components" [44] and "a set of physical systems controlled in a principled manner via engineering technologies" [45].

Sensors and actuator are captured as interfaces between computational and physical components in [46]. CPSs are characterized by integrating computation and physical processes [47] and the primary goal of a CPS is to efficiently control physical processes [48]. The authors of [12] identified detection, diagnosis and recovery as the three main steps of self-healing. In addition, three types of recovery policies were explained and evaluated in [5, 49].

To further understand self-healing CPSs, we generalized and abstracted concepts from the literature of CPSs and self-healing systems. First, *Situation* is identified as an important concept of self-healing CPSs, representing represents inherent uncertain events happened in the operational environment of a CPS, which is targeted by several approaches [50, 51]. Second, error criteria, adaptation actions, the classification of probes and effector were elicited from error detection [52] and recovery processes [12]. Third, according to the self-learning mechanisms used by self-detection and self-diagnosis [53], inputs (measurement, execution results) and outputs (pattern, recovery policy) of self-learning mechanisms were defined. Fourth, inspired by goal oriented self-healing approaches [54], goals of self-healing behaviors is captured in the conceptual model.

How to cope with uncertainty is a grand challenge and a definition and taxonomy of uncertainty in the context of CPSs is difficult to find [55]. In the past, effort was mostly spent on identifying uncertainty sources in self-healing CPSs. The authors of [28] proposed a taxonomy of uncertainty sources in dynamically adaptive systems at the requirement, design and execution phases along with existing mitigation techniques for each type of uncertainties. Though the taxonomy is extensive and generic, it is not designed for a specific usage and needs to be specialized for specific applications. In [29], the authors gave another nine uncertainty sources in self-adaptive systems, which needs to be considered during design. As the types of uncertainties were extracted in an ad hoc manner, the types are not orthogonal and are not well structured. Whereas, we applied a component centric approach to build the uncertainty model. Particularly, we adopt the an uncertainty taxonomy (U-Model) from [10, 11]. As U-Model gives a set of concepts related with uncertainty in CPS, it helped us to systematically analyze the potential uncertainties for self-healing CPS. For each new defined uncertainty type, it is mapped to the U-Model concepts, which also instantiates the U-Model in the context of self-healing CPS.

In summary, to fulfill their dependability requirements, CPSs are expected to be more autonomic, e.g., in terms of having built-in self-healing capabilities. However, self-healing CPSs are an emerging field and uncertainty in CPSs is a relevantly under-studied subject in software engineering. Despite numerous approaches proposed [12, 17, 56], a conceptual model of CPSs and their self-healing behaviors together with uncertainty is still missing. We, in the paper, took the initiative and constructed such a conceptual model, aiming at providing a common ground for understanding self-healing CPSs under uncertainty and facilitating analyses in the future. However, we believe that this conceptual model is an initial attempt and must be specialized such as for other types of autonomic behaviors, e.g., self-configuring and different types of analyses such as model-based testing.

6. CONCLUSION

Smart Cyber-Physical Systems (CPSs) are becoming increasingly autonomic and thus must be able to cope with diverse uncertainties originating from both environment and their internal components. In addition, autonomic behaviors themselves induce uncertainties in the system and thus CPSs must have self-healing capabilities to deal with errors introduced by these uncertainties. As a first step towards understanding self-healing (one type of autonomic behaviors) and uncertainty in CPSs, we proposed a unified conceptual model comprising of the following three conceptual models: CPS, Self-Healing, and Uncertainty. The conceptual model is developed to provide a unified and comprehensive understanding of self-healing CPS and uncertainty. Based on this conceptual model, a MBT approach will be proposed to discover more flaws in self-healing CPS in the presence of uncertainty in our future work. The conceptual model was evaluated with six case studies from the literature and industry.

7. Acknowledgements

This research was supported by RCN funded MBT4CPS project. Tao Yue and Shaukat Ali are also funded by the EU Horizon 2020 funded project U-Test (Testing Cyber-Physical Systems under Uncertainty), RCN funded Zen-Configurator project, RFF Hovedstaden funded MBE-CR project, and RCN funded Certus SFI.

8. REFERENCES

[1] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. Cheng, 2004, "Composing adaptive software," *Computer*, no. 7, pp. 56-64.

[2] A. W. Colombo, S. Karnouskos, and T. Bangemann, 2014, "Towards the Next Generation of Industrial Cyber-Physical Systems," *Industrial cloud-based cyber-physical systems*, pp. 1-22: Springer.

[3] ISO/IEC/IEEE, 2010, "24765:2010 Systems and software engineering," http://www.iso.org/iso/catalogue_detail.htm?csnumber=50518.

[4] A. Dardenne, A. Van Lamsweerde, and S. Fickas, 1993, "Goal-directed requirements acquisition," *Science of computer programming,* vol. 20, no. 1, pp. 3-50.

[5] J. O. Kephart, and W. E. Walsh, Year, "An artificial intelligence perspective on autonomic computing policies." pp. 3-12, Published.

[6] S. Ali, and T. Yue, Year, "U-Test: Evolving, Modelling and Testing Realistic Uncertain Behaviours of Cyber-Physical Systems." pp. 1-2, Published.

[7] R. De Lemos, H. Giese, H. A. Müller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N. M. Villegas, and T. Vogel, 2013, "Software engineering for self-adaptive systems: A second research roadmap," *Software Engineering for Self-Adaptive Systems II*, pp. 1-32: Springer.

[8] K. Koskimies, and E. Mäkinen, 1994, "Automatic synthesis of state machines from trace diagrams," *Software: Practice and Experience,* vol. 24, no. 7, pp. 643-658.

[9] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, 2004, "Rainbow: Architecture-based self-adaptation with reusable infrastructure," *Computer,* vol. 37, no. 10, pp. 46-54.

[10] M. Zhang, S. Ali, and T. Yue, 2016, "An Uncertainty Taxonomy for Cyber-Physical Systems," *Modelling Foundations and Applications: 12th European Conference, ECMFA 2016*.

[11] Z. Man, B. Selic, S. Ali, T. Yue, O. Okariz, and R. Norgren, 2015, "Understanding Uncertainty in Cyber-Physical Systems: A Conceptual Model," *Modelling Foundations and Applications: 12th European Conference, ECMFA 2016 Available in https://*http://www.simula.no/file/u-modeltrfinalpdf/download.

[12] H. Psaier, and S. Dustdar, 2011, "A survey on self-healing systems: approaches and systems," *Computing,* vol. 91, no. 1, pp. 43-73.

[13] D. Macleman, W. Bik, and A. Jones, Year, "Evaluation of a self healing distribution automation scheme on the Isle of Wight." pp. 1-4, Published.

[14] A. Avižienis, J.-C. Laprie, B. Randell, and C. Landwehr, 2004, "Basic concepts and taxonomy of dependable and secure computing," *Dependable and Secure Computing, IEEE Transactions on,* vol. 1, no. 1, pp. 11-33.

[15] V. Venkatasubramanian, R. Rengaswamy, K. Yin, and S. N. Kavuri, 2003, "A review of process fault detection and diagnosis: Part I: Quantitative model-based methods," *Computers & chemical engineering,* vol. 27, no. 3, pp. 293-311.

[16] V. Venkatasubramanian, R. Rengaswamy, S. N. Kavuri, and K. Yin, 2003, "A review of process fault detection and diagnosis: Part III: Process history based methods," *Computers & chemical engineering,* vol. 27, no. 3, pp. 327-346.

[17] D. Ghosh, R. Sharman, H. R. Rao, and S. Upadhyaya, 2007, "Self-healing systems—survey and synthesis," *Decision Support Systems,* vol. 42, no. 4, pp. 2164-2185.

[18] M. Leucker, and C. Schallhart, 2009, "A brief account of runtime verification," *The Journal of Logic and Algebraic Programming,* vol. 78, no. 5, pp. 293-303.

[19] F. Chen, and G. Roşu, 2009, "Parametric trace slicing and monitoring," *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 246-261: Springer.

[20] K. Havelund, 2015, "Rule-based runtime verification revisited," *International Journal on Software Tools for Technology Transfer,* vol. 17, no. 2, pp. 143-170.

[21] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, and G. Sullivan, Year, "Automatically patching errors in deployed software." pp. 87-102, Published.

[22] V. Koutsoumpas, Year, "A model-based approach for the specification of a virtual power plant operating in open context." pp. 26-32, Published.

[23] J. Simmonds, S. Ben-David, and M. Chechik, 2010, "Monitoring and recovery of web service applications," *The smart internet*, pp. 250-288: Springer.

[24] S.-W. Cheng, D. Garlan, and B. Schmerl, Year, "Architecture-based self-adaptation in the presence of multiple objectives." pp. 2-8, Published.

[25] M. Amoui, M. Salehie, S. Mirarab, and L. Tahvildari, Year, "Adaptive action selection in autonomic software using reinforcement learning." pp. 175-181, Published.

[26] P. Siripongwutikorn, S. Banerjee, and D. Tipper, 2003, "A survey of adaptive bandwidth control algorithms," *Communications Surveys & Tutorials, IEEE,* vol. 5, no. 1, pp. 14-26.

[27] D. Garlan, and B. Schmerl, Year, "Model-based adaptation for self-healing systems." pp. 27-32, Published.

[28] A. J. Ramirez, A. C. Jensen, and B. H. Cheng, Year, "A taxonomy of uncertainty for dynamically adaptive systems." pp. 99-108, Published.

[29] N. Esfahani, and S. Malek, 2013, "Uncertainty in self-adaptive software systems," *Software Engineering for Self-Adaptive Systems II*, pp. 214-238: Springer.

[30] A. Elkhodary, N. Esfahani, and S. Malek, Year, "FUSION: a framework for engineering self-tuning self-adaptive software systems." pp. 7-16, Published.

[31] M. Elhadi, and A. Abdullah, Year, "Layered biologically inspired self-healing software system architecture." pp. 1-9, Published.

[32] R. Pegoraro, M. A. R. Sacoman, and J. M. Rosário, Year, "A Self-Healing Architecture for Web Service-Based Applications." pp. 221-226, Published.

[33] S. Neti, and H. A. Muller, Year, "Quality criteria and an analysis framework for self-healing systems." pp. 6-6, Published.

[34] J. P. Magalhaes, and L. Moura Silva, Year, "A Framework for Self-healing and Self-adaptation of Cloud-hosted Web-based Applications." pp. 555-564, Published.

[35] T. A. Nguyen, M. Aiello, T. Yonezawa, and K. Tei, Year, "A Self-healing Framework for Online Sensor Data." pp. 295-300, Published.

[36] J. Y. Lee, D. W. Cheun, and S. D. Kim, Year, "A comprehensive framework for mobile cyber-physical applications." pp. 1-6, Published.

[37] L. Hu, N. Xie, Z. Kuang, and K. Zhao, Year, "Review of cyber-physical system architecture." pp. 25-30, Published.

[38] S. Ali, L. C. Briand, and H. Hemmati, 2012, "Modeling robustness behavior using aspect-oriented modeling to support robustness testing of industrial systems," *Software & Systems Modeling,* vol. 11, no. 4, pp. 633-670.

[39] P. Vromant, D. Weyns, S. Malek, and J. Andersson, Year, "On interacting control loops in self-adaptive systems." pp. 202-207, Published.

[40] K. Gama, and D. Donsez, 2014, "Deployment and activation of faulty components at runtime for testing self-recovery mechanisms," *ACM SIGAPP Applied Computing Review,* vol. 14, no. 3, pp. 44-54.

[41] T. Cioara, I. Anghel, I. Salomie, M. Dinsoreanu, G. Copil, and D. Moldovan, Year, "A reinforcement learning based self-healing algorithm for managing context adaptation." pp. 859-862, Published.

[42] J. Park, S. Lee, T. Yoon, and J. M. Kim, 2015, "An autonomic control system for high-reliable CPS," *Cluster Computing,* vol. 18, no. 2, pp. 587-598.

[43] R. De Lemos, H. Giese, H. Müller, M. Shaw, J. Andersson, L. Baresi, B. Becker, N. Bencomo, Y. Brun, and B. Cikic, Year, "Software engineering for self-adaptive systems: a second research roadmap," Published.

[44] N. S. Foundation, "Cyber Physical Systems," http://www.nsf.gov/pubs/2014/nsf14542/nsf14542.htm.

[45] K.-D. Kim, and P. R. Kumar, 2012, "Cyber–physical systems: A perspective at the centennial," *Proceedings of the IEEE,* vol. 100, no. Special Centennial Issue, pp. 1287-1308.

[46] E. A. Lee, and S. A. Seshia, 2011, *Introduction to embedded systems: A cyber-physical systems approach*: Lee & Seshia.

[47] J. Shi, J. Wan, H. Yan, and H. Suo, Year, "A survey of cyber-physical systems." pp. 1-6, Published.

[48] S. Sridhar, A. Hahn, and M. Govindarasu, 2012, "Cyber–physical system security for the electric power grid," *Proceedings of the IEEE,* vol. 100, no. 1, pp. 210-224.

[49] S. R. White, J. E. Hanson, I. Whalley, D. M. Chess, and J. O. Kephart, Year, "An architectural approach to autonomic computing." pp. 2-9, Published.

[50] M. C. Bujorianu, M. L. Bujorianu, and H. Barringer, Year, "A unifying specification logic for cyber-physical systems." pp. 1166-1171, Published.

[51] G. A. Moreno, J. Cámara, D. Garlan, and B. Schmerl, 2015, "Proactive Self-Adaptation under Uncertainty: a Probabilistic Model Checking Approach}."

[52] A. E. Goodloe, and L. Pike, 2010, "Monitoring distributed real-time systems: A survey and future directions."

[53] C. Schneider, A. Barker, and S. Dobson, 2014, "A survey of self‐healing systems frameworks," *Software: Practice and Experience*.

[54] M. Morandini, L. Penserini, and A. Perini, Year, "Automated mapping from goal models to self-adaptive systems." pp. 485-486, Published.

[55] T. Bures, D. Weyns, C. Berger, S. Biffl, M. Daun, T. Gabor, D. Garlan, I. Gerostathopoulos, C. Julien, and F. Krikava, 2015, "Software Engineering for Smart Cyber-Physical Systems--Towards a Research Agenda: Report on the First International Workshop on Software Engineering for Smart CPS," *ACM SIGSOFT Software Engineering Notes,* vol. 40, no. 6, pp. 28-32.

[56] S. K. Khaitan, and J. D. McCalley, 2014, "Design techniques and applications of cyberphysical systems: a survey."

[57] M. Utting, and B. Legeard, 2010, *Practical model-based testing: a tools approach*: Morgan Kaufmann.

[58] ISO/IEC, 2013, "ISO/IEC29119 The International Software Testing Standard," http://www.softwaretestingstandard.org/.

9. Appendix

9.1 Conceptual Model of Testing

Considering the complexity and uncertainty of Self-HealingCPS, designing and deploying systematic verification and validation methods for Self-HealingCPS is a big challenge. One promising way to address this challenge is to apply MBT , which provides a systematic way to automatically derive test cases from models. However, traditional MBT methods are typically used for testing at the design time and must be extended to support testing Self-HealingCPS's FunctionalBehaviors, tightly integrated with PhysicalProcesses and runtime HealingBehaviors. To facilitate MBT for Self-HealingCPS, this section relates testing concepts (abstracted from the ISO/IEC 29119 Software Testing Standard) with the concepts from the Self-HealingCPS conceptual model. The ultimate goal is to define MBT techniques for testing self-healing behaviors of CPSs in the future work.

Figure 9 presents the Testing conceptual model, whereas the details of each concept are provided in Table 4 of Appendix. A Self-HealingCPS is the TestItem, which is tested by a set of TestCases. Each TestCase specifies the Preconditions of its execution, inputs and ExceptedResults. TestCases are executed in a TestEnvironment that simulates the actual operational Environment of the Self-HealingCPS. Such a TestEnvironment should include all facilities, hardware and software required to perform the testing of the Self-HealingCPS. In addition, numerous TestSituations are created by simulators, mocks or actual systems to imitate various Situations in the Self-HealingCPS's real operational Environment. Hence, as part of the TestEnvironment, the TestSituations are used to put the TestEnvironment in suitable states such that the TestCases can be executed.

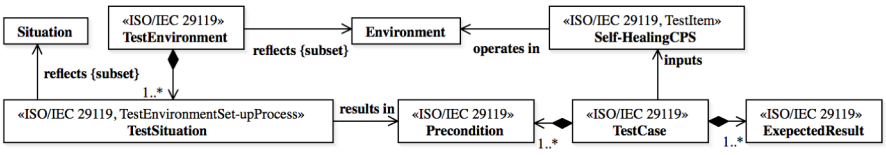

Figure 9. Testing Conceptual Model

9.2 Concept Definition

Complete Concept Definition for CPS

Concept	Definition	Attributes
C1. Self-HealingCPS	A CPS, which can autonomously detect, diagnose and recover from *Errors* (C18)	-type [1]: The *ArchitectureType* of the CPS -environment [1]: The *Environment* in which the CPS operates -physicalUnit [2..*]: Two or more *PhysicalUnits* (C3) constituting the CPS -network [1]: The *Network* (C4) via which all *PhysicalUnits* (C3) communicate with each other
C2. PhysicalProcess	A sequence of chemical, physical, or biological activities for the conversion, transport, or storage of material or energy	-interrelatedProcess [*]: A set of physical processes affected by this physical process.
C3. PhysicalUnit	A physical device that can communicate with the others, optionally having the computation and control capabilities	-sensor [*]: A *Sensor* (C5) monitors the variables of the *PhysicalProcess* (C2). -actuator [*]: An *Acutator* (C6) controls the *PhysicalProcess*. -controller [1..*]: *Controller* (C7) is responsible for monitoring and controlling the *PhysicalUnit*. -network [1]: *Network* (C4) that is used for communication with other *PhysicalUnits*.
C4. Network	The medium used as the communication channel among *PhysicalUnit*s (C3)	-sender [1..*]: A set of physical units that send data via the *Network* (C4). -receiver [1..*]: A set of physical units that receive data from the *Network*.
C5. Sensor	A device that measures the physical variables of a *PhysicalProcess* (C2)	-physicalProcess [1..*]: The *PhysicalProcesses* (C2) that the *Sensor* is supposed to monitor.
C6. Actuator	A device able to change physical quantities of a *PhysicalProcess* (C2)	-physicalProcess [1..*]: *PhysicalProcesses* (C2) that the *Actuator* is supposed to control.
C7. Controller	A software deployed on the *PhysicalUnit* (C3), controlling *Sensors* (C5) and *Effectors* (C23) either directly or indirectly, communicating with other *Controllers* and providing computational capability	-sensor [*]: A set of sensors controlled by the *Controller* to monitor the *PhysicalProcess* (C2) . -actuator [*]: A set of actuators controlled by the *Controller* to control the *PhysicalProcess* (C2) .

Complete Concept Definition for Self-Healing CPS

Concept	Definition	Attributes	Constraints
C8. Behavior	Describing a sequence of actions executed by a *Controller* (C7)	-fault [*]: A set of *Faults (C19)* in the behavior. If encountered, may cause one or more *Errors* (C18) [3].	None
C9. FunctionalBehavior	The business logic of a *Controller* (C7) [1]	None	None
C10. HealingBehavior	A sequence of *Self-Detection* (C17), *Self-Diagnosis* (C20) and *Self-Recovery* (C22) actions aiming at the recovery of a *Controller* (C7), a *PhysicalUnit* (C3) or the whole *Self-HealingCPS* (C1) from *Errors* (C18)	-level [1]: The *HierarchicalLevel* (C11) specifies the target of *HealingBehavior*. -type [1]: The *ApproachType* (C12) of the *HealingBehavior*. -goal [1..*]: A set of *Goals* (C13) that the *Controller* should satisfy during its execution. -self-Detection [1..*]: A set of *Self-Detection* (C17) actions used to detect *Errors* (C18). -self-Diagnosis [1..*]: A set of *Self-Diagnosis* (C20) actions used to locate *Fault* (C19) and generate *RecoveryPlan* (C21). -self-Recovery [1..*]: A set of *Self-Recovery* actions used to apply the *RecoveryPlan* (C21).	- If the *Self-Detection* (C17) or *Self-Diagnosis* (C20) has *Self-Learning* (C24) mechanisms, the *ApproachType* (C12) of the healing behavior should be *Dynamic*
C11. HierarchicalLevel	The subordination level within the structure of the *Self-HealingCPS* (C1) [3]	-SystemLevel: The highest level covering the whole Self-HealingCPS (C1) -PhysicalUnitLevel: The medium level with the scope of one *PhysicalUnit* (C3). -ControllerLevel: The lowest level with the scope of one *Controller* (C7).	None
C12. ApproachType	Indicating if a *HealingBehavior* (C10) is static or dynamic.	-Static: The kind of *Behavior* (C8) that is known at the design time and remains fixed. -Dynamic: The kind of *Behavior* (C8) that can be changed at the runtime.	None
C13. Goal	*"*... a non-operational objective to be achieved by the composite system" [4]	None	-Goals of *Controller* (C7) (or *PhysicalUnit)* (C3) should conform to *PhysicalUnit*'s (C3) goal (or *Self-HealingCPS's* (C1) goal).
C14. State	A particular combination of the attribute values of a *Controller* (C7) [8]	None	None
C15. Probe	A system measurement mechanism, which observes and measures the *States* (C14) of a *Controller* (C7) [9].	-state [1..*]: A set of *Controller*'s (C7) *States* (C14) that this probe is supposed to monitor -measurement [1..*]: A set of *Measurements* (C16) collected by the probe.	None
C16. Measurement	A value of a *State* variable, such as performance metrics	None	None
C17. Self-Detection	The action that detects *Errors* (C18) from monitored *Measurements* (C16) and reports the *Errors* (C18) to *Self-*Diagnosis [12]	-measurement [1..*]: A set of *Measurements* (C16) monitored by the *Self-Detection* action. -error [1..*]: A set of *Errors* (C18) detected or predicted by the *Self-Detection* action. -self-Learning [*]: The *Self-Learning* (C24) mechanism used by the *Self-Detection* to recognize system behavior *Patterns* (C25).	None

Concept	Definition	Attributes	Constraints
C18. Error	Difference between monitored *Measurement*s (C16) and specified ones [3]	None	None
C19. Fault	The cause of an *Error* (C18) [14]	-error [*]: A set of *Errors* (C18) that the *Fault* may lead to.	None
C20. Self-Diagnosis	The action that locates *Fault*s (C19), which lead to detected *Error*s (C18), and calculates an appropriate *RecoveryPlan* (C21) together with *RecoveryPolicy* (C39) [12]	-error [1..*]: The detected *Errors* (C18) reported from *Self-Detection* (C17). -fault [1..*]: A set of *Faults* (C19) that may lead to *Errors*. -recoveryPlan [1..*]: A set of *RecoveryPlans* (C22) generated by *Self-Diagnosis* to repair the *Fault*. -self-Learning [*]: The *Self-Learning* (C24) mechanism used by the *Self-Diagnosis* to infer *RecoveryPlan* (C21).	
C21. RecoveryPlan	A sequence of *AdaptationAction*s (C33), aiming to recover the CPS from *Error*s (C18)	None	None
C22. Self-Recovery	The action that applies a *RecoveryPlan* (C21) on the CPS [12]	-effector [1..*]: A set of *Effectors* (C23) used to execute the *RecoveryPlan*. -recoveryPlan [1..*]: A set of *RecoveryPlans* (C21) to be executed.	None
C23. Effector	A mechanism carrying out *AdaptationActions* (C33) [9]	-behavior [1..*]: A set of *Behaviors* (C8) that can be adapted by the *Effector*.	None
C24. Self-Learning	A mechanism that can recognize system behavior *Patterns* (C25) and/or infer *RecoveryPlan*s (C21) from historic *ExecutionResult*s (C43)	-measurement [1..*]: A set of *Measurements* (C16) used to analyze system's *Behaviors* (C8). -pattern [1..*]: A set of system behavior *Patterns* (C25) recognized from the *Measurements* (C16). -executionResult [1..*]: A set of *ExecutionResults* (C43) used to analyze the effect of RecoveryActions (C33). -recoveryStrategy [1..*]: A set of recovery strategies inferred from *ExecutionResults* (C43).	None

Complete Concept Definition for Self-Healing Behavior

Concept	Definition	Attributes
C25. Pattern	The mode/style of a system behavior characterized by a combination of *Measurement*s (C16)	None
C26. ErrorCriterion	The standard, rule, or test, on which a judgment of an *Error* (C18) can be based	-goal [*]: A set of *Goals* (C13) used to detect *Errors* (C18). -pattern [*]: The captured *Patterns* (C25) used to detect *Errors* (C18).
C27. Rule	A pair of a set of preconditions and a set of actions. If all preconditions are satisfied, then the actions are executed	Inherited from *ErrorCriterion* (C26)
C28. Threshold	A limit or boundary of *Measurement*s (C16) used to distinguish normal and abnormal values	Inherited from *ErrorCriterion* (C26)
C29. Constraint	A specification of system properties that the system must hold during its execution.	Inherited from *ErrorCriterion* (C26)
C30. PerformanceProbe	A *Probe* (C15) for monitoring system's performance.	Inherited from *Probe* (C15)
C31. EventProbe	A *Probe* (C15) for monitoring events occurred in *Self-HealingCPS* (C1)	Inherited from *Probe* (C15)
C32. PhysicalProcessProbe	A *Probe* (C15) for monitoring the state of a *PhysicalProcess* (C2)	Inherited from *Probe* (C15)
C33. AdaptationAction	The runtime modification of control data, controlling or affecting a *Controller* (C7)	-delay [1..*]: A set of possible *Delay* (C34) for this *AdaptationAction* to take effect -effect [1..*]: A set of *Effects* (C35) as the consequence of the action. -overhead [1..*]: A set of possible *Overhead* (C36) caused by the action.
C34. Delay	The time interval between the initiation and completion of an *AdaptationAction* (C33)	None
C35. Effect	The change of one or more *Behavior*s (C8) caused by an *AdaptationAction* (C33)	None
C36. Overhead	Extra resources and/or time used to execute *AdaptationAction*s (C33), in addition to planned resources and/or time at the design time	None
C37. ResourceOverhead	Extra resources for executing *AdaptationAction*s (C33), as compared to regular CPS resource consumption without adaptation	None
C38. TimeOverhead	Extra time for executing *AdaptationAction*s (C33), as compared to regular CPS response time without adaptation	None
C39. RecoveryPolicy	A type of formal behavioral guide for *HealingBehavior* (C10) [5]	-adaptationAction [1..*]: A set of *AdaptationActions* (C33) referred in the *RecoveryPolicy*.
C40. ActionPolicy	Specifying which *AdaptationAction*(s) (C33) should be taken for a *Fault* (C19) [5]	Inherited from *RecoveryPolicy* (C39)
C41. GoalPolicy	Specifying a set of desirable *States* (C14) of *Self-HealingCPS* (C1), providing the target states for *HealingBehaviors* (C10) [5]	Inherited from *RecoveryPolicy* (6)
C42. UtilityFunctionPolicy	Assigning each *State* (C14) of *Self-HealingCPS* (C1) a utility value directing the system moving towards states with a higher utility value [5]	Inherited from *RecoveryPolicy* (6)
C43. ExecutionResult	The consequence and the effect of a *RecoveryPlan* (C21)	None

Complete Concept Definition for Testing

Concept	Definition
C44. TestItem	Self-HealingCPS (C1) that is an object of testing [6]
C45. TestCase	A set of Preconditions (C46), invoked Behaviors (C8), and ExpectedResults (C47), developed to drive the execution of a test item to meet test objectives [6]
C46. Precondition	The prerequisites to execute a TestCase
C47. ExpectedResult	The expected consequence of invoked Behaviors (C8)
C48. TestEnvironment	"Facilities, hardware, software, firmware, procedures, and documentation intended for or used to perform testing of software." [6]
C49. TestSituation	A course of events used to simulate an actual Situation and stimulate the TestItem (C44)
C50. TestEnvironmentSet-upProcess	A sequence of TestSituations (C49) for establishing and maintaining the TestEnvironment (C48) to execute TestCases (C45) [6]