Targeting Uncertainty in Smart CPS by Confidence-Based Logic

Tomas Bures[a], Petr Hnetynka[a], František Plášil[a]*,

Dominik Skoda[a], Jan Kofroň[a], Rima Al Ali[a], and Ilias Gerostathopoulos[b]

[a]*Charles University in Prague, Faculty of Mathematics and Physics, Malostranske nam. 25, Prague 1, 11800, Czech Republic*
[b]*Vrije Universiteit Amsterdam, Faculty of Science, De Boelelaan 1105, 1081 HV Amsterdam, Netherlands*
* *Corresponding author. Tel.: +420951554266; e-mail: plasil@d3s.mff.cuni.cz*

ABSTRACT

Since Smart Cyber-Physical Systems (sCPS) are complex and decentralized systems of dynamically cooperating components, architecture-based adaptation is of high importance in their design. In this context, a key challenge is that they typically operate in uncertain environments. Thus, an inherent requirement in sCPS design is the need to deal with the uncertainty of data coming from the environment. Existing approaches often rely on the fact that an adequate model of the environment and/or base probabilities or a prior distribution of data are available. In this paper, we present a specific logic (CB logic), which, based on statistical testing, allows specifying transition guards in architecture-based adaptation without requiring knowledge of the base probabilities or prior knowledge about the data distribution. Applicable in state machines' transition guards in general, CB logic provides a number of operators over time series that simplify the filtering, resampling, and statistics-backed comparisons of time series, making the application of multiple statistical procedures easy for non-experts. The viability of our approach is illustrated on a running example and a case study demonstrating how CB logic simplifies adaptation triggers. Moreover, a library with a Java and C ++ implementation of CB logic's key operators is available on GitHub.

Keywords: Software architecture, adaptation, uncertainty, smart cyber-physical systems, statistical testing.

1. Introduction

Smart Cyber-Physical Systems (sCPS) are complex distributed decentralized systems of cooperating mobile and stationary devices, which closely interact with the physical environment. Examples of sCPS include smart home/office, smart cities, smart traffic, smart manufacturing, etc. The mobility aspect of sCPS, their openness, and potential open-endness bring about a high level of dynamicity to the system. Therefore, traditional software development techniques have shown not to be very suitable for developing such systems. Instead, novel approaches and techniques (e.g., [1], [2], [3]) have been proposed to address the related issues of sCPS software development.

Specifics of sCPS include the need for powerful means for dealing with environmental uncertainty, the necessity of coping with resource constrains, the strong emphasis on dependability, as well as architectural dynamicity and adaptation/reconfiguration. Since large sCPS are inherently composed of a number of components, applying software component models in sCPS software design is a natural choice, preferably those supporting adaptation at the architectural level – at least via component modes as a fundamental means for this purpose. Nevertheless, such component models should reflect the sCPS specifics mentioned above. An approach further in this direction is component ensembles introduced in the scope of the EU FP7 ASCENS project[1]. As a follow up of our participation in it, we have proposed several methods for dealing with uncertainty at the architectural level, such as meta-adaptation strategies [10], and architectural homeostasis [11]. Nevertheless, based on several experiments with architectural specifications, we have realized that the specifics of sCPS should also be addressed at the lower level of the *transition guards* – the conditions, written in a logic, in the control flow influencing dynamic architectural reconfiguration. In particular, the expression power of the employed logic should allow for handling uncertainty in the data sensed from the environment. The aim here is to make it possible to take a conservative (safe) decision when the data are noisy and/or incomplete.

Problem statement. Overall, in this paper, we assume that environmental uncertainty comes from the fact that data from the environment cannot be observed directly without noise and that a precise model of the environment cannot be obtained ("Sensing" uncertainty source in [12], "Uncertainty due to noise" in [13]). These two assumptions make it difficult to apply existing architectural approaches to adaptation, which often rely on the fact that an adequate model of the environment is available and that there are prior assumptions about the distribution of the data (e.g., in the form of base probabilities or a prior distribution). On the other hand, approaches that focus on sensing and diagnosis in architecture-based adaptive systems ([14]–[16]) do not specifically address the problem of making transition guards

[1] http://ascens-ist.eu/

robust in the face of noisy data. Finally, even though statistical methods and tests are generally well-known, it is typically hard for non-experts in statistics to apply them in engineering adaptive systems.

Goal. To tackle the problem, we propose a dedicated confidence-based logic (CB logic), which, by employing trends and historical observations, allows us to express the degree of confidence of a future event occurrences; it does this by making probabilistic conclusions about a system's environment. Conceptually, the historical observations are modeled by time series; CB logic provides a number of operators over the time series that simplify the filtering, resampling, and statistics-backed comparisons of time series, making the application of multiple statistical procedures easy for non-experts. We showcase CB logic on architecture-based self-adaptation in sCPS software – namely its application in transition guards controlling component modes and component ensembles. In detail, we rely on the DEECo component model [17], where architectural transitions are based on these concepts.

The paper is a substantially extended and generalized version of [18]; the modifications are primarily related to (a) an extension to the logic proposed in [18], (b) employment of the logic in transition guards in component ensembles, (c) a novel running example, experimental evaluation, an additional case study, and related work, (d) discussion about verification of transition guard conditions and about tuning the threshold values in transition guards.

The paper is structured as follows. In Section 2, we present a running example that is used as a motivation and for explanation and evaluation of novel concepts. In Section 3, we summarize the main idea of our approach to handling uncertainty in transition guards, whereas Section 4 brings up the core concepts and semantics of CB logic. In Section 5, we discuss several related issues in more detail, verification of transition guards and completeness and correctness in particular. Section 6 is devoted to experimental evaluation, whereas Section 7 discusses the related work, and, finally, Section 8 sums up the contribution of the paper.

2. Running Example and Transition Guards in Component Modes and Ensembles

2.1. RoboCup Rescue Simulations

As a motivation example, we present a scenario from the RoboCup Rescue Simulations (RCRS)[2]. In essence, RCRS targets the evaluation of rescue activities in a city after a disaster (such as an earthquake). In particular, there are (i) people injured inside the building ruins that need to be disengaged from the ruins and brought to a hospital, and (ii) buildings on fire to be extinguished. Since these events can be found all over the city, the undertaken rescue actions have to be highly coordinated. This represents the major challenge in this scenario. A natural way to tackle it is to see RCRS as an sCPS, a distributed and decentralized system of dynamically cooperating components (*agents* in the RCRS terminology).

Conceptually, RCRS introduces stationary agents (fire stations, police offices, hydrants, and ambulance centers) and platoon agents (fire brigades, police forces, and ambulance teams). To determine the location of agents, RCRS provides a map with streets, intersections, and buildings. Each agent has a dedicated task in the scenario. For instance, fire brigades are responsible for extinguishing fires, while police forces remove blocking debris off the streets to allow for the passing of other agents. Ambulance teams, as expected, are in charge of unburdening injured people and bringing them to refuges – special buildings collecting people before they are taken to a hospital (which is not a part of the scenario, though). An important aspect of this scenario is that the platoon agents have limited information about the overall situation, being able just to observe the situation in the nearest neighborhood. Thus, they need to be coordinated with other platoon agents to complement them on their tasks, not to compete with them.

The challenge is how to describe the behavior of the RCRS agents in the presence of uncertainty inherent to the environment. In particular, sensing under uncertainty (i.e., noisy or imprecise readings) will impact the way the agents make decisions at run-time about transitions among their actions and about how to cooperate.

[2] http://roborescue.sourceforge.net/

```
role RefillProviderRole:
  mode, id, position, freeStand

role RefillConsumerRole
  mode, id, water, position, refillTarget

role FireBrigadeRole:
  mode, id, water, position, helpTarget,
  helpingFireBrigades, burningBuildings

component type Hydrant features
    RefillProviderRole
  knowledge:
    mode {vacant, occupied}
    id
    position
    freeStands
  ...

component type FireStation features
    RefillProviderRole
  knowledge:
    mode {vacant, occupied}
    id
    position
    freeStands
  ...

component type FireBrigade features
    RefillConsumerRole, FireBrigadeRole
  knowledge:
    mode {refill, search, extinguish,
          moveToFire, moveToRefill}
    id
    water
    position,
    refillTarget
    helpTarget,
    helpingFireBrigades,
    burningBuildings,
    closestBurningBuilding
    buildingsInRange

  function existsBurning(temperature,
   timeWindow):
    for building in buildingsInRange:
        if above(fireSensor(building),
                 temperature,
                 timeWindow):
            burningBuildings.add(building)
      closestBurningBuilding =
            closest(burningBuildings)
    return not burningBuildings.isEmpty()

function isWaterLevel(relation, W, TIME_WINDOW):
  switch relation:
    case above:
      return above(water, W, TIME_WINDOW)
    case below:
      return below(water, W, TIME_WINDOW)

process refillProc in mode refill
  if isWaterLevel(above, $W_{max}$, TIME_WINDOW):
    if burningBuildings.size > 0:
      mode = moveToFire
    else
      mode = search
  else
    refill()

process moveToFireProc in mode moveToFire
  if existsBurning($Temp_{min}$, TIME_WINDOW):
    mode = extinguish
  else
    // Sorted burning buildings as in
    // extinguishProc
    move(burningBuildings.last)

process moveToRefillProc in mode moveToRefill
. . .
process searchProc in mode search
  if existsBurning($Temp_{min}$, TIME_WINDOW):
    mode = extinguish
  else
    move()

process extinguishProc in mode extinguish
  if isWaterLevel(below, $W_{min}$, TIME_WINDOW):
    mode = moveToRefill
  elif not existsBurning($Temp_{min}$, TIME_WINDOW):
    mode = search
  else
    sort(burningBuildings,
      (building1, building2) => compare(
     afv(occupancyHistory(building1, week), day, 0),
     afv(occupancyHistory(building2, week), day, 0)))
    building = burningBuildings.last
    extinguish(building)
    if below(fireSensor(building),
             PUT_OUT_TEMPERATURE,
             TIME_WINDOW):
      burningBuildings.remove(building)
      building = closest(burningBuildings)

process recovery in mode default
. . .
```

Figure 1. A fragment of RCRS components in DSL. The CB logic operators are typed in *`italic`*.

2.2. Modeling the example as components with modes and ensembles

The RCRS scenario can be modeled using the DEECo component model [9]. In DEECo, agents are modeled as components that operate under different modes and form ensembles (cooperation groups). A fragment of the scenario, specified in a Domain Specific Language (DSL) akin to the one of DEECo, is in Figure 1.

Components. A component is, as traditionally, defined by its state, activities, and communication contracts. The state is given by a list of data fields (*knowledge* in DEECo). The activities are modeled as *processes* (tasks). These are periodic or event-triggered (such as a mode transition) and typically involve sensing, computation, modification of knowledge, and actuating—not necessarily in that order. A communication contract is based on the concept of a component *role* which lists the knowledge fields that reflect the specific parts of the component's functionality that participate in its communication with other components. A component may feature several roles – their list is specified in the component type definition. For instance, `FireBrigade` features `RefillConsumerRole` and `FireBrigadeRole`. The mechanism of component instantiation and its processes' periodic/triggered activation is beyond the scope of this paper (and not important for its message).

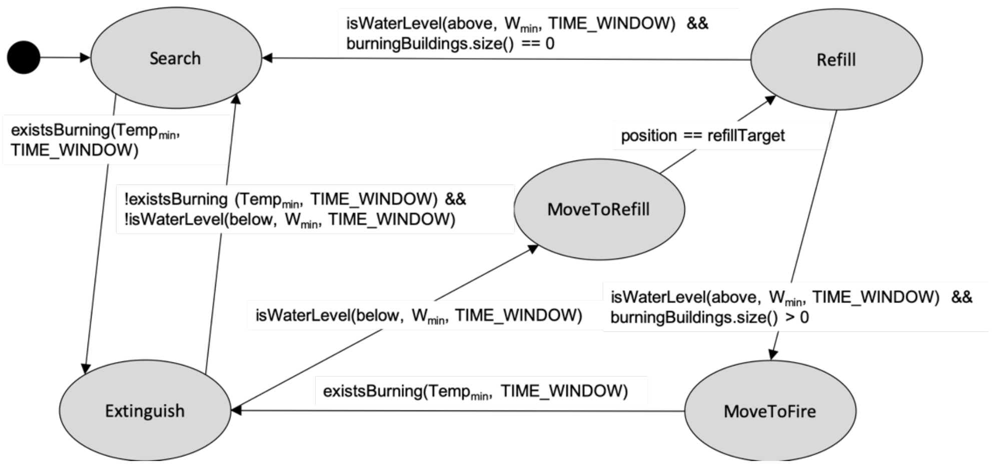
Search
Refill
isWaterLevel(above, W_min, TIME_WINDOW) &&
burningBuildings.size() == 0
existsBurning(Temp_min, TIME_WINDOW)
position == refillTarget
!existsBurning (Temp_min, TIME_WINDOW) &&
!isWaterLevel(below, W_min, TIME_WINDOW)
MoveToRefill
isWaterLevel(above, W_min, TIME_WINDOW) &&
burningBuildings.size() > 0
isWaterLevel(below, W_min, TIME_WINDOW)
existsBurning(Temp_min, TIME_WINDOW)
Extinguish
MoveToFire

Figure 2. Mode state machine of FireBrigade. W_{max} is a water level above which the tank is considered full; W_{min} is a water level below which the tank is considered empty; I is the time interval (scenario specific) of the time series in use.

In support of self-adaptation, a component's activities are controlled by its *active mode*, which is represented by one of the mode values specified in the component's knowledge (**mode** denotes the knowledge field that keeps the active mode of the component). The active mode (i) determines the process(es) currently active in the component, and (ii) can influence the component's membership in an ensemble, since mode testing may appear in a membership condition. For instance, in Figure 1, the activities of `FireBrigade` are defined by six processes, each of them being executed in a specific active mode; e.g., `refillProc` is executed when the active mode is `refill`. For simplicity, Figure 1 illustrates the situation in which each process is executed in exactly one active mode; in general, a process can be executed in multiple modes.

To describe mode switching (modification of active mode values), the component type specification is associated with a mode state machine at design time. In Figure 2, there is the mode-state machine of `FireBrigade`. During development, the mode switching functionality is directly encoded in the processes of a component. The `FireBrigade` instance onsets activities in the `search` mode, aiming at finding a building on fire to be extinguished. If the `FireBrigade` finds such a building, it moves to the fire (`moveToFire` mode), and once it is close to the scene, it starts extinguishing (`extinguish` mode). If the `FireBrigade` gets low on water while extinguishing (checked via the *below* predicate explained in Section 3.1), it moves to a refill point (mode `MoveToRefill)`, and finally refills its tank (`refill` mode). After the fire has been put out, the `FireBrigade` starts searching for another building on fire or seeks a refill point, etc.

Ensembles. The members of an ensemble instance are dynamically determined by a periodical evaluation of its membership condition – transition guard. There are two types of ensembles in the example in Figure 3: (i) the `TargetFireZoneEnsemble` grouping several `FireBrigade` instances to coordinate their activities close to a fire zone,

```
ensemble type RefillStationEnsemble
  roles
    RefillProviderRole coord
    RefillConsumerRole memb
  membership condition
    vacant == coord.mode &&
                        distance(coord.position, memb.position) <distance(memb.refillTarget, memb.position)
  knowledge exchange
    memb.refillTarget = coord.position

ensemble type TargetFireZoneEnsemble
  roles
    FireBrigadeRole coord
    FireBrigadeRole memb
  condition
    //member is close and fire is spreading
    coord.helpingFireBrigades.size() < MAX_HELP_CNT &&
      extinguish == coord.mode && (search == memb.mode || moveToFire == memb.mode) &&
      distance(coord.position,memb.position) < T_D &&
      fabove(burningBuildings.size,THRESHOLD, TIME_WINDOW, currentTime + observationStep)
  knowledge exchange
    memb.helpTarget = coord.position
    coord.helpingFireBrigades.add(memb.id)
```

Figure 3. A fragment of RCRS ensembles in DSL. The CB logic operators are typed in *italic*.

and (ii) the `RefillStationEnsemble` which communicates information on the nearest refill place between a refill consumer (`FireBrigade`) and a refill provider (`FireStation` or `Hydrant`). Each ensemble instance has its coordinator component and a set of member components. Specifically, any component instance featuring the role `RefillProviderRole` triggers instantiation of an ensemble of the type `RefillStationEnsemble` (becoming thus its coordinator) and the component instances featuring the role `RefillConsumerRole` and satisfying the membership condition of `RefillStationEnsemble` will become the members of this ensemble instance. As an aside, the membership condition of an ensemble may require its member components or coordinator to be in a particular active mode. For example, in `TargetFireZoneEnsemble` the active mode of member components is required to be either `search` or `moveToFire` (tested by `moveToFire == memb.mode || search == memb.mode`). The coordinator periodically triggers knowledge exchange among its knowledge and the knowledge of the members. As to a `TargetFireZoneEnsemble` instance, its coordinator is the first `FireBrigade` being on the scene; it then coordinates the activities of the member `FireBrigade` instances. Again, a detailed description of ensemble instantiation and knowledge exchange is beyond the scope of this paper (and not important for the message it aims to deliver).

Self-adaptation. Both component ensembles and component modes are mechanisms of architectural self-adaptation; these mechanisms are based on the evaluation of transition guards (i.e., the membership conditions and conditions labeling transitions of mode state machines) [11]. Conceptually, each membership condition and each mode state machine defines a self-adaptation strategy in the sense of [19]. These strategies follow the MAPE-K loop [20], in which variables in component knowledge that appear in the transition guards are Monitored, their values then Analyzed, the corresponding actions are Planned, and finally Executed. The membership condition strategy is applied by the run-time framework by periodically evaluating its membership condition (M and A phases of MAPE-K) and the consequent modification of components' membership in the ensemble (P and E phases); consequently, this causes a dynamic modification to the current software architecture). The mode switching strategy is applied by both the components themselves (M and A phases) and the run-time framework (P and E phases). In particular, each component monitors the data needed for evaluating mode transitions, evaluates the relevant transition guards, and outputs the new mode to be applied. The run-time framework applies the new mode by activating the associated processes, which changes the current software architecture.

3. Handling uncertainty in transition guards – the essence of the approach

3.1. Employing times series via CB logic

Coping with uncertainty directly in transition guards needs to consider the fact that when a transition guard is evaluated, the result can depend on the changes in the environment. These changes are detected in components by sensor readings; e.g., by a sensor measuring the water level in the tank, sensor detecting fires, etc. Since the accuracy of sensors is limited and the actual readings may be influenced by additional factors (wind, darkness, sudden obstacles in the line of direct sight, etc.), the related data noise has to be echoed in a well-thought-out way in the transition guards. A cornerstone of our approach to deal with this kind of uncertainty is the ability to consider the historical development of observed knowledge values. This is important not only to filter out temporary disturbances but also to predict trends in the observed data. To this end, we use time series instead of single-valued data (as would be the case in the traditional approach to expressing transition guard conditions). On top of the time series, we provide several operators (forming the core of CB logic) to perform statistical reasoning and to construct expressions that can be used in transition guards.

For example, in Figure 1, lines 47 and 65, such an operator – *above*(…) is used. In general, *above*(x, y, w) means that with 95% confidence the current value of x is greater than y, evaluated over the past time interval of length w. Thus, the construct `above(fireSensor(building), temperature, timeWindow)` means that with 95% confidence the value of the fire sensor reading for `building` is greater than `temperature` based on the linear regression over the readings in `timeWindow`. Employing similarity in data patterns, in Figure 1, lines 102-103, the operator *afv*(A,s,f) returns the value x forecasted in time now + f by the ARIMA model (given the time series A, and using a seasonal pattern of size s). Thus, the construct `afv(occupancyHistory(building1, week), day, 0)` provides an estimate of the current occupancy of `building1` based on the history of this value over the last month. Precise definitions of all CB logic operators are in Section 4.

3.2. Default mode method

A way to cope with uncertainty in mode transition guards when data for employing time series are not available is to introduce a default active mode (represented by the mode value `default`). Thus, technically similar to exception handling, it is assumed that in any state of the mode state machine, there is a transition to the default mode guarded by the conjunction of negated guards of all the other transitions from the state. This is specified implicitly, so that not directly visible from the state machine diagram. Nevertheless, in Section 4.1, we show a way to analyze the cases when the default mode is applied, since this may be a source of potential errors in design that may demonstrate themselves as late as at run-time.

3.3. The proposed approach in a nutshell

In our running example, we have seen how uncertainty on the events happening in the environment is captured in components and ensembles in DEECo – in essence, this is done by employing specific operators of CB logic in its self-adaptation transition guards. Our approach is that these operators are defined upon time series and allow, with defined confidence, to make decisions in the transition guards based on the historical development of sensor-observed (noisy) data, and even predict the development of their future values. This is achieved via combining CB logic operators in concise language constructs (e.g., *below*()) that allow even non-experts in statistics to easily perform filtering, resampling, and statistically backed comparisons in time series. Moreover, it should be emphasized that these operators are also provided with variants of a higher level of abstraction to support the ease of use of these operations even further.

From the self-adaptation perspective, these operators are applied in the Monitoring and Analysis phases on the MAPE-K loop. Even though in this section we rely on ensembles and mode-state machines as the basic modeling formalisms, the use of CB logic operators is not limited to these DEECo self-adaptation concepts. Rather, they can be applied in any self-adaptive system where statistical reasoning on noisy data are appropriate to employ in the Monitoring and Analysis phases of the MAPE-K loop.

Nevertheless, CB logic is applicable in state machines' transition guards in general, wherever the need for addressing uncertainty occurs and related data, forming time series, are available.

4. Confidence-Based logic

In this section, we describe the Confidence-Based (CB) logic, which forms the backbone of our uncertainty-handling approach.

4.1. Definitions

Formally, we define CB logic as a many-sorted logic in the following way:

Variables $A = A_1, \dots, A_n$; B; $C, \dots$ are time series. We denote $T(A) = T(A_1), \dots, T(A_n)$ the series of time when $A_1, \dots, A_n$ was sampled.

In addition to the standard logical connectives, we add the following operators that return a time series:

Selection and resampling

- Selection $A_i \dots A_j$ denoting a sequence consisting of elements A_i through A_j. This can also be one element time series – e.g., A_1 denotes a time series where only the first element has been preserved.
- Selection $[A]_{t1}^{t2} = A_l \dots A_r$ such that $l = \min \{i | T(A_i) \geq t1\}$ and conversely $r = \max \{i | T(A_i) \leq t2\}$. We call this selection an $(t1, t2)$ window over A.
- Selection $[A|\Phi]$ denoting a sub-series containing only those A_i for which $\Phi(A_i)$ is true.
- Resampling $A \sim T(B)$ of a time series A to sample times $T(B)$ by linear interpolation.
- Resampling $A \eqsim T(B)$ of a time series A to sample times $T(B)$ by using the closest previous value.

Operators over time series

- Basic arithmetic over time series (assumes that $T(A) = T(B)$, and $c \in \mathbb{R}$):
  - cA – multiplication of each element by a constant
  - $A + c$ – addition of a constant
  - $A + B, A - B$ – element-wise addition/subtraction
  - $A \cdot B$ – element-wise product
  - $=$ – equality of two time series

  Operators that return a scalar

  - $\min(A)$ – returns the minimum value of the time series A

- max (A) – returns the maximum value of the time series A

  Operators that return a cumulative distribution function F

  - $mean(A)$ – distribution used for comparing the sample mean of A.

  Technically (in the light of Section 4.4), it is a distribution of sample means of the time series elements under the hypothesis that the mean is the sample mean of A. The resulting distribution serves as a plug-in distribution used for evaluation of $\leq_\gamma, \ldots$ relational operators (defined below) by means of hypothesis testing. The quantiles of this distribution are used to establish p-values for the test. For $mean(A)$, we assume that the samples are i.i.d. random variables with a normal distribution $mean(A)$ is thus a shifted and scaled Student's t-distribution (see Section 4.4). Note that A in this case, and in the rest of the section, is potentially a window over a time series A, above denoted as $[A]_x^y$.

  Linear regression operators

  - $lra(A)$ – distribution used for comparing the value of intercept $\hat{\alpha}$ in linear regression $\hat{\alpha} + \hat{\beta}t$ fitted to the time series A via ordinary least squares (OLS)[3].

  Technically, it is a distribution of intercepts $\hat{\alpha}$ under the hypothesis that the true intercept is the one estimated by OLS.

  - $lrb(A)$ – distribution used for comparing the slope $\hat{\beta}$. Technically, it is a distribution of slopes $\hat{\beta}$ under the hypothesis that the true slope is the one estimated by OLS.
  - $lr(A, t)$ – distribution of $y = \hat{\alpha} + \hat{\beta}t$ as above. (Note that since $\hat{\alpha}$ and $\hat{\beta}$ are random variables, y is a random variable as well.)

Traditional relational operators over scalars

- $x \leq y, x = y, \ldots$ – if scalar x is less than y, equal to y, etc.

Relational operators over the distribution functions realized by statistical testing

- $F \leq_\gamma c, F_1 \leq_\gamma F_2$ – if the null hypothesis $X \leq c$ or $X_1 \leq X_2$ respectively cannot be rejected at confidence level γ, where X, X_1, X_2 are random variables with distributions F, F_1, F_2 respectively.
- $F <_\gamma c, F_1 <_\gamma F_2$ – if the null hypothesis $X \geq c$ or $X_1 \geq X_2$ respectively can be rejected at confidence level γ, where X, X_1, X_2 are as above.
- $\geq_\gamma$ and $>_\gamma$ are defined correspondingly.
- $=_\gamma$ is defined as $\leq_\gamma$ & $\geq_\gamma$

Additionally, we include the standard logical operators $\&, \vee, \neg$.

4.2. Examples

With this apparatus in hand, we can express transition guards such as:

- $mean([tmp]_{now-10s}^{now}) <_{0.95} 20$ – with confidence 95% the expected value of temperature tmp in the past 10 seconds has been lower than 20 degrees. Note that tmp is the original time series A, $[tmp]_{now-10s}^{now}$ is its (now-10s, now) window, $M = mean([tmp]_{now-10s}^{now})$ is the distribution of the sample mean of the window, and $M <_{0.95} 20$ denotes that the null hypothesis $M \geq 20$ can be rejected at confidence level 0.95.
- $mean([tmpB \sim T(A)]_{now-10s}^{now}) - 20 <_{0.95} mean([tmpA]_{now-10s}^{now})$ – with confidence 95% the expected value of temperature $tmpA$ has been in the last 10 seconds at least by 20 degrees lower than the expected value of $tmpB$. Note that $tmpB$ is first resampled to sample times of $T(A)$ by linear interpolation.

This assumes that within the 10-seconds window, the temperature is constant, and the measurement is subject to a normally distributed error with constant mean and variance. If it is assumed that the samples within the window have a

[3]The ordinary least squares (OLS) estimation for $y = \hat{\alpha} + \hat{\beta}t$ is given by $\hat{\beta} = \frac{\sum_{i=1}^{n}(t_i - \bar{t})(y_i - \bar{y})}{\sum_{i=1}^{n}(t_i - \bar{t})^2}$, $\hat{\alpha} = \bar{y} - \hat{\beta}\bar{t}$.

linear trend, the comparison can be executed based on statistics of confidence intervals obtained from the linear regression.

- $lr([wat]_{now-10s}^{now}, now) <_{0.95} 100$ – with confidence 95%, the current expected value of the water level is less than 100 liters evaluated over the past 10 seconds.

The lr operator can also be exploited to predict values in the near future based on the current linear trend. Of course, care has to be taken in interpreting the confidence level of the extrapolation as the confidence speaks only about the current trend, not about its accuracy in predicting the future.

- $lr([wat]_{now-60s}^{now}, now + 120s) >_{0.95} 100$ – based on the trend observed in the past minute, the value of the water level after 120 seconds from now will be, with the confidence of 95%, over 100 liters.

Generally, the t parameter of $lr(A, t)$ can be arbitrary; however, it is necessary to remember that the variance of the prediction grows with the distance of t from the mean of $T(A)$. The confidence bounds around $\hat{\alpha} + \hat{\beta}t$ form the usual "hourglass" shape. This means that the statistical test used in the comparison will be the strongest roughly in the middle of the window and will get weaker towards the boundaries of the window (see Section 4.4 where these underlying mathematical formulas are given).

The $mean$ and lr operators can also be exploited to approximately describe that a value is increasing/decreasing:

- $mean([wat]_{now-10s}^{now}) <_{0.95} mean([wat]_{now-70s}^{now-60s})$
- $lr([wat]_{now-10s}^{now}, now - 10s) <_{0.95} lr([wat]_{now-10s}^{now}, now)$

However, the use of $mean$ is often incorrect as it assumes no trends in the observation window. The use of lr is correct, but does not permit to easily reason about the rate of decrease/increase. Better control is achieved by lra and lrb, which expose the distribution of the linear regression coefficients

- $lrb([wat]_{now-10s}^{now}) <_{0.95} - 1$ means that with confidence 95%, the water level decreases faster than $f(t) = -t$.

4.3. Difference from traditional semantics of relational operators

Note that the statistical nature of the relational operators brings a few unexpected features. In particular, it does **not** hold that:

a) $F_1 \leq_\gamma F_2 \,\&\, F_2 \leq_\gamma F_3 \Longrightarrow F_1 \leq_\gamma F_3$

b) $F \leq_\gamma c_1 \,\&\, c_2 \leq_\gamma F \,\&\, c_1 \leq c_2 \Longrightarrow c_1 = c_2$

The violation of (a) is caused by the fact that while the distance between F_1, F_2 and F_2, F_3 was small enough to prevent the hypothesis test from rejecting (i.e., $\leq_\gamma$ is true), the distance between F_1, F_3 may be big enough to allow the test to reject, thus evaluating $\leq_\gamma$ to false.

The violation of (b) has a similar cause – c_1, c_2 are close to the mean of F; thus, no rejection takes place. However, c_1 may lie below the mean of F while c_2 may be above it (though $F \leq_\gamma c_1$ & $c_2 \leq_\gamma F$).

The violation of (b) nevertheless leads to an elegant way of expressing uncertainty:

- $lr([wat]_{now-10s}^{now}, now) <_{0.95} 100$ – "provably low water level, drive to the hydrant"
- $lr([wat]_{now-10s}^{now}, now) >_{0.95} 100$ – "provably sufficient water level, move to fire"
- $lr([wat]_{now-10s}^{now}, now) =_{0.95} 100$ – "nothing definite can be said about the water level, monitor area around"

Note that this is a rather significant difference from the traditional semantics of relational operators with real-valued quantities. There the likelihood that a real-valued observation is exactly equal to a certain quantity is extremely low (in fact, it is 0). In our interpretation, the equality $F =_{0.95} 100$ means "it cannot be shown by a statistical test with enough confidence that a mean is strictly higher or strictly lower". Thus, the mean is not compared exactly to the number 100, but to a confidence interval around 100. The confidence interval widens with an increasing variance of samples. Consequently, the likelihood that $F =_{0.95} 100$ may be rather high.

4.4. Evaluation of the relational operators

The interpretation we use for $F \leq_\gamma c$ and related operators rely on checking whether the value c lies within the confidence bounds given by F, i.e., the $1 - \gamma$ and γ quantiles of F.

Recall that F denotes here the distribution of the statistical quantity under the hypothesis that the true mean is the sample mean. In the case of the $mean(A)$ operator, it is a distribution of $\bar{A} + sT/\sqrt{n}$, where T is a random variable with Student's t-distribution with $|A| - 1$ degrees of freedom, $\bar{A}$ is the sample mean of A and s^2 is a sample variance of A.

In case of the $lrb(A)$ operator, assuming the model $\alpha + \beta t + \epsilon$ and normality of the error terms ϵ, the $lrb(A)$ is a distribution of $\hat{\beta} + s_{\hat{\beta}} t$, where $\hat{\beta}$ is the ordinary least square estimate of the slope, $s_{\hat{\beta}}$ is the standard error of the estimator $\hat{\beta}$, and t has Student's t-distribution of $|A| - 2$ degrees of freedom. A similar relation holds for the distribution of intercept $\hat{\alpha}$ returned by $lra(A)$ and the prediction $\hat{\alpha} + \hat{\beta} t$ returned by $lr(A, t)$.

The actual test $F \leq_\gamma c$ is interpreted as:

$$F \leq_\gamma c \quad \Leftrightarrow \quad F(2\mu - c) \leq \gamma$$

where μ is the mean of F and $F(t)$ denotes the cumulative distribution function of F.

The expression $2\mu - c$ is derived from the fact that we shift F such that its mean is c, which is the null hypothesis. This yields a distribution $F - \mu + c$. We then reject the null hypothesis if μ is greater than the γ quantile of $F - \mu + c$. With a few trivial rearrangements, we arrive at the $F(2\mu - c) \leq \gamma$.

The test $F_1 \leq_\gamma F_2$ is interpreted as:

$$F_1 \leq_\gamma F_2 \quad \Leftrightarrow \quad (F_1 - F_2)(2\mu_1 - 2\mu_2) \leq \gamma$$

where μ_1, μ_2 denote the means of F_1, F_2 respectively; $(F_1 - F_2)$ denotes the distribution of subtraction of random variables $X_1 - X_2$ where random $X_1 \sim F_1$ and $X_2 \sim F_2$.

The expression is derived in a similar way as above. We shift each of the two distribution to have mean 0 and subtract them: $F_1 - \mu_1 - F_2 + \mu_2$. This forms a distribution for the null hypothesis that the mean of $F_1 - F_2$ is less or equal to 0. We reject if $\mu_1 - \mu_2$ is greater than the γ quantile of $F_1 - \mu_1 - F_2 + \mu_2$.

4.5. Forecasting the future by ARIMA

For recurrent events, where linear regression may not be an appropriate approximation, we provide more fitting handling in terms of ARIMA models [21]. ARIMA stands for AutoRegressive Integrated Moving Average and is one of the statistical methods for time series analysis. Its primary purpose is to forecast future values of a given time series. A necessary condition is that the data has to exhibit certain regularity in terms of recurring sequences of values of the same length. Those sequences are referred to as *windows*. Corresponding values in different windows are not required to be exactly the same – they can exhibit linear movement in time (called *trend*) as well as changes to variation inside a particular window. When choosing an appropriate length of the window – which can be rather challenging in some cases – ARIMA can provide a precise forecast of a future value with a known confidence level. To support this form of forecasting in our approach, we provide the following operators:

- $afv(A, s, f)$ – returns the value forecasted by the ARIMA model given the time series A, using a seasonal pattern of size s, and forecasting the future value of x in time $now + f$.
- $afc(A, s, f)$ – returns the confidence of the forecast of the future value of x in time $now + f$.
- $afd(A, s, f)$ – returns the distribution of the value in time $now + f$ forecasted by the ARIMA model given the time series A, using a seasonal pattern of size s.

4.6. Short-hand notation

To simplify the application of the operators defined above, we provide a short-hand notation for the most common cases. These include the *above* and *below* operators we have already seen in Figure 1.

- $below(x, y, w) \Leftrightarrow lr([x]_{now-w}^{now}, now) <_{0.95} y$ … with confidence 0.95%, the current value of x is less than y. It assumes that there is a linear trend (potentially a zero trend) in x over the past time interval of length w.
- $above(x, y, w) \Leftrightarrow lr([x]_{now-w}^{now}, now) >_{0.95} y$ … similar as above with x greater than y.
- $fbelow(x, y, w, f) \Leftrightarrow lr([x]_{now-w}^{now}, now + f) <_{0.95} y$ … the future value of x in time f from now is expected to be less than y.
- $fabove(x, y, w, f) \Leftrightarrow lr([x]_{now-w}^{now}, now + f) >_{0.95} y$ … similar as above with the future x greater than y.

Similarly, we provide short-hand notation for min/max:

- $maxfbelow(x,y,f) \Leftrightarrow \max\left([x]_{now}^{now+f}\right) < y$ … the maximum of the time series over the next time f is less than y
- $minfabove(x,y,f) \Leftrightarrow \min\left([x]_{now}^{now+f}\right) > y$ … similar as above with minimum future x less than y.

5. Discussion

In this section, we will discuss in more detail the issues already raised in the previous sections. Specifically, these are (i) verification of completeness of transition guards in mode-switching state machine, (ii) automated learning of particular parameters for transition guards, and (iii) options for more robust statistics to deal with outliers.

5.1. Verification of mode transition guards

When designing the modes switching state machine of a component, an important aspect is the completeness of the mode transition guards. If none of the conditions guarding the outgoing transition of a particular state are satisfied, the default mode is chosen; note here, that its guard (implicitly formed as a conjunction of negated guards of all the other transitions) is satisfied only if there is no other guard satisfied. This brings two consequences: (1) the default mode guard can be unsatisfiable, which corresponds to the case where the other guards in the state machine completely cover the option space, and, on the other hand, (2) satisfying the default mode guard can serve as an alarm that the designer has missed an option that should be covered explicitly by a mode transition. An example of (1) can be the situation, where two explicit mode transitions are guarded by conditions $x \neq 1$ and $x \neq 2$, respectively. This results in the transition guard of the default mode containing the (clearly unsatisfiable) subexpression $(x = 1) \wedge (x = 2)$. Note that this is an expected situation,

$$
\begin{aligned}
&\neg\Big[\big((mode = SEARCH) \wedge (existsBurning(Temp, TIME\_WINDOW)\big) \vee \\
&\big((mode = SEARCH) \wedge (\neg existsBurning(Temp, TIME\_WINDOW\,))\big) \vee \\
&\Big((mode = REFILL) \wedge \big(isWaterLevel(above, WMAX, TIME\_WINDOW)\big) \wedge (burningBuildingSize = \ 0)\Big) \vee \\
&\Big((mode = REFILL) \wedge \big(isWaterLevel(above, WMAX, TIME\_WINDOW)\big) \wedge (\neg burningBuildingSize = 0)\Big) \vee \\
&\Big((mode = REFILL) \wedge \big(\neg isWaterLevel(above, WMAX, TIME\_WINDOW)\big)\Big) \vee \\
&\Big((mode = MOVETOFIRE) \wedge \big(existsBurning(temp, TIME\_WINDOW)\big)\Big) \vee \\
&\Big((mode = MOVETOFIRE) \wedge \big(\neg existsBurning(temp, TIME\_WINDOW)\big)\Big) \vee \\
&\Big((mode = EXTINGUISH) \wedge \big(isWaterLevel(below, \mathrm{WMIN}, TIME\_WINDOW)\big)\Big) \vee \\
&\Big((mode = EXTINGUISH) \wedge \big(\neg isWaterLevel(below, \mathrm{WMIN}, TIME\_WINDOW)\big) \wedge \\
&\qquad \big(\neg existsBurning(Temp, TIME\_WINDOW)\big)\Big) \vee \\
&\Big((mode = EXTINGUISH) \wedge \big(\neg isWaterLevel(below, \mathrm{WMIN}, TIME\_WINDOW)\big) \wedge \\
&\qquad \big(existsBurning(Temp, TIME\_WINDOW)\big)\Big)\Big]
\end{aligned}
$$

Figure 4: Part of the formula for mode switching verification (corresponding to Fig. 1).

since, assuming x being a scalar variable always taking a value, x is always different from either 1 or 2 or both. In Figure 4, the formula used for verification of the uncovered cases for the example in Section 2.2 is listed. Each line of the formula is a transition guard used for mode switching in Figure 2, while the disjunction of the guards is negated to discover uncovered cases. In the formula, $mode$ and $burningBuildingSize$ are treated as integer variables, while $SEARCH$, $REFILL$, and other capitalized strings are integer constants. Operators, such as $isWaterLevel$ and $existsBurning$, are treated as independent Boolean variables for each given combination of parameters (there is just one combination for each operator in our example). A call to an SMT solver (we employ the Z3 SMT solver [22]) provides a set of satisfying assignments, each corresponding to an uncovered combination of the variables' values. It is then up to the system designer to decide which of them are intentionally left out to be covered by the default mode, and which of them have been omitted by mistake. We believe that such a check can reveal unforeseen situations since the number of the variable values' combinations in complex systems can be enormous. To cope with a high number of found variable assignments, grouping them by the mode and further filtering (based on domain knowledge and a particular application) can significantly help handling them. As to our example, the SMT call resulted in 7 uncovered variable values' combinations, all of them corresponding to missing options in the default mode. Since the default mode is not specified, we can conclude that our specification is complete in terms of the guard specification.

Note that even though the default-mode guards can be filtered out in an automatic way, it is still beyond the abilities of automated formal methods to decide about a particular situation whether it is a real flaw or a spurious one. As to the limits of the SMT solver, we assume that the sizes of formulae to verify are by orders of magnitude less than what the solver can handle (typically tens of thousands of variables); hence, this is not an issue here.

5.2. Thresholds tuning

Setting up optimal threshold values (e.g., distance to a building to start its extinguishing) within the mode switching and ensemble membership condition specification may be a tricky task, especially because of the system complexity. Below we describe a method based on simulation to tackle the problem. The method consists of two steps: (1) run-time data about the system behavior are collected, and (2) the specification is modified according to an optimization goal and then, together with the run-time data, is used for simulation.

As an example, consider that the distance to a burning building (involved in transition guard to the `extinguish` mode) is to be optimized. We would like to set the threshold distance value as far as possible while still having a reasonable efficiency of the extinguishing process, i.e., to ensure that the stream of water hits its target. In general, it is clear that optimal values strongly depend on the current environment state, e.g., on the visibility conditions and fire intensity; that is why run-time data on the current environment conditions have to be employed.

The simulation approach (see also Section 6) may not yield good results due to unforeseen modifications of the environment factors since this approach requires a relatively precise model of the environment taking account the physical laws governing these factors; therefore, a purely run-time solution turns out to be a more viable way in many cases.

In practice, multiple simulation runs need to take place with different settings of threshold values. The results of these simulations are to be evaluated to determine how a particular threshold value influences the performance of the system. In order to find an optimal value of a variable, an algorithm searching the space of sensible values is to be employed, e.g., simulated annealing. These simulations can be repeated for all the variables of interest, and, finally, the system setup with these values is to be tested again.

Some threshold values may be context-dependent, e.g., the value that increases system performance may reduce its robustness. Thus, the choice of particular values depends on the context and system goal. For instance, setting the distance threshold value to a building on fire depends on the wind speed – the stronger the wind, the closer the fire brigade has to be to reach the fire with a stream of water. On the other hand, if multiple fire brigades are extinguishing the same fire, it is necessary to balance their distance to the fire to achieve their appropriate coordination.

5.3. Quantile-based interpretation

The operators $mean$, lra, lrb, and lr all give a distribution of a mean value. This is a typical use-case as the mean is well understood by practitioners. One, however, has to be aware of the fact that the mean is typically rather sensitive to outliers. The computation of the mean can be thus preceded with some form of outlier detection and exclusion. Care has to be taken because the definition of an outlier is purely domain-specific and requires a good understanding of outliers' cause. This is because the removal of outliers inherently changes the sample mean and the variance of the mean, which influences the results of the statistical tests used for the relational operators. Typically, this renders the test overconfident and increases the number of false positives.

An alternative to filtering outliers is to use a statistical value that is inherently robust to outliers. In particular, the median is a favorite choice. Also, generalizing the median to an arbitrary quantile has a nice advantage of giving the ability to reason about extremal values – e.g., with confidence γ, 90% of the measurements fall below a given threshold.

Nevertheless, in general, the use of median or quantiles comes with a relatively high computational cost. Though there exist relatively simple non-parametric tests for comparing the median of a set of i.i.d. observations (i.e., an operator in assumptions similar to $mean$), the quantile regression (i.e., yielding operators similar to lra, lrb, and lr) is much more complex. It turns out that its parameters $\hat{\alpha}$, $\hat{\beta}$ cannot be computed directly by a formula as in the case of ordinary least-squares, but such computation requires minimization, for instance, by means of linear programming.

5.4. Completeness and correctness

Our work is a set of operators and functions within propositional logic, designed in such a way that each term used in our formulas is either true or false. Since propositional logic is both complete and sound, so is our CB logic. As for the choice of statistical functions and the other operators we provide, we do not aim at completeness here. We rather consider our approach as open to additional statistical functions and tests (e.g., median and test over median). We selected a diverse set of the most important functions – covering means, linear regression, and ARIMA. Nevertheless, the CB logic library allows for incorporating other statistical functions, where it is possible to provide distribution F that can be used in hypothesis testing using operators $\geqslant_\gamma, >_\gamma$, etc. (as described in Section 4).

6. Evaluation

6.1. Research questions and experimental results

To evaluate our approach, we have applied it to the RCRS scenario described in Section 2. We set out several variants of the RCRS scenario with the intention to obtain an indication of whether employing CB logic operators in transition guards has the potential to improve system utility (explained below). In essence, the experiments with these scenarios were carried out with the aim to answer the following research question:

Does the use of CB logic in system design improve system utility w.r.t. to the baseline where decisions are based only on current readings (i.e., without the use of statistics (CB logic operators))?

Scenario	above	below	afv	notes
1	no	no	no	Baseline - Raw readings
2	no	no	no	Simple moving average of last 3 readings
3	no	no	yes	ARIMA
4	yes	yes	no	Linear regression
5	yes	yes	yes	Linear regression + ARIMA

Figure 5. Scenarios and operators considered in the evaluation.

In order to respond to this question, the variants of the RCRS scenario (Figure 5) include (1) the baseline (no employment of statistics), and using (2) simple moving average, (3) ARIMA, (4) linear regression, and (5) combination of linear regression and ARIMA. The experiments with these scenarios are to be compared to each other with a focus on if/how each of these improves system utility.

A key part of the RCRS scenario is `FireBrigade`. Key fragments of its "fully flagged" specification (corresponding to Scenario 5) are in Figure 1 and Figure 2. This fully autonomous component decides on its actions based on the sensor readings and modifies its behavior accordingly. Information about nearby buildings that are on fire is gathered using sensors that inherently provide uncertain data. For instance, the readings from thermographic cameras are affected by the distance to the building being observed, by the visibility conditions (dust, smoke, mist, …), and by the obstacles in direct sight between the camera and the building. This uncertain data can affect the immediate reaction of the `FireBrigade` so that it could perform a spurious mode transition. Furthermore, when `FireBrigade` is choosing which of the buildings on fire to extinguish, the key criterion considered is to save as many lives as possible; this requires predicting the number of people present in a building (another source of uncertainty). This is done by considering the type of building (commercial/residential) and the time and day of the week—commercial buildings are populated mainly in business hours and residential buildings conversely. The system utility is measured by the fraction of intact buildings to all buildings (the higher the fraction, the better).

To reflect this perspective, `FireBrigade` decisions in Scenario 1 are driven just by the "raw" values of sensor readings. Consequently, transition guards' evaluations are prone to spurious architectural transitions due to random readings outliers. In Scenario 2, raw reading values are replaced by the average of the latest 3 values to smoothen the output value (simple moving average). Scenario 3 differs from the baseline by predicting the actual occupation of buildings via the ARIMA *afv* operator. Scenario 4 utilizes linear regression CB operators in transition guards to gauge data provided by sensors.

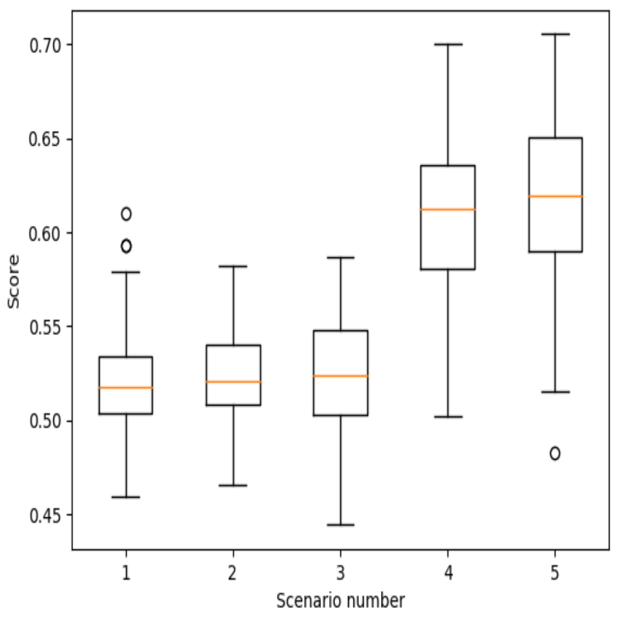

Figure 6. Results for 100 runs of each simulated scenario described in Figure 5. System utility ("score") is measured as the fraction of intact buildings to all buildings (the higher, the better).

Scenario	2	3	4	5
1	0.424017	0.186357	**5.48E-44**	**3.49E-46**
2		0.523249	**2.35E-44**	**1.89E-46**
3			**3.25E-39**	**8.24E-42**
4				0.079561

Figure 7. Calculated p-values for pairs of simulated scenarios. Statistically significant results (p-value< 0.05) in bold.

Accessible at GitHub, the implementations of these scenarios are based on a simulation realized by the customized RoboCup server[4] and on partially tailored RoboCup agents[5]. Figure 6 shows the results of the simulation runs (100 runs per Scenario). Here the efficiency of fire brigades in extinguishing fires (the system utility) is assessed via the score of an RCRS run, which is measured as the fraction of intact buildings to all buildings at the end of the run. Overall, they clearly show that using time series and CB logic in transition guards' evaluations improves system utility. Besides, they also indicate that the ARIMA model predicting building occupation values based on the time series of past values further improves system utility.

As to the research question, we answer it in the following way: By comparing the first and the second boxplot of Figure 6, we see that the use of moving average in the evaluation of mode transitions improves the system utility only marginally. The comparison of the first and the third boxplots indicates that the use of ARIMA CB operators also improves the system utility only slightly. On the other hand, the use of linear regression CB operators results in a considerable improvement of the system utility (the fourth boxplot). Finally, the comparison of the fourth and fifth boxplots shows that a combination of linear regression and ARIMA CB operators (Scenario 5) brings only marginal improvement with respect to Scenario 4 (linear regression only). Thus, in summary, the application of ARIMA improves system utility marginally, whilst linear regression has a substantial effect. A combination of it with ARIMA yields an additional improvement.

To support our claims, we performed two-tailed t-test (at significance level 0.05) for each pair of simulation results. The computed p-value for each pair of simulation results is in Figure 7, while taking into account that individual simulation runs are independent. The p-value confirms the non-directional hypothesis that there is a significant statistical difference between scenarios 1/2/3 and 4/5.

6.2. Details on the experiments, note on CB logic implementation, and threats to validity,

Details on the experiments: To get stable results, as already mentioned, we simulated each scenario 100 times. Each simulation run had a duration of 600 steps (the standard RCRS simulation runs take 300 steps). The RCRS simulator's build-in standard score function was used to assess the results of simulations (score 1 means all the buildings in the city are not damaged, while 0 means the worst case). As an aside, in an RCRS simulation, a building can be damaged by fire or by water. However, the latter could happen when a fire brigade takes a false positive decision (extinguishing a building that is not on fire) – this potentially worsens the score; ignoring a burning building has a similar consequence.

The scene of scenarios is simulated by the Kobe2013 map that represents a part (1.5. km^2) of the city Kobe in Japan during an earthquake. On the scene, there are twelve mobile fire brigades, having available 18 hydrants and a single fire station, which are located at fixed places in all the experiments. A hydrant can serve just a single fire brigade at a time, while the fire station can serve arbitrarily many. The water tank of a fire brigade can store 50,000 units of water which can be filled in at the fire station by 7,000 water units per simulation step and at a hydrant by 5,000 water units per simulation step.

Building fires are ignited randomly featuring a Poisson distribution with a lambda value of 0.3 ("to ensure enough fires") and spread depending on changing wind conditions. Each fire brigade is equipped with its fire sensor, which

[4] https://github.com/d3scomp/rcrs-server/tree/extended_modes

[5] https://github.com/d3scomp/rcrs-adapting-firefighters/tree/extended_modes

detects a fire with a probability depending on the distance from an observed building. If the distance is closer than a low threshold (8,000 distance units, defined by the RCRS simulator), the fire is detected with full certainty. If the distance is greater than a high threshold (40,000 distance units), the fire is not detected all (this is the limit of fire detectability in the simulator). Based on these two thresholds, a sensing probability is calculated (in the range from 0 to 1) linearly dependent on the actual distance and then subjected to Gaussian noise with a standard deviation of 0.1 (this is to simulate the inherent inaccuracies of the thermographic camera of the sensor). The final probability is used to determine whether a fire is detected. On top of this, even when a building is not on fire, the fire sensor might detect one (this may happen, e.g., when a building with a glass facade reflects the sun directly into the camera). This consequently triggers a false positive decision (in the simulation, this happens with the probability of 0.2).

Finally, the operators of CB logic feature a configurable window size. The exact configuration of these is highly application-dependent. In our simulations, we used a fixed window size of 8 last values for the CB operators and of one-week-long data for the ARIMA ones.

Note on implementation: In the simulation, we use the implementation of the key operators of CB logic (*mean*, *lra*, *lrb*, *lr*), which we also provide as a Java library tailored towards being used in sCPS projects (there also exists a C++ version of this library). The library offers a simple API for collecting data as time series and evaluating the relational operators over these key operators via statistical testing. The library also provides a means to train an ARIMA model and use it for value predictions.

Threads to validity: We performed an evaluation of how CB logic improves system utility on a single use-case only. Even though one cannot generalize from this, it has the value of showing that CB logic has the potential to help. Nevertheless, we aimed at mitigating the problem by selecting a case that serves as a widely acknowledged benchmark for multi-agent systems (and thus for adaptation involving multiple parties).

6.3. Additional case study

We have applied our proposed CB logic operators in a second case study from the cloud computing domain. In this case study, the focus is on the scheduling of services belonging to multiple applications to cloud servers in an optimal way. The optimization problem is, in general, multi-objective and considers both the minimization of cloud deployment costs (by using the provisioned servers to their full capacity), the load balancing between different servers in a cloud data center, and the satisfaction of performance requirements of certain services. In the context of the AFarCloud EU project[6], we focus on latency-critical applications in the farming sector such as analysis of images from drones that monitor soil, crop, cattle, and operation of tractors. In such a setting, the image recognition service in the cloud has a latency requirement that needs to be guaranteed, in order for the application to be at all usable and to provide real-time or at least time-bounded feedback to an operator even when complex data processing tasks are performed in the cloud.

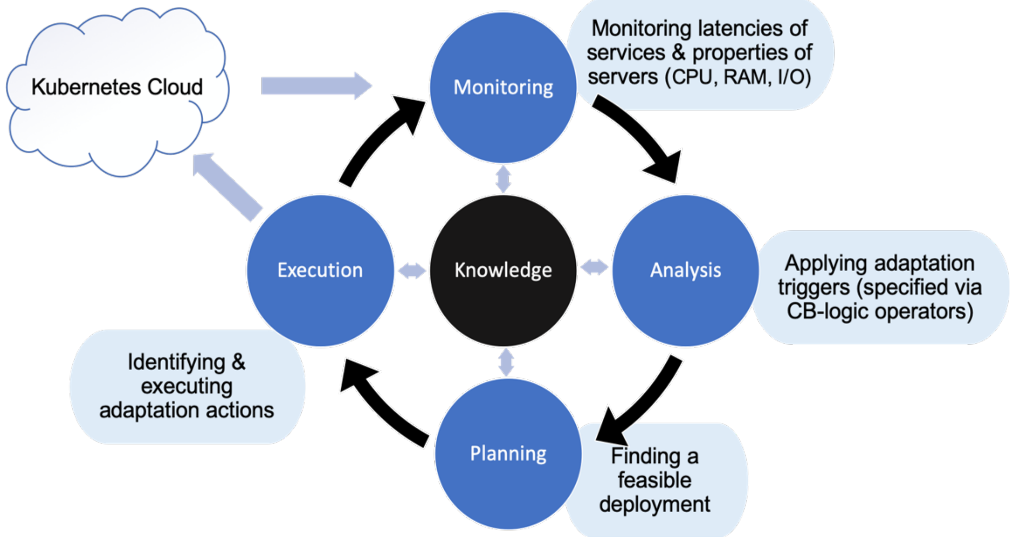
Kubernetes Cloud
Monitoring
Monitoring latencies of services & properties of servers (CPU, RAM, I/O)
Execution
Knowledge
Analysis
Applying adaptation triggers (specified via CB-logic operators)
Identifying & executing adaptation actions
Planning
Finding a feasible deployment

Figure 8. MAPE-K loop of the cloud computing case study.

To ensure that the cloud deployment cost is kept to a minimum and all near real-time guarantees of services are satisfied even in the face of continually changing conditions (changes in the background load, current utilization), we use a self-adaptation loop (Figure 8) structured according to the MAPE-K loop. The loop is embodied in an Adaptation Manager, which hosts the Knowledge (includes the database of measured data). We support two types of adaptation (re-planning in cloud terms): (i) emergency adaptation, which is simple but must be fast. We do this by

[6] http://www.afarcloud.eu

keeping a server aside and relocating/scaling the services if they seem to start failing to cope with the load; (ii) long-term adaptation, which potentially involves re-deployment of all services and requires complex planning phase (that involves a CSP problem solver) that is very resource-intensive and easily takes minutes to complete. We want to execute the long-term adaptation only when really necessary. Concretely, our adaptation loop consists of the following phases:

- **Monitoring.** Both the properties of servers (CPU, RAM, I/O utilization) and the performance of individual services are monitored. Technically, in our case study, we focus on Kubernetes (K8s) cloud for the scheduling of services packaged as Docker containers and hence rely on K8s cloud monitoring facilities. All the monitored time series data are sent to the Adaptation Manager.
- **Analysis.** A number of adaptation triggers are applied to the collected data to determine whether any service or resource requirement is violated or expected to be violated. In such a case, the Adaptation Manager triggers an adaptation process (explained in the Planning phase). Due to complex interactions of applications deployed on a single server, network issues, or monitoring anomalies, the measured CPU and RAM, as well as service latencies may fluctuate and create sensing uncertainty. Hence, we apply the CB logic operators to deal with this uncertainty by gathering a number of data points and applying statistical tests and identifying existing trends on their values. The application of the CB logic operators in this phase also provides a basis for proactive adaptation explained below.
- **Planning.** In this phase, the data collected in the Monitoring phase along with the service performance are used for adaptation. Technically, in this phase, a CSP solver may be invoked, in cases of a long-term adaptation to derive a feasible deployment, i.e. a deployment in which all the service requirements are satisfied.
- **Execution.** In this phase, the identified deployment is compared to the current one. For every change in the configuration, a number of actions have to be performed to change the state of the cloud accordingly. Based on that comparison, the Adaptation Manager selects and executes a number of actions to be performed on the K8s cloud, on the network, and on the running services (e.g., start a server, remove it, etc.).

In this case study, we have used the CB logic operators to provide probabilistic guarantees and re-plan only if we cannot show (by the statistical tests) that the average latency or CPU, RAM, or I/O load is below a particular threshold over a certain time window. Concretely, we have used CB logic to specify the following adaptation triggers:

1. "$above(CPU_{load}, 0.7, 5\ mins)$": For each server, if the CPU load is higher than 80% consistently the past 5 minutes, an adaptation is triggered. We used similar triggers for the RAM and I/O utilization metrics.
2. "$not\ maxfbelow(CPU_{load}, 0.9,\ 10\ mins)$": For each server, if the maximum value of its predicted (via linear regression) load over the next 10 minutes is not less or equal to 90%, a trigger is issued. Again, similar triggers were specified for the RAM and I/O utilization metrics.
3. "$not\ below(latency, V, T)$": For certain services, a trigger of this form was specified to bound the value of observed latency to a maximum value above which an adaptation should be triggered. The concrete values of *V* and *T* depend on the service being deployed. For the image recognition service, we used *V*=100ms and *T*=1min.
4. $\text{afv}(\text{RPM}, 24\text{ hours}, 10\text{ min}) > 10 * currect\_RPM$ and $\text{afc}(\text{requests}, 24\text{ hours}, 10\text{ min}) \geq 95\%$: For each service, we keep a record of the number of requests per minute (RPM) issued. Such requests tend to have a daily pattern. This trigger was put in place to trigger an adaptation in anticipation of a high predicted increase in the RPM, which could result in increased service latencies due to high load.

In this case study, the mean-based triggers (1 and 3) pertain to the emergency adaptation case, while the forecasting-based triggers (2 and 4) pertain to the long-term adaptation case. Overall, in this real-life case study, there are not many adaptation triggers. Still, when we first attempted to express them inline, we observed that their specification quickly becomes large and error-prone. When we then used CB logic for the specification of adaptation triggers, such specification became more comprehensible, intuitive, and easy to maintain.

7. Related work

In this section, we first focus on related work in handling uncertainty in self-adaptation and data streaming systems, and secondly on the different types of logics that might have been used in our approach.

7.1. Handling uncertainty in self-adaptation and data streams

To our knowledge, there are no other approaches that do not need initial (base) probabilities or a model of the system to be available upfront in architecture-based self-adaptation. Nevertheless, there are a number of works that are closely related to our approach and/or also employ statistical methods for handling uncertainty and managing variability – a recurring requirement in software engineering on capturing context dynamicity and reflecting its effect on system behavior. A systematic literature review in this is provided in [23], including run-time variability in self-adaptive systems. We compare our method to the state-of-the-art in self-adaptive systems, and for this purpose partially follow the classification in [12] for uncertainty sources options:

Changes in adaptation mechanisms. A well-known approach to capture uncertainties in self-adaptation is the language Stitch [19]. One of Stitch's basic concepts is tactics. It defines a condition over the architecture state, an action that has to be performed if the condition holds, and, finally, its effect that is the condition, which should hold after applying the action. Tactics are used in strategies, which describe dynamic adaptation processes. Compared to modes, Stitch offers a more fine-grained adaptation of an architecture, which, however, may not always be ideal for sCPS due to potentially limited hardware resources of embedded systems. Compared to our method, Stitch does not allow reasoning about history/future and time series in conditions. Probabilities are in Stitch used only in strategies to describe the likelihood that a condition will evaluate to true and subsequently employed in selecting a strategy to be executed.

Based on Stich, the Rainbow framework [23] relies on the MAPE-K model and captures different kinds of uncertainties, such as in sensing and effecting. The uncertainties are represented with different models, such as ranges or different kinds of distributions. For instance, the readings of a temperature sensor have physical limitations that are represented as a range, and any reading should fall in that range, whilst errors in readings are represented by the normal distribution function. The set of provided functions includes the normal distribution, linear function, and exponential distribution. Nevertheless, history/future and time series in architecture-based adaptation are not considered.

Focusing on the requirements in self-adaptive systems and building on KAOS goal-oriented models [24], FLAG (Fuzzy Live Adaptive Goals for Self-adaptive systems) introduces the notions of fuzzy goals and adaptive goals [25], [26]. As opposed to crisp goals whose fulfillment is Boolean, fuzzy goals have a degree of satisfaction in the [0,1] range and are formulated in a fuzzy temporal language [25]. Using the fuzzy operators of the language, one can express properties such as "a variable is almost always zero", "something holds almost until a certain point", or "a variable has a value less than zero with the exception of five cases". Finally, adaptive goals in FLAGS describe countermeasures that should be triggered when a fuzzy or crisp goal is violated. Similar to FLAGS, our approach tries to handle uncertainty at the level of adaptation mechanisms and aims at providing non-experts with the ability to express rich adaptation conditions. Contrary to FLAGS though, we do not model the adaptation logic via goals but via modes, and we do not capture uncertainty in conditions via fuzzy logic operators but via CB operators that apply statistical methods.

Finally, POISED is another approach that has tried to handle uncertainty in self-adaptive systems [27]. POISED provides a holistic framework for reasoning under uncertainty, both external (due to the environment variability and sensor noisiness, which is also our focus) and internal (due to the unpredictable effects of adaptation actions). Similar to our approach, POISED works with probability distributions of the sensed data, acknowledging that their statistical characteristics should not be ignored. However, contrary to our approach, POISED does not rely on statistical tests but on fuzzy mathematics and possibility theory to quantify the overall uncertainty in the models.

Sensing and Effecting. In [28], the authors target almost-sure reachability in stochastic multi-mode systems (SMMS); their approach is demonstrated on an example of a robot/autonomous vehicle that is trying to follow a path with arbitrary precision. The idea is to add bounded stochastic uncertainties with modes and consider tolerance in a control strategy (e.g., a path-follow algorithm that depends on arbitrary precision because of noisy sensors). Nevertheless, our method has a fine-grained representation of handling uncertainties over transitions guards. Moreover, the method in the paper does not consider a confidence level in accessing uncertainties. It rather keeps the robot/vehicle in a safe area in a neighborhood of the target point, which is associated with a probability distribution without giving the ability to determine the required confidence level.

As to learning the limits, an approach for adaptation of sensor networks used for modeling the behavior of a fire prediction system is presented in [29]. It is based on predictive analysis of historical data, i.e., time series of data measured by sensors. The goal is to proactively adapt parameters of the sensor network (e.g., rate of measurements and data reporting) depending on the situation. For instance, the system sets a higher rate of taking measurements in case of a possible forthcoming dangerous situation. For the actual prediction, the authors use the Multi-Layer Perception (MLP) model and employ the KNIME tool[7]. Even though the prediction depends on historical data and trends, it does not involve a confidence level that deals with fluctuations, nor does it consider the human presence in decision making.

A dynamic adaptation at run-time is discussed in [30]. To avoid issues with oscillations of context measures, the authors suggest using an event processing engine, namely Esper[8]. Esper offers a SQL-based event processing language, which

[7] http://www.knime.org/

[8] http://www.espertech.com/products/esper.php

allows for defining queries on run-time events with time windows and aggregation functions (like min, max, average). However, the analysis via linear regression and/or predictions of future events is not supported. A similar method is presented in [31], where the authors describe a self-adaptation approach. Processing of events, which are collected in a system, is described using an XML-based language that (as in the method above) allows for specifying time windows, aggregation functions, etc. Nevertheless, no predictions about the future are considered.

In our work [32], a linear state-space model is associated with component knowledge fields. The approach aims at capturing the uncertainty in knowledge caused by delays during data exchange in decentralized systems. This model is used to make predictions about maximal and minimal possible values of the actual unobserved knowledge. These predictions are used to make proactive mode switching.

CLARO [33] is a comprehensive approach for modeling and processing uncertain data streams. Similar to our approach, it tackles the uncertainty explicitly and allows evaluating the validity of predicates based on given uncertainty. Contrary to our approach, CLARO is based on the inference of the distribution of samples. It builds and algebra over the distributions. Though this is a more general approach, it also is prone to require more samples to reliably determine the uncertainty. On the other hand, our approach relies on distributions of aggregates (e.g., the mean), at which it assumes the normality (which follows even for not entirely normal observations from the Central Limit Theorem). By assuming the normal model, it requires fewer data to make accurate predictions.

In [34], the authors report on a method and tools for adaptation of an augmented reality application to balance its demand on resources (CPU, etc.) plus operation latency and the resources available. The adaptation method is based upon evaluating historical data in logs via a variant of linear regression - recursive least-squares regression with exponential decay. The purpose of the adaptation is to scarify the fidelity of results as least as possible. Nevertheless, the level of confidence is not explicitly evaluated.

7.2. Other logics

In the following, we provide a discussion of several alternative approaches to tackle uncertainty as to the underlying logics [35]. In particular, we focus on their differences compared to BP logic that make them inconvenient or even impossible to use for transition guards.

Probabilistic logic [36] generalizes the interpretation of logic sentences, i.e., combinations of elementary formulas. Even though it can capture probabilities of composed formulas, the elementary formulas are always evaluated just to *true* or *false*. The resulting probability is computed based on the knowledge of the number of options forming the complete set, i.e., all the combinations of the true valuations of particular elements. The logic also does not support reasoning about history (generally data series), which is theoretically possible, e.g., by introducing versioning of variables, but that would be a bit cumbersome.

Logic for uncertain probabilities (i.e., Subjective Logic) [37] [38] [39] is centered around the term *belief* referring to a subjective perception of an observer assessing the level of truth for elementary and composed elements. The aim of the logic is to capture the fact that different observers can assign different trueness values to the same elements, thus making the logic closer to the real world. In this approach, the concept of super states (containing all or just some of the options out of which at most one elementary state is true) plays a central role. The observer then assigns various values to the elements, reflecting his/her/its perception of trueness levels. Then, for example, the combinations of the most probable (elementary) states can be chosen for further exploration. Similar to the probabilistic logic, the logic for uncertain probabilities does not directly support history, although it does allow for capturing several observations of an event in time (which would model a time series with just values true/false) with different trueness values.

A similar approach is taken by *fuzzy logic*[40] [41] [42], where the variables do not take just the binary values (*true, false*), but any real number in the [0,1] interval, reflecting how much true the variable is considered (1 is equal to *true*, while 0 to false). This, to some extent, also reflects the real world. For instance, in the control domain, the authors in [43] target mission-critical systems and present fuzzy controller in the self-adaptation loop, so the rules are modified, whilst in [44] involved fuzzy logic in the controller of a robot in such a way that it adapts by modifying both the rules and the scale factors. Another example is presented in [45], where the paper presents a combination of Petri-Nets with fuzzy rules to accomplish self-adaptation. The logic defines the logical operators of conjunction, disjunction, and negation as the minimum, maximum, and complement to one, respectively, to cope with the real valuations. Although this logic could be taken as a base for our work, we would not be able to apply the statistical reasoning to it, which is the central part of our approach. Also, a variable of fuzzy logic keeps uncertainty "inside its value"; this, e.g., prevents a later evaluation of confidence of the computed value.

Dempster-Shafer theory [46] [47] [48] or *evidential reasoning* is probably the approach closest to ours by defining probabilities on (propositional) sentences (of the logic) and the notion of *belief* to reason about imprecise and uncertain evidence. However, it focuses on proving logical sentences to be true or false based on the observations by various agents, rather than on computing a value along with its confidence level, as the standard statistical methods do.

Markov decision processes [49] [50], including both *Discrete-Time Markov Chains* (DTMC) and *Continuous-Time Markov Chains* (CTMC) [51], provide the developers with the means to capture probabilities of elementary events. There are plenty of approaches equipped with solid tool support, which allow for reasoning about the stochastic properties of the chains. To employ them to perform system analysis, however, it is necessary to have available an entire chain; this is rarely the case in the dynamically-evolving systems we target. In our case, we capture the situation about components and ensembles around, but just as to the past and present, attempting to predict values in the near future, often with the aim of comparing the impact of several alternative choices to be taken. Furthermore, at each state of a Markov Chain, the probabilities for particular transitions are defined regardless of the past. This makes it hard to employ any type of Markov Chains in our approach.

Probabilistic Computational Tree Logic (PCTL) [51] [52] is one of the ways to specify the stochastic properties of systems, namely DTMC mentioned above. It extends the traditional CTL logic with probabilities and optional specification of the number of steps until a sub-property should be satisfied. Although being general enough to capture most real-life properties, again, its semantics is defined upon a system model with explicitly given probabilities; this is an unrealistic assumption for applying PCTL in our case. An example of such a use is the case presented in [53], where the aim is to verify self-adaptive systems using probabilistic model checking (PMC).

All the logics above either require the input (basic) probabilities to be a priori defined or a model of the entire system to be available. These two requirements make it hard or even practically impossible to employ these logics for transition guards.

For forecasting of the future values, the ARIMA approach has been chosen. In principle, other methods can be employed, such as neural networks and machine learning in general. Even though these methods can provide better results in some cases, especially when the history data exhibit a lot of noise and irregularity, we consider them inappropriate due to their extensive computational demands and the need for large sets of training data, which is typically not available in the use cases considered by us.

8. Conclusion

In summary, the paper targets uncertainty in architecture-based self-adaptation in sCPS. In particular, it focuses on coping with environmental uncertainty and dynamic architectural adaptation; it handles related issues at the level of transition guards – the conditions, written in a logic, controlling the reconfiguration of SW architecture. As to uncertainty, it is assumed that data from the environment cannot be observed directly without noise and that a precise model of the environment cannot be obtained.

In contrast to other approaches targeting uncertainty that typically employ a model of the environment and/or assume a prior distribution or base probabilities of data, the paper proposes a confidence-based logic (CB logic) employing just the observed data (forming time series) to make probabilistic conclusions about the environment. CB logic defines a number of operators allowing to handle noise and incompleteness of the data and to predict future event occurrences with a specific level of confidence. By combining different operators in concise language constructs, CB logic allows even non-experts in statistics to easily perform statistically rich operations upon time series data.

The viability of CB logic is demonstrated by employing it in transition guards controlling component mode transitions and membership evaluation in component ensembles. This is illustrated on a running example specified in a DSL language akin to the one of the DEECo component model which, by following the MAPE-K idea, supports architecture-based adaptation. Based on the scenario from the RoboCup Rescue Simulation, the running example allowed us to carry out a number of simulation experiments, which showed a considerable improvement in system utility when CB logic is applied. For the potential adoption of the approach, a Java and C++ implementation of key CB logic operators is available at GitHub.

Acknowledgments

The work has been partially supported by the project no. LTE117003 (ESTABLISH) from the INTER-EUREKA LTE117 programme by the Ministry of Education, Youth, and Sports of the Czech Republic and partially supported by the Charles University institutional funding SVV 260451.

References

[1] M. Hölzl, A. Rauschmayer, and M. Wirsing, “Software Engineering for Ensembles,” in *Software-Intensive Systems and New Computing Paradigms*, M. Wirsing, J.-P. Banâtre, M. Hölzl, and A. Rauschmayer, Eds. Springer, 2008, pp. 45–63.

[2] B. Morin, F. Fleurey, and O. Barais, “Taming Heterogeneity and Distribution in sCPS,” in *Proceedings of SEsCPS 2015, Firenze, Italy*, 2015, pp. 40–43, doi: 10.1109/SEsCPS.2015.15.

[3] I. Ruchkin, B. Schmerl, and D. Garlan, “Architectural Abstractions for Hybrid Programs,” in *Proceedings of CBSE 2015, Montreal, Canada*, New York, NY, USA, 2015, pp. 65–74, doi: 10.1145/2737166.2737167.

[4] C. Ghezzi, L. S. Pinto, P. Spoletini, and G. Tamburrelli, “Managing non-functional uncertainty via model-driven adaptivity,” in *2013 35th International Conference on Software Engineering (ICSE)*, San Francisco, CA, USA, 2013, pp. 33–42, doi: 10.1109/ICSE.2013.6606549.

[5] A. Filieri *et al.*, “Software Engineering Meets Control Theory,” in *Proceedings of SEAMS 2015, Florence, Italy*, 2015, pp. 71–82, doi: 10.1109/SEAMS.2015.12.

[6] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, “Rainbow: architecture-based self-adaptation with reusable infrastructure,” *Computer*, vol. 37, no. 10, pp. 46–54, Oct. 2004, doi: 10.1109/MC.2004.175.

[7] D. Garlan, B. Schmerl, and S.-W. Cheng, “Software Architecture-Based Self-Adaptation,” in *Autonomic Computing and Networking*, Boston, MA: Springer US, 2009, pp. 31–55.

[8] J. Cámara, P. Correia, R. de Lemos, and M. Vieira, “Empirical Resilience Evaluation of an Architecture-based Self-adaptive Software System,” in *Proceedings of the 10th International ACM Sigsoft Conference on Quality of Software Architectures, Lille, France*, New York, NY, USA, 2014, pp. 63–72, doi: 10.1145/2602576.2602577.

[9] J. Kephart and D. Chess, “The Vision of Autonomic Computing,” *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[10] I. Gerostathopoulos, T. Bures, P. Hnetynka, A. Hujecek, F. Plasil, and D. Skoda, “Strengthening Adaptation in Cyber-Physical Systems via Meta-Adaptation Strategies,” *ACM Transactions on Cyber-Physical Systems*, vol. 1, no. 3, pp. 1–25, 2017, doi: 10.1145/2823345.

[11] I. Gerostathopoulos, D. Škoda, F. Plášil, T. Bureš, and A. Knauss, “Tuning Self-Adaptation in Cyber-Physical Systems through Architectural Homeostasis,” *Journal of Systems and Software*, Feb. 2019, doi: 10.1016/j.jss.2018.10.051.

[12] S. Mahdavi-Hezavehi, P. Avgeriou, and D. Weyns, “A Classification Framework of Uncertainty in Architecture-Based Self-Adaptive Systems With Multiple Quality Requirements,” in *Managing Trade-Offs in Adaptable Software Architectures*, Elsevier, 2017, pp. 45–77.

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

[14] R. J. Anthony, M. Pelc, and W. Byrski, “Context-aware reconfiguration of autonomic managers in real-time control applications,” in *Proceeding of the 7th international conference on Autonomic computing - ICAC ’10*, Washington, DC, USA, 2010, p. 73, doi: 10.1145/1809049.1809061.

[15] P. Casanova, D. Garlan, B. Schmerl, and R. Abreu, “Diagnosing architectural run-time failures,” in *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2013, pp. 103–112.

[16] P. Casanova, B. Schmerl, D. Garlan, and R. Abreu, “Architecture-Based Run-Time Fault Diagnosis,” in *Software Architecture*, Berlin, Heidelberg, 2011, pp. 261–277, doi: 10.1007/978-3-642-23798-0_29.

[17] T. Bures, I. Gerostathopoulos, P. Hnetynka, J. Keznikl, M. Kit, and F. Plasil, “DEECo: An ensemble-based component system,” in *Proceedings of CBSE 2013, Vancouver, Canada*, 2013, pp. 81–90, doi: 10.1145/2465449.2465462.

[18] T. Bures, P. Hnetynka, J. Kofron, R. Al Ali, and D. Skoda, “Statistical Approach to Architecture Modes in Smart Cyber-Physical Systems,” in *Proceedings of WICSA 2016, Venice, Italy*, 2016, pp. 168–177, doi: 10.1109/WICSA.2016.33.

[19] S.-W. Cheng and D. Garlan, “Stitch: A language for architecture-based self-adaptation,” *Journal of Systems and Software*, vol. 85, no. 12, pp. 2860–2875, Dec. 2012, doi: 10.1016/j.jss.2012.02.060.

[20] J. Kephart and D. Chess, “The Vision of Autonomic Computing,” *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[21] P. J. Brockwell and R. A. Davis, Eds., *Introduction to Time Series and Forecasting*. New York, NY, USA: Springer, 2002.

[22] L. De Moura and N. Bjørner, “Z3: An Efficient SMT Solver,” in *Proceedings of TACAS’08, Budapest, Hungary*, 2008, vol. 4963, pp. 337–340, doi: 10.1007/978-3-540-78800-3_24.

[23] J. Cámara, D. Garlan, W. G. Kang, W. Peng, and B. Schmerl, “Uncertainty in Self-Adaptive Systems Categories, Management, and Perspectives,” Institute for Software Research School of Computer Science Carnegie Mellon University, Pittsburgh, PA 15213, CMU-ISR-17-110, 2017.

[24] A. V. Lamsweerde, “Requirements Engineering: from Craft to Discipline,” in *Proc. of FSE ’08*, 2008, pp. 238–249, doi: 10.1145/1453101.1453133.

[25] L. Baresi, L. Pasquale, and P. Spoletini, “Fuzzy Goals for Requirements-driven Adaptation,” in *Proc. of RE ’10*, 2010, pp. 125–134, doi: 10.1109/RE.2010.25.

[26] L. Baresi and L. Pasquale, “Live Goals for Adaptive Service Compositions,” in *Proc. of SEAMS ’10*, New York, NY, USA, 2010, pp. 114–123, doi: 10.1145/1808984.1808997.

[27] F. Somenzi, B. Touri, and A. Trivedi, “Almost-Sure Reachability in Stochastic Multi-Mode System,” *arXiv:1610.05412 [math]*, Oct. 2016.

[28] Ivan Dario Paez Anaya, V. Simko, J. Bourcier, N. Plouzeau, and J. Jézéquel, “A Prediction-Driven Adaptation Approach for Self-Adaptive Sensor Networks,” in *Proceedings of SEAMS 2014, Hyderabad, India*, 2014, pp. 145–154, doi: 10.1145/2593929.2593941.

[29] B. Morin, O. Barais, J.-M. Jezequel, F. Fleurey, and A. Solberg, “Models@ Run.time to Support Dynamic Adaptation,” *Computer*, vol. 42, no. 10, pp. 44–51, Oct. 2009, doi: 10.1109/MC.2009.327.

[30] L. Baresi, S. Guinea, and G. Tamburrelli, “Towards Decentralized Self-adaptive Component-based Systems,” in *Proceedings of SEAMS’08, Leipzig, Germany*, 2008, pp. 57–64, doi: 10.1145/1370018.1370029.

[31] R. Al Ali, T. Bures, I. Gerostathopoulos, J. Keznikl, and F. Plasil, “Architecture Adaptation Based on Belief Inaccuracy Estimation,” in *Proceedings of WICSA 2014, Sydney, Australia*, 2014, pp. 87–90, doi: 10.1109/WICSA.2014.20.

[32] T. T. L. Tran, L. Peng, Y. Diao, A. McGregor, and A. Liu, “CLARO: modeling and processing uncertain data streams,” *The VLDB Journal*, vol. 21, no. 5, pp. 651–676, Oct. 2012, doi: 10.1007/s00778-011-0261-7.

[33] D. Narayanan and M. Satyanarayanan, “Predictive Resource Management for Wearable Computing,” in *Proceedings of the 1st international conference on Mobile systems, applications and services*, San Francisco, California, 2003, pp. 113–128, doi: 10.1145/1066116.1189041.

[34] M. J. Wierman, “An Introduction to the Mathematics of Uncertainty,” Creighton University, College of Arts and Sciences, 2010.

[35] N. J. Nilsson, “Probabilistic logic,” *Artificial Intelligence*, vol. 28, no. 1, pp. 71–87, 1986, doi: https://doi.org/10.1016/0004-3702(86)90031-7.

[36] A. Josang, “A Logic for Uncertain Probabilities,” *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, vol. 9, no. 3, pp. 279--311, 2001, doi: 10.1142/S0218488501000831.

[37] A. Jøsang, *Subjective Logic: A Formalism for Reasoning Under Uncertainty*. Springer International Publishing, 2016.

[38] A. Josang, “Generalising Bayes’ theorem in subjective logic,” in *2016 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI)*, Baden-Baden, Germany, 2016, pp. 462–469, doi: 10.1109/MFI.2016.7849531.

[39] L. A. Zadeh, “Fuzzy logic,” *Computer*, vol. 21, no. 4, pp. 83–93, Apr. 1988, doi: 10.1109/2.53.

[40] T. J. Ross, *Fuzzy logic with engineering applications*, 3rd ed. Chichester, U.K: John Wiley, 2010.

[41] L. A. Zadeh, “The role of fuzzy logic in the management of uncertainty in expert systems,” *Fuzzy Sets and Systems*, vol. 11, no. 1–3, pp. 199–227, 1983, doi: https://doi.org/10.1016/S0165-0114(83)80081-5.

[42] Q. Yang, J. Lu, J. Xing, X. Tao, H. Hu, and Y. Zou, “Fuzzy Control-Based Software Self-Adaptation: A Case Study in Mission Critical Systems,” in *2011 IEEE 35th Annual Computer Software and Applications Conference Workshops*, 2011, pp. 13–18, doi: 10.1109/COMPSACW.2011.13.

[43] I. Rojas, H. Pomares, J. Gonzalez, L. J. Herrera, A. Guillen, and O. Valenzuela, “Self-adaptive robot control using fuzzy logic,” in *9th IEEE International Workshop on Advanced Motion Control, 2006.*, 2006, pp. 687–691, doi: 10.1109/AMC.2006.1631743.

[44] Z. Ding, Y. Zhou, and M. Zhou, “Modeling Self-Adaptive Software Systems by Fuzzy Rules and Petri Nets,” *IEEE Transactions on Fuzzy Systems*, vol. 26, no. 2, pp. 967–984, Apr. 2018, doi: 10.1109/TFUZZ.2017.2700286.

[45] M. Beynon, B. Curry, and P. Morgan, “The Dempster–Shafer theory of evidence: an alternative approach to multicriteria decision modelling,” *Omega*, vol. 28, no. 1, pp. 37–50, Feb. 2000, doi: 10.1016/S0305-0483(99)00033-X.

[46] J. Gordon and E. H. Shortliffe, “The Dempster-Shafer Theory of Evidence,” 1990, pp. 272--292.

[47] A. P. Dempster, “Upper and Lower Probabilities Induced by a Multivalued Mapping,” *The Annals of Mathematical Statistics*, vol. 38, no. 2, pp. 325–339, 1967, doi: 10.1214/aoms/1177698950.

[48] M. L. Puterman, “Chapter 8 Markov decision processes,” in *Handbooks in Operations Research and Management Science*, vol. 2, Elsevier, 1990, pp. 331–434.

[49] C. C. White and D. J. White, “Markov decision processes,” *European Journal of Operational Research 39*, pp. 1--16, 1989.

[50] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds., *Handbook of Model Checking*. Cham: Springer International Publishing, 2018.

[51] F. Ciesinski and M. Größer, “On Probabilistic Computation Tree Logic,” in *Validation of Stochastic Systems*, vol. 2925, C. Baier, B. R. Haverkort, H. Hermanns, J.-P. Katoen, and M. Siegle, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 147–188.

[52] J. Cámara and R. de Lemos, “Evaluation of resilience in self-adaptive systems using probabilistic model-checking,” in *2012 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, 2012, pp. 53–62, doi: 10.1109/SEAMS.2012.6224391.