Enabling Temporal-Aware Contexts for Adaptative Distributed Systems

Ludovic Mouline
University of Luxembourg
Univ Rennes, Inria, CNRS, IRISA
ludovic.mouline@{uni.lu,inria.fr}

Amine Benelallam
Univ Rennes, Inria, CNRS, IRISA
amine.benelallam@inria.fr

Thomas Hartmann
University of Luxembourg
thomas.hartmann@uni.lu

François Fouquet
University of Luxembourg
francois.fouquet@uni.lu

Johann Bourcier
Univ Rennes, Inria, CNRS, IRISA
johann.bourcier@inria.fr

Brice Morin
SINTEF
Brice.Morin@sintef.no

Olivier Barais
Univ Rennes, Inria, CNRS, IRISA
olivier.barais@inria.fr

ABSTRACT

Distributed adaptive systems are composed of federated entities offering remote inspection and reconfiguration abilities. This is often realized using a MAPE-K loop, which constantly evaluates system and environmental parameters and derives corrective actions if necessary. The *OpenStack Watcher* project uses such a loop to implement resource optimization services for multi-tenant clouds. To ensure a timely reaction in the event of failures, the MAPE-K loop is executed with a high frequency. A major drawback of such reactivity is that many actions, *e.g.,* the migration of containers in the cloud, take more time to be effective and their effects to be measurable than the MAPE-k loop execution frequency. Unfinished actions as well as their expected effects over time are not taken into consideration in MAPE-K loop processes, leading upcoming analysis phases potentially take sub-optimal actions. In this paper, we propose an extended context representation for MAPE-K loop that integrates the history of planned actions as well as their expected effects over time into the context representations. This information can then be used during the upcoming analysis and planning phases to compare measured and expected context metrics. We demonstrate on a cloud elasticity manager case study that such *temporal action-aware context* leads to improved reasoners while still be highly scalable.

CCS CONCEPTS

• **Software and its engineering** → **Software design engineering**; • **Computer systems organization** → *Reconfigurable computing*;

SAC 2018, April 9–13, 2018, Pau, France

ACM ISBN 978-1-4503-5191-1/18/04…$15.00
https://doi.org/10.1145/3167132.3167286

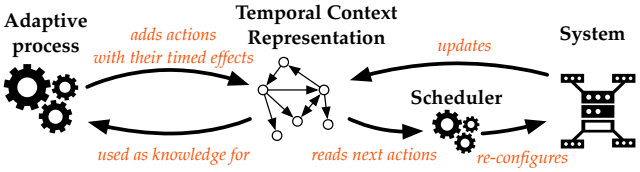

Figure 1: Overview of temporal-aware adaptation

KEYWORDS

model-driven engineering, context modelling, action modelling, models@run.time

ACM Reference Format:
Ludovic Mouline, Amine Benelallam, Thomas Hartmann, François Fouquet, Johann Bourcier, Brice Morin, and Olivier Barais. 2018. Enabling Temporal-Aware Contexts for Adaptative Distributed Systems. In *SAC 2018: SAC 2018: Symposium on Applied Computing , April 9–13, 2018, Pau, France.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3167132.3167286

1 INTRODUCTION & MOTIVATION

In response to the increasing complexity of systems and their ever-changing environment, many approaches to design and develop software able to adapt according to the current context have emerged. Adaptive software systems are characterized by their ability to, dynamically and reactively, evaluate their context and adjust their behavior accordingly. A systematic approach to realize adaptation is by using a feedback control loop. The most commonly used approach for adaptive systems is the MAPE-K loop, for Monitor, Analyse, Plan and Execute over knowledge. Based on knowledge/-context information, the MAPE-K loop continuously reasons about the system context to take appropriate and optimal actions. System context can be defined as the circumstance in which a computing task takes place [10].

Fined and rich context information directly influences the accuracy of the actions taken. Various techniques to represent context information have been proposed; among which we find the models@run.time [5, 13]. The models@run.time paradigm inherits model-driven engineering concepts to extend the use of models not

only at design time but also at runtime. This model-based representation has proven its ability to structure complex systems and synthesize its internal state as well as its surrounding environment.

The MAPE-K loop has been successfully incorporated into distributed software systems for numerous goals. Actions generate during the adaptation step could take time upon completion. Moreover, the expected effects resulting from such action is reflected in the context representation only after a certain delay. We refer to these actions as long actions.

For instance, the OpenStack Watcher project implements a MAPE-K loop to assist cloud administrators in their activities to tune and rebalance their cloud resources according to some optimization goals (e.g., CPU and network bandwidth). The same applies for the allocation of software containers to host nodes by recent Docker Swarm and Kubernetes projects. Activities carried out by the Watcher project include metric collection and analysis, action planning (e.g., live resource migration), and action recommendations (advise mode) or action execution (active mode) that satisfies the optimization goals. Nevertheless, activities such as cloud rebalancing involve long actions like resource migration. **Due to the lack of information about unfinished actions and their expected effects on the system, the reasoning component may take repeated or sub-optimal decisions while these actions are being performed.**

A commonly used workaround is the selection, often empirically, of an optimistic time interval between two iterations of the MAPE-K loop such that this interval is bigger than the longest action execution time. However, the time to execute an action is highly influenced by system overload or failures, making such empirical tuning barely reliable. We argue that by enriching context representation with support for past and future planned actions and their expected effects over time, we can highly enhance reasoning processes and avoid empirical tuning. As illustrated in Figure 1 **our approach uses temporal context to represent expected action effects and enable their use in reasoning processes.** Our current approach is limited to the representation of measurable effects of any action. Using a cloud manager scenario, we show the efficiency of such context representation, in order to detect container migration delay for instance.

The paper is structured as follows. We first introduce some preliminary concepts in Section 2. Later we introduce our contribution in Section 3. Finally, Section 4 evaluates our approach using a cloud elasticity manager case study, while Section 6 concludes the paper and draws some future research lines.

2 BACKGROUND

This section describes the background for this work. First, we describe the original MAPE-K loop, which we aim to extend in this paper. Then, we discuss the models@run.time paradigm, a well-known implementation of the MAPE-K loop, on top of which we build our proposed approach. Finally, we describe the background related to the representation of temporal context.

2.1 The MAPE-K loop

The MAPE-K feedback loop, originally introduced by IBM [1], is the reference model for autonomic and self-adaptive systems. As shown in Figure 2, it describes a continuous feedback loop of monitoring context information through sensors, analysing these data, planning corrective actions if necessary, and executing these actions through actuators. The main idea of this feedback loop is that a system maintains a model of itself and its environment—*i.e.,* a context representation—at runtime and uses this model to reason about itself in order to achieve the system's goals. This model is in the original MAPE-K feedback loop referred to as *knowledge*, which is shared among the monitor, analyze, plan, and execution phases. It is created by the monitor phase and might be updated by the execution phase.

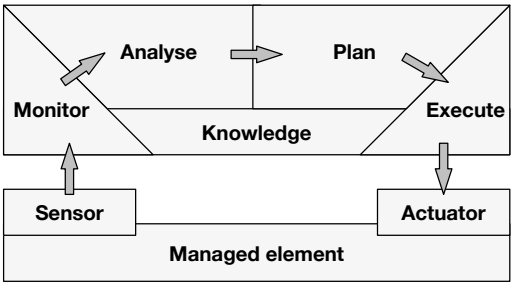

Figure 2: Overview of the MAPE-K loop process

2.2 Models@run.time

The models@run.time paradigm [5, 13] is a well-known realization of the MAPE-K loop to handle the challenges faced in self-adaptive systems development. It extends model-driven engineering (MDE) concepts by spanning the use of models from design time to runtime (*i.e.,* during the execution of a system). The model, as an abstraction of a real system, can be used during runtime to reason about the state of the actual system. A conceptual link between the model and the real system allows modifying the actual system through the model and vice versa. Models in MDE provide semantically rich ways to define contexts that can be used in reasoning activities. The models@run.time paradigm uses meta-models to define the domain concepts of a real system together with its surrounding environment. Consequently, the runtime model depicts an abstract and yet rich representation of the system context that conforms to (is an instance of) its meta-model. In this work, we seek to use the models@run.time paradigm to represent, reason, and update the context of a system.

2.3 Temporal context representations

Most context representations are defined as snapshots of a running system at given times. Nonetheless, the actual system keeps changing over time. Many reasoning activities need to read past or future states. Such time-evolving contexts can be either defined as a costly sequence of snapshots and deltas or by the aggregation of independent time series, resulting in the loss of relationships between context elements. To overcome such issues for temporal context representations, Hartmann *et al.,* [8, 9] suggested the use of a cross-cutting temporal dimension, by implicitly adding a *time validity* to every context element. A *version* of a context element is then valid from a time t_i to a time t_e. In between, an infinite

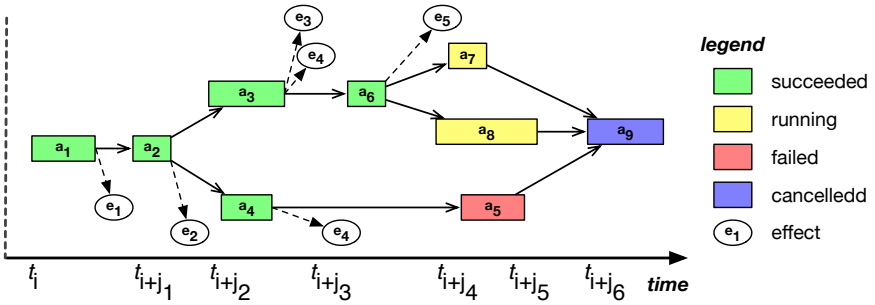

Figure 3: Directed acyclic graph of actions

number of versions of time t_j can be inserted and the whole history can be accessed. For this paper, we assume that the following four temporal functions are given:

Func 1. ***start(id)*** $\rightarrow$ ***time***, *where id is a unique identifier of a context element and time is the timestamp where the lifeline of this element starts.*

Func 2. ***end(id)*** $\rightarrow$ ***time***, *where id is a unique identifier of a context element and time is the timestamp where the lifeline of this element ends.*

Func 3. ***resolve(id,time)*** $\rightarrow$ ***{A, R}***, *where id is a unique identifier of a context element, time is a timestamp, and A_n R_n are respectively the set of attributes and relationships values associated to the context element at the given time.*

Func 4. ***time(id)*** $\rightarrow$ ***time***, *where id is a unique identifier of a context element, time is the last timestamp at which the element has been modified.*

Using these functions, any structure definition, such as a meta-model, can be turned into a temporal representation. In this paper, we suppose that our context representation formalism satisfies this assumption. **Our context representation is a temporal structure definition following the models@run.time paradigm.**

3 ACTIONS AS CONTEXT INFORMATION

In order to enhance sustainable decision-making in time evolving contexts, we strongly think that actions together with their expected effects should be part of the context representation of adaptive systems. In this perspective, we first describe a cluster infrastructure management example for illustrating our approach. Then, we show how an action plan can be represented as a directed acyclic graph (DAG) of actions spanned over time. Then, we detail on how we enrich this DAG with the expected effect of actions, and later with their measured equivalences, *i.e.,* how we link the DAG to the context representation. Finally, we illustrate how this information can be used by the MAPE-K loop to improve reasoning processes through four reasoning operations.

3.1 Example: cluster management

Throughout this section, we refer to a cluster management case study, the Watcher project, to exemplify our approach. This framework performs adaptation processes over a cloud infrastructure in order to optimize resource usage, such as CPU, memory, and network. To achieve this, Watcher defines primitive actions to be performed, such as container's creation, migration, and duplication. Furthermore, Watcher comes with a set of metrics, either simple or derived, for setting up optimization algorithms. For instance, deciding whether to allocate or deallocate resources in the cluster. In this paper, we confine to two metrics: container status and container workload. Being part of the context information, these metrics values play a crucial role in planning and executing adaptation processes. These values are first modified: the system will update the container status and their workload. Moreover, these values can be subject to changes as a result of executing the planned actions.

A common adaptation scenario is minimizing the amount of resources provided by the cluster to answer a given service workload (W_S) that is expected a priori. The framework thus has to decide whether to allocate or deallocate resources, and thereby, increase or decrease the amount of the provided workload (W_P). To do so, the framework first compares W_S to W_P, then determines the most convenient action plan. As soon as an allocation or deallocation action is performed, the workload values are increased or decreased and the container statuses are modified accordingly (e.g., *READY*, *DOWN*). Neither of these actions is effective immediately and they can be subject to failure. As shown in [11], a virtual machine based on Linux could take from 90 seconds to more than 200 seconds to be ready.

3.2 Action planning

System adaptation is commonly performed as a set of actions. In complex scenarios, these actions are often interdependent and require advanced action planning. It takes into consideration not only dependency but also actions execution status. Furthermore, existing simplified models,*e.g.,* [4, 12], do not capture action execution time and assume that they complete immediately after they are triggered. Capturing such temporal information (*e.g.,* (de)allocation delays) contributes to the quality of context information and improves the reasoning process of the MAPE-K loop.

We propose to represent action plans by means of DAGs of actions spanning over time, where each node corresponds to an action and an edge to a precedence relationship between actions. Figure 3 shows an example of an action plan represented using an actions DAG. Actions can be executed in parallel or sequential way. Each

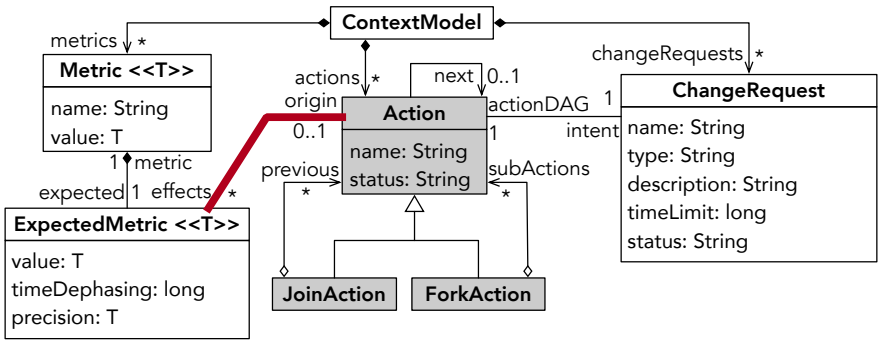

Figure 4: Principle metamodel integrating actions into context models

action has a *start time*, an *execution duration*, and *a time until its effects are measurable* by the system. These properties can be reproduced using the three temporal functions defined in Section 2.3. The execution duration will be the difference between the *start time* and the *end time* values, defined by the *start(id)* and *end(id)* functions. Expected effects resulting from the execution of an action are also attached to actions, as highlighted in Figure 3 by a dashed arrow. The *time until action effects are measurable* will correspond to the start time, *i.e.,* the result of the function *start(id)*, of the context element. We differentiate between several actions statuses, namely, created, pending, running, failed, succeeded, or canceled. An action is triggered when all the actions of the incoming edges succeed. In case of actions failure or cancellation, the subsequent actions are canceled then rescheduled for the next planning phase. We provide more details about expected effects representation below. This DAG is then fed to the knowledge component of the MAPE-K feedback loop to improve the reasoning process.

We describe in the next section how we represent this DAG and how we link it to the context presentation.

3.3 Linking action planning and context representation

In order to enable reasoning on actions, we include two concepts in our context representation model, the *measured* value and *expected* value. While *measured* represents the value of data measured by the system's sensors, *expected* represents data reflecting the planned effects of actions. We assume that actions are triggered by change requests made either by a human or the system itself (*e.g.,* the reasoning process). Based on change requests, the reasoning process will generate the DAG of actions with effects on metrics. For the sake of understandability, this concept should encapsulate the intent behind performing an action.

An excerpt of a meta-model representing these concepts is provided in Figure 4. The right-hand side of the meta-model depicts how we model change requests, thanks to the *ChangeRequest* metaclass. The *name* and the *type* are used by the reasoning process to infer the desired state as well as the actions to reach it. The *timeLimit* attribute allows to define a time-out for each individual request.

The center part of the meta-model (with grey background) depicts the meta-model used to create instances of the DAG of actions. To support the sequential and parallel execution, we define three classes: *Action*, *Fork Action* and *Join Action*. While sequential actions can be modeled using the relationship *next*, parallel nodes can be modeled using fork actions and join actions. The *JoinAction* is simply a node with multiple incoming edges, while the *ForkAction* node with multiple outgoing edges. The actions are part of the context model thanks to the reference *next* associating the *Action* class to the *ContextModel* class.

The left-hand side deals with the sub-model that supports expected effects. A value of a metric can be of any type. We represent it in the metamodel through the generic type parameter *T*. Metrics with *measured values* are directly connected to the context model. This corresponds to the usual semantics of context models. In the Figure 4, this is shown by the *Metric* class. *ExpectedMetric* are used to represent the expected values. Thanks to the *timeDephasing* and the *precision* attributes we are able to handle time expressions with a high level of precision. Conceptual links between an action and its corresponding expected metrics are established using the relationship *effects* (depicted in bold red). Using this representation, we enable the analysis using expected values, measured values, actions, and their effects. As we will show in the next section, this representation improves significantly the analysis and planning phases of traditional MAPE-K loops.

Coming back to our case study, Figure 5 depicts a model instance of our proposed example. The meta-model contains four metrics: container status (*contStatus1* and *contStatus2*) and container workloads (*containerWorkload1* and *containerWorkload2*). All these metrics are affected by two actions: *action1*, an action to switch a container on, or *action2*, an action to migrate a container to another host. As a scale up request has been created, the reasoning process decides to create an action plan: first switching a container on and then migrating another container.

3.4 Enabled reasoning

By associating the action model to the context model, we aim at enhancing adaptation process with new abilities to reason. In this section, we list four kinds of reasoning that is possible thanks to our approach: (i) answering change requests (ii) immediate supervision, (iii) future supervision, (iv) meta supervision. We also present the necessary code to extract information during the reasoning phase. We use a dot notation for navigating inside our context model, *i.e.,*

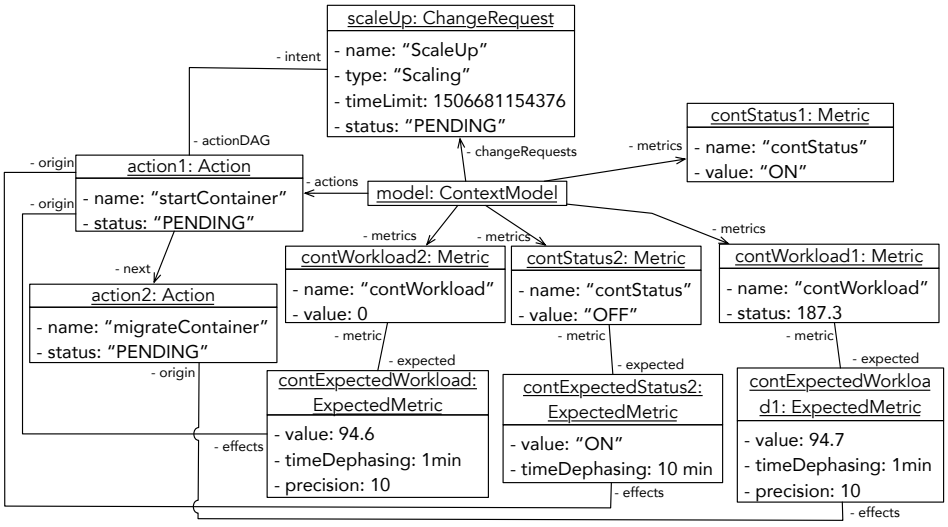

Figure 5: Instance model of the cluster management use cases.

```
model.changeRequests
     .select(ChangeRequest cr -> cr.status == "PENDING")
     .select(ChangeRequest cr -> cr.timeLimit <= NOW})
     .forEach(ChangeRequest cr -> createActions(cr))

//Rise an error for unsatisfied requests
model.changeRequests
     .select(ChangeRequest cr -> cr.status == "PENDING")
     .select(ChangeRequest cr -> cr.timeLimit > NOW)
     .forEach(ChangeRequest cr -> riseError(cr))
```

Listing 1: Generation of the DAG of actions from the change requested

for accessing the properties (attributes and relationships). Moreover, we consider the elements as stream, *i.e.,* each context elements are contained in a stream of elements. For example, the metrics are contained in a stream of context elements. In addition, context elements define the same operation as the ones of the Stream Java interface[1]. The execution semantics also remains similar. When the operations are executed, they are executed at a precise time, *i.e.,* the current timestamp. When operations need to access a context element, they apply the *resolve(id,time)* functions as described in Section 2. For readability purpose, we do not show the call to this function. Listed codes have been made using the context model presented in Figure 5.

Answering change requests. Thanks to change requests present in the context representation, the reasoning will create a DAG of actions to reach a desired state. It thus needs to extract the change requests that have not been processed and on which the time limit has not been reached. If the time limit has been reached or overtaken, the reasoning process could throw errors. We present the code to extract the change requests from our model in Listing 1.

Immediate Supervision. When a reasoning process plans adaptation actions, it also sets the expected effect on the metric of the context model. We refer to this operation as immediate supervision. However, the measured value could diverge from the expected one for different reasons: error in actions, context that has not changed as expected, or wrong effect extrapolation. The reasoning process needs to detect this divergence for taking counteractions. In other words, the reasoning process needs to analyse the difference between the expected value and the measured one. We present the code to achieve the extraction of the expected and the measured value of a metric in Listing 2. As shown in the code, the checking should consider both the precision of the expected value and the possible temporal dephasing between the measured value and the expected effects. We check if the value is in the following range: $[effectvalue - precision; effectvalue + precision]$ and the time of the measured value is the following range: $[time - timedephasing; time + timedephasing]$.

```
model.metrics
     .select(Metric m -> m.name == "contWorkload")
     .forEach(Metric m -> {
          ExpectedMetric e = m.expected;
          if(m.value < e.value - e.precision ||
             m.value > e.value + e.precision) {
               if(time(m.id) < start(e.id) - e.timeDephasing ||
                  time(m.id) > start(e.id) + e.timeDephasing) {
                    createCorrectiveActions(m); }
          }
     })
```

Listing 2: Detection of difference between measured and expected values

Future supervision. Actions are taken for adapting the system to the current context. However, they are not immediately and successfully executed and their effects are measurable with delay. As discussed in Section 1, the reasoning frequency could be inferior to this delay. It results on a reasoning process that will be executed on a similar context as the last execution. It will thus take the same action(s) whereas the context will be adapted soon. To tackle this problem, using our approach, the reasoning process can also consider the future expected metrics. By analysing the current

[1]https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html, Last visited: 28 September 2017

context for detecting any miss adaptation and the future expected one for detecting any future correction, the adaptation process will avoid repetition of actions. The code below show how to extract the current container workload and the future expected one. The code remains similar to the previous one (Listing 2) except that we resolve the expected value at another time, here 5 min after the current time (*NOW*).

```
model.metrics
      .select(Metric m -> m.name == "contWorkload")
      .forEach(Metric m -> {
          ExpectedMetric futureE = resolve(m.expected,NOW + 5min)
          //similar to the code presented in Listing 1
       })
```

Listing 3: Extraction of expected and measured value of a metric with a delta time

Meta supervision. Supervision allows to verify that the reasoning has been successfully executed, *i.e.,* that the expected effect really happened on the system. As explained in the previous paragraph, the system will also diverge from this desired state for different reasons. The meta supervision aims at finding extra information. For example, we will detect if a desired state is not reachable, or that an action often creates this divergence. The code presented in Listing 4 allows to detect actions that did not achieve the expected effect. For this purpose, we check if an action did not surpass the expected percentage *THRESHOLD* of miss expectations.

```
Map<Action,Integer> error= new HashMap();
int sum = 0;

model.timeRange(NOW - 2H, NOW)
      .metrics
      .forEach(Metric m -> {
         [...] //code of previous listing
         if(...) {// if presented in the previous listing
              Action a = m.expected.origin;
              int count = error.get(a) + 1;
              error.put(a, count);
              sum++;
         }
       });

error.forEach((key,value) -> {
          if(value / sum > THRESHOLD) {
              takeActions();
          }
    });
```

Listing 4: Extraction of expected and measured value of a metric with a delta time

4 VALIDATION & EVALUATION

In this paper, we argue that by adding action information directly within context representations the reasoning phase (*i.e.,* the *Analyse and Plan* of the MAPE-k loop) is improved. We validate the proposal using our case study example, the cloud management scenario. We demonstrate the scalability of our approach could by applying it to large-scale adaptive systems.

To implement our approach and build this experimentation we used the GreyCat framework[2]. It is a models@run.time framework with time as a built-in concept[7]. The code for this experimentation is open source and publicly accessible[3].

[2]http://greycat.ai/, Last visited: 21 September 2017
[3]https://bitbucket.org/ludovicpapers/sac-eval

4.1 Validation

In our experimentation, we simulate a cluster that executes a unique service on several containers. The adaptation process allocates or deallocates resources in/from the cluster infrastructure by relying on the measured workload of resource usage. The objective function is to minimize the number of used resources while reducing the difference between W_S and W_P. The targeted state is $W_S = W_P$.

We use the fastStorage[4] dataset[15] to simulate the service workload W_S. To reproduce a service workload, we sum up the CPU usage in the dataset. Then, the workload of the containers W_C is computed by dividing W_S by the number of available containers N_C. A maximum of 8,600[5] is set up to represent the allocated CPU of containers. Due to this maximum constraint, the sum of the container workloads ($W_P = \sum_{i=0}^{N_C} <= W_S$), also referred to as the provided workload of the service W_P, could be inferior to the workload of the service. In addition, the maximum value of the provided workload W_{Pmax} is equal to the sum of the maximum of the container: $W_{Pmax} = N_C * 8600$.

We compare the output results of similar adaptation processes but different sets of rules. The first set uses a traditional rule-based decision-making process [2, 3] while the second one replicates the rules and takes into consideration the expected effects of actions.

We identify two rules:

- **Rule 1** if $W_P < W_S$ then start as many containers as needed
- **Rule 2** if $W_{Pmax} > W_S$ then stop as many containers as needed

The equivalent rules in a temporal-aware context are depicted below:

- **Rule 1** if $W_P(now) < W_S(now)$ *and* $W_E(now + 3min) < W_S(now)$ then start $(W_S(now) - W_E(now + 3min)/8600$ containers
- **Rule 2** if $W_{Pmax}(now) > W_S(now)$ *and* $W_Pmax(now+3min) > W_S(now)$ then stop $(W_{Pmax}(now + 3min) - W_S(now))/8600$ containers

$W_E(t)$ represents the expected workload of a service at a time t, which is also the sum of the expected workload ($W_{CE}(t)$) of all the containers at the time t. We formalize it as: $W_E(t) = \sum_{i=0}^{N_C} W_{CE_i}(t)$.

The adaptation process is executed every 1 minute, for 100 times, in order to simulate the reactivity of the clusters. We show that rules that ignore the expected effects of actions perform poorly. We compute the number of start actions and the number of stop actions against the two sets of rules. Using the first set of rules, we reach a total of 433 start actions and 515 stop actions whereas using our approach 168 start actions and 235 stop actions. Thus, the second set of rules reduces the number of start and stop actions by a factor of ~2.6 and ~2.2 respectively. The difference between the provided workload and the service workload remains similar in the two usages.

4.2 Evaluation of reasoning latency

In this section, we evaluate the efficiency of our context representation for enhancing a reasoner implementing the previously defined rules (2nd set of rules). All the experimentations have been executed

[4]http://gwa.ewi.tudelft.nl/datasets/gwa-t-12-bitbrains, Last visited: 22 September 2017
[5]Average of allocated CPUs in MHz in the dataset

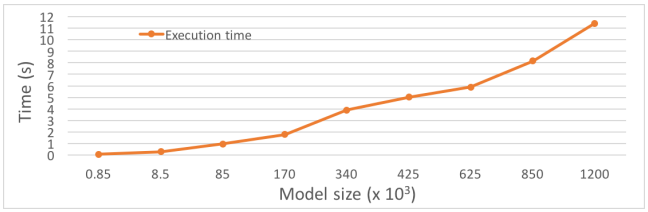

Figure 6: Latency to detect faults within temporal contexts given different sizes of cloud clusters

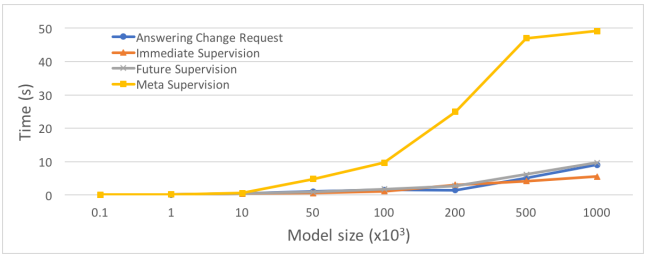

Figure 7: Execution time of the new reasoning abilities

in a MacBook Pro with an Intel Core i7 processor (2.8GHz, 4 cores, 16GB main memory (RAM), macOS Sierra version 10.12.4) using Oracle JDK 1.8.0_65. Our proposed temporal context representation introduces inescapably an overhead compared to a simple vector of metrics, as it is commonly used in cloud management implementations. Using an experimental evaluation, we answer this question by varying the size of the context model in order to simulate various sizes of clusters. Then, we evaluate during this experiment the latency of our cloud model-based reasoner to trigger corrective actions. The results are presented in Figure 6. The depicted results have been obtained by computing the average execution time over ten (10) runs.

Experimentally, we obtained a very good latency *(less than 200ms*, which is comparable to a single Remote Procedure Call [14]) for a cloud cluster with fewer than 1000 containers. For larger clusters, the latency remains acceptable *(less than 1s*) until reaching 85,000 elements. Finally, we evaluated a very large-scale cluster *(1,200,000 containers*), where we obtained a latency of less than 12s. In the later case, we obtained a throughput of 100,000 rules/second. The performance of our approach can be considered as very good as cloud clusters rarely go beyond few thousands of elements [15].

4.3 Evaluation of our reasoning element

In the last experimentation, we evaluate the four reasoning capacities presented in Section 3. Figure 7 depicts the average results of this experimentation of ten (10) executions. The first three reasoning operations, answering change requests, immediate supervision, and future supervision, give very good execution time. These operations can be evaluated these three operations over 1 million of elements in less than 10 seconds. As shown in Table 1, it results in a throughput of more than 100,000 operations per second. However, our results are less good for the meta-supervision. We can execute more than 20,000 of operations per second. This is due to the computational complexity of this supervision. Contrary to the previous reasoning that are executed at a precise time, the meta supervision is executed over a time range. But, the meta supervision operations should not be used with the same frequency as the three other ones. It could be used with less frequency as it analyses the data over bigger time range.

Table 1: Reasoning operation throughput

Reasoning operation	Throughput (op/s)
Answering change requests	110,702.74
Immediate Supervision	157,748.61
Future supervision	102,409.70
Meta supervision	20,345.88

5 RELATED WORK

The challenge implied by temporal representation is also raised by the scheduling and planning communities. In particular, reasoning activities that should schedule actions order is especially hard when durations are not stable and controllable. To tackle such a challenge, Cimatti *et al.,* [6] proposed an extension to state-space temporal planning algorithm to consider duration uncertainty. They discussed various heuristics to mitigate the risk of delayed planning, ultimately, to generate a more robust action planning. Our contribution tackles a different problem than the solving algorithm. We enable the representation of temporal uncertainty and alternatives within the context. However, such uncertainty-aware algorithm can take advantage of our context to improve the upcoming reasoning using for instance past executions feedback. Villegas *et al.,* [16] also studied the limitations of a control loop and proposed the use of dynamic context to deal with metric evolution. This approach improves reasoning by integrating feedback loops and context for upcoming reasoning task. Nonetheless, this approach does not store reasoning decisions and cannot help for long actions where effects are still not measurable.

More recently, Yuan *et al.,* [17] also address the challenge of temporal scheduling within a hybrid cloud environment. There are solutions focusing on maximizing the quality of service for task scheduling using time-varying prices. As many other approaches in cloud and grid computing, such approaches are limited to scheduling tasks that are easily moved from a worker to another. Unfortunately, long actions such as container migration, cannot rely on a high-frequency scheduling algorithm. For the best of our knowledge, existing solutions do not handle the context representation of time action associated with expected effect in order to ease data transmission between reasoning loop cycles.

6 CONCLUSION AND FUTURE WORK

Adaptive systems continuously need to adjust their configurations according to the current situation. A common approach to realize such reasoning and adaptation mechanism is the MAPE-K loop. However, this loop relies on context representations that aggregate all instantaneously measurable values, which—in the context of this work—we called metrics. Without considering past values and already planned but unfinished actions, traditional context representations lead to a weak analysis and planning phases in MAPE-K

loop-based approaches, and therefore to suboptimal adaptations. In this paper, we proposed an extended context representation to support the past and future actions, and more importantly, their related expected effect over time on the measurable context. Such approach enables reasoning questions like *"what should happen?"* instead of being limited to *"what happened"?* and *"what is measurable?"*. We leverage a temporal models@run.time paradigm to define at which time every action should start, end, and when effects are supposed to be measurable (duration of action). We demonstrated through a cloud management case study the suitability of such extended context representations to enable efficient reasoning.

As future work, we plan to extend the temporal representation of contexts to handle fuzzy and acceptance range in order to represent reasoning under uncertain conditions. Additionally, we expect to obtain a better result by offering the ability to precisely define the operational semantics of actions, and therefore, be able to predict other measures. To achieve this we plan to use "executable models". Finally, an important characteristic of metrics in context representation is the uncertainty. As future work, we intend to extend the expected effects with uncertainty definition.

REFERENCES

[1] 2005. *An Architectural Blueprint for Autonomic Computing.* Technical Report. IBM.

[2] Agnar Aamodt and Enric Plaza. 1994. Case-based reasoning: Foundational issues, methodological variations, and system approaches. *AI communications* 7, 1 (1994), 39–59.

[3] Darko Anicic, Paul Fodor, Sebastian Rudolph, Roland Stühmer, Nenad Stojanovic, and Rudi Studer. 2010. A rule-based language for complex event processing and reasoning. In *International Conference on Web Reasoning and Rule Systems*. Springer, 42–57.

[4] P. Arcaini, E. Riccobene, and P. Scandurra. 2015. Modeling and Analyzing MAPE-K Feedback Loops for Self-Adaptation. In *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. 13–23. https://doi.org/10.1109/SEAMS.2015.10

[5] Gordon S. Blair, Nelly Bencomo, and Robert B. France. 2009. Models@ run.time. *IEEE Computer* 42, 10 (2009), 22–27.

[6] Alessandro Cimatti and Andrea Micheli. 2015. Strong temporal planning with uncontrollable durations: a state-space approach. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*. AAAI Press, 3254–3260.

[7] Thomas Hartmann. 2016. *Enabling Model-Driven Live Analytics For Cyber-Physical Systems: The Case of Smart Grids.* Ph.D. Dissertation. University of Luxembourg. http://orbilu.uni.lu/handle/10993/28924

[8] Thomas Hartmann, François Fouquet, Matthieu Jimenez, Romain Rouvoy, and Yves Le Traon. 2017. Analyzing Complex Data in Motion at Scale with Temporal Graphs. In *The 29th International Conference on Software Engineering and Knowledge Engineering, Wyndham Pittsburgh University Center, Pittsburgh, PA, USA, July 5-7, 2017*. 596–601. https://doi.org/10.18293/SEKE2017-048

[9] Thomas Hartmann, François Fouquet, Grégory Nain, Brice Morin, Jacques Klein, Olivier Barais, and Yves Le Traon. 2014. A Native Versioning Concept to Support Historized Models at Runtime. In *Model-Driven Engineering Languages and Systems - 17th International Conference, MODELS 2014, Valencia, Spain, September 28 - October 3, 2014. Proceedings*. 252–268.

[10] Karen Henricksen, Jadwiga Indulska, and Andry Rakotonirainy. 2002. Modeling Context Information in Pervasive Computing Systems. In *Pervasive Computing, First International Conference, Pervasive 2002, Zürich, Switzerland, August 26-28, 2002, Proceedings*. 167–180. https://doi.org/10.1007/3-540-45866-2_14

[11] Ming Mao and Marty Humphrey. 2012. A Performance Study on the VM Startup Time in the Cloud. In *2012 IEEE Fifth International Conference on Cloud Computing, Honolulu, HI, USA, June 24-29, 2012*. 423–430. https://doi.org/10.1109/CLOUD.2012.103

[12] M. Maurer, I. Breskovic, V. C. Emeakaroha, and I. Brandic. 2011. Revealing the MAPE loop for the autonomic management of Cloud infrastructures. In *2011 IEEE Symposium on Computers and Communications (ISCC)*. 147–152. https://doi.org/10.1109/ISCC.2011.5984008

[13] Brice Morin, Olivier Barais, Jean-Marc Jézéquel, Franck Fleurey, and Arnor Solberg. 2009. Models@ Run.time to Support Dynamic Adaptation. *IEEE Computer* 42, 10 (2009), 44–51.

[14] Gavin Mulligan and Denis Gračanin. 2009. A comparison of SOAP and REST implementations of a service based interaction independence middleware framework. In *Winter Simulation Conference*. Winter Simulation Conference, 1423–1432.

[15] Siqi Shen, Vincent van Beek, and Alexandru Iosup. 2015. Statistical Characterization of Business-Critical Workloads Hosted in Cloud Datacenters. In *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2015, Shenzhen, China, May 4-7, 2015*. 465–474. https://doi.org/10.1109/CCGrid.2015.60

[16] Norha M Villegas, Gabriel Tamura, Hausi A Müller, Laurence Duchien, and Rubby Casallas. 2013. DYNAMICO: A reference model for governing control objectives and context relevance in self-adaptive software systems. In *Software Engineering for Self-Adaptive Systems II*. Springer, 265–293.

[17] Haitao Yuan, Jing Bi, Wei Tan, and Bo Hu Li. 2017. Temporal task scheduling with constrained service delay for profit maximization in hybrid clouds. *IEEE Transactions on Automation Science and Engineering* 14, 1 (2017), 337–348.