A Unified Modeling Framework to Abstract Knowledge of Dynamically Adaptive Systems

Ludovic Mouline

May 3, 2019

Abstract

Vision: As state-of-the-art techniques fail to model efficiently the evolution and the uncertainty existing in dynamically adaptive systems, the adaptation process makes suboptimal decisions. To tackle this challenge, modern modeling frameworks should efficiently encapsulate time and uncertainty as first-class concepts.

Context Smart grid approach introduces information and communication technologies into traditional power grid to cope with new challenges of electricity distribution. Among them, one challenge is the resiliency of the grid: how to automatically recover from any incident such as overload? These systems therefore need a deep understanding of the ongoing situation which enables reasoning tasks for healing operations. **Abstraction** is a key technique that provided an illuminating description of systems, their behaviors, and/or their environments alleviating their complexity. **Adaptation** is a cornerstone feature that enables reconfiguration at runtime for optimizing software to the current and/or future situation.

Abstraction technique is pushed to its paramountcy by the model-driven engineering (MDE) methodology. However, information concerning the grid, such as loads, is not always known with absolute confidence. Through the thesis, this lack of confidence about data is referred to as **data uncertainty**. They are approximated from the measured consumption and the grid topology. This topology is inferred from fuse states, which are set by technicians after their services on the grid. As humans are not error-free, the topology is therefore not known with absolute confidence. This data uncertainty is propagated to the load through the computation made. If it is neither present in the model nor not considered by the adaptation process, then the adaptation

process may make suboptimal reconfiguration decision.

The literature refers to systems which provide adaptation capabilities as dynamically adaptive systems (DAS). One challenge in the grid is the phase difference between the monitoring frequency and the time for actions to have measurable effects. Action with no immediate measurable effects are named **delayed action**. On the one hand, an incident should be detected in the next minutes. On the other hand, a reconfiguration action can take up to several hours. For example, when a tree falls on a cable and cuts it during a storm, the grid manager should be noticed in real time. The reconfiguration of the grid, to reconnect as many people as possible before replacing the cable, is done by technicians who need to use their cars to go on the reconfiguration places. In a fully autonomous adaptive system, the reasoning process should be considered the ongoing actions to avoid repeating decisions.

Problematic **Data uncertainty and delayed actions are not specific to smart grids.**

First, data are, almost by definition, uncertain and developers always work with estimates. Hardware sensors have by construction a precision that can vary according to the current environment in which they are deployed. A simple example is the temperature sensor that provides a temperature with precision to the nearest degree. Software sensors approximate also values from these physical sensors, which increases the uncertainty. For example, CPU usage is computed counting the cycle used by a program. As stated by Intel, this counter is not error-prone[1].

Second, it always exists a delay between the moment where a suboptimal state is detected by the adaptation process and the moment where the effects of decisions taken are measured. This delayed is due to the time needed by a computer to process data and, eventually, to send orders or data through networks. For example, migrating a virtual machine from a server to another one can take several minutes.

Through this thesis, I argue that this data uncertainty and this delay cannot be ignored for all dynamic adaptive systems. To know if the data uncertainty should be considered, stakeholders should wonder **if this data uncertainty**

[1] https://software.intel.com/en-us/itc-user-and-reference-guide-cpu-cycle-counter

affects the result of their reasoning process, like adaptation. Regarding delayed action, they should verify **if the frequency of the monitoring stage is lower than the time of action effects to be measurable**. These characteristics are common to smart grids, cloud infrastructure or cyber-physical systems in general.

Challenge These problematics come with different challenges concerning the representation of the knowledge for DAS. The global challenge address by this thesis is: **how to represent the uncertain knowledge allowing to efficiently query it and to represent ongoing actions in order to improve adaptation processes?**

Vision **This thesis defends the need for a unified modeling framework which includes, despite all traditional elements, temporal and uncertainty as first-class concepts.** Therefore, a developer will be able to abstract information related to the adaptation process, the environment as well as the system itself.

Concerning the adaptation process, the framework should enable abstraction of the actions, their context, their impact, and the specification of this process (requirements and constraints). It should also enable the abstraction of the system environment and its behavior. Finally, the framework should represent the structure, behavior and specification of the system itself as well as the actuators and sensors. All these representations should integrate the data uncertainty existing.

Contributions Towards this vision, this document presents two approaches: a temporal context model and a language for uncertain data.

The temporal context model allows abstracting past, ongoing and future actions with their impacts and context. First, a developer can use this model to know what the ongoing actions, with their expect future impacts on the system, are. Second, she/he can navigate through past decisions to understand why they have been made when they have led to a sub-optimal state.

The language, named Ain'tea, integrates data uncertainty as a first-class concept. It allows developers to attach data with a probability distribution which represents their uncertainty. Plus, it mapped all arithmetic and boolean operators to uncertainty propagation operations. And so, developers will automatically propagate the uncertainty of data without additional effort, compared to an algorithm which manipulates certain data.

Validation Each contribution has been evaluated separately. The language has been evaluated through two axes: its ability to detect errors at development time and its expressiveness. Ain'tea can detect errors in the combination of uncertain data earlier than state-of-the-art approaches. The language is also as expressive as current approaches found in the literature. Moreover, we use this language to implement the load approximation of a smart grid furnished by an industrial partner, Creos S.A.[2].

The context model has been evaluated through the performance axis. The dissertation shows that it can be used to represent the Luxembourg smart grid. The model also provides an API which enables the execution of query for diagnosis purpose. In order to show the feasibility of the solution, it has also been applied to the use case provided by the industrial partner.

Keywords: dynamically adaptive systems, knowledge representation, model-driven engineering, uncertainty modeling, time modeling

[2]Creos S.A. is the power grid manager of Luxembourg. `https://www.creos-net.lu`

Table of Contents

1

Introduction

Contents

Model-driven engineering methodology and dynamically adaptive systems approach are combined to tackle new challenges brought by systems nowadays. After introducing these two software engineering techniques, I give one example of such systems: the Luxembourg smart grid. I will also use this example to highlight two of the problematics: uncertainty of data and delays in actions. Among the different challenges which are implied by them, I present the global one addressed by the vision defended in this thesis: modeling of temporal and uncertain data. This global challenge can be addressed by splitting up in several ones. I present two of them, which are directly tackled by two contributions presented in this thesis.

Introduction

Use case: Luxembourg smart grid

Should contain: - veg iqu grqeub

General background

should contain: - MDE / metamodel / model - DAS

2

A temporal knowledge meta-model to represent, reason and diagnose decisions, their circumstances and their impacts

Contents

Adaptation processes are executed with a high frequency to react to any incident whereas the delay for decision applications are constrained by the time to execute the delayed actions. We identified two problems that result from these different paces. First, not considered unfinished actions, together with their expected effects, over time lead upcoming analysis phases potentially make suboptimal decisions. Second, explanations of adaptation processes remain challenging due to the lack of tracing ability of current approaches. To tackle this problem, we first propose a knowledge formalism to define the concept of a decision. Second, we describe a novel temporal knowledge model to represent, store and query decisions as well as their relationship with the knowledge

(context, requirements, and actions). We validate our approach through a use case based on the smart grid at Luxembourg. We also demonstrate its scalability both in terms of execution time and consumed memory.

Introduction

TODO: **We consider that decision, delayed action, context and knowledge have been defined in the global introduction**

Adaptive systems have proven their suitability to handle the increasing complexity of systems and their ever-changing environment. To do so, they make adaptation decisions, in the form of actions, based on high-level policies. For instance, the OpenStack Watcher project [**OpenStack:Watcher:Wiki**] implements the 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). For readability purpose, we refer to adaptation decision as decision in the remaining part of this document.

Despite the reactivity of adaptation processes, impacts of their decisions can be measurable long after they have been taken. We identified two problematics caused by this difference of paces:

- How to diagnose the self-adaptation process?
- How to enable reasoning over unfinished actions and their expected effects?

To address them, we propose a temporal knowledge model which can trace decisions over time, along with their circumstances and effects. By storing them, the adaptation process could consider ongoing actions with their expected effects, also called impacts. Plus, in case of faulty decisions, developers may trace back their effects to their circumstances. Our current approach is limited to the representation of measurable effects of any decision, and therefore action.

The meta-model allow structuring and storing the state and behavior of a running adaptive system, together with a high-level API to efficiently perform diagnosis routines. Our framework relies on a temporal model-based solution that efficiently abstracts decisions and their corresponding circumstances. Specifically, based on existing approaches for modeling and monitoring adaptation processes, we identify a set of properties that characterize context, requirements, and actions in self-adaptive systems. Then, we formalize the common core concepts implied in adaptation processes, also referred to as knowledge, by means of temporal graphs and a set of relations that trace decisions impact to circumstances. Finally, thanks to exposing common interfaces in adaptive processes, existing approaches in requirements and goal modeling engineering can be easily integrated into our framework.

The rest of this chapter is structured as follows. In the remaining part of this section, we motivate our approach, we summarize core concepts manipulated in adaptation processes, and we present a use case scenario based on the Luxembourg Smart Grid (*cf.* Chapter *TODO*: **add ref**). *TODO*: **Update with last version** Then, we provide a formal definition of these concepts in Section 2.2. Later, we describe the proposed data model in Section 2.3. In Section 2.4, we demonstrate the applicability of our approach by applying it to the smart grid example. We conclude this chapter in Section 2.6.

Motivation

Delayed action

In this section, we motivate the need to reason over delayed actions. To do so, we first give four examples of these actions. Then we detail why the effects of actions should be considered. Finally, we summarize and motivate the need for incorporating actions and their effects on the knowledge.

Delayed action examples Until here, we have claimed that adaptation processes should handle delayed actions. In order to show their existence, we give four different examples: two based on our use case, one on cloud infrastructure and a last one on smart homes. From our understanding, three phenomena can explain this delay: the time to execute an action(s) (Example 1), the time for the system to handle the new configuration (Example 3) and the inertia of the measured element (Example 2 and 4).

Example 1: Modification of fuse states in smart grids Even if the Luxembourg power grid is moving to an autonomous one, not all the elements can be remotely controlled. One example is fuses that still need to be open or closed by a human. Open and close actions in the Luxembourg smart grid both imply technicians who are contacted, drive to fuse places and manually change their states. If several fuses need to be changed due to one decision, only one technician will drive to them, sequentially,

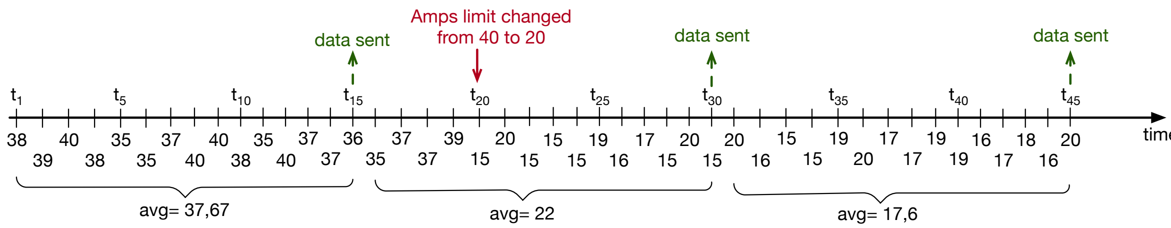


Figure 2.1: Example of consumption measurement before and after a limitation of amps has been executed at t_{20}.

and executes the modifications. For example, in our case, our industrial partner asks us to consider that each fuse modification takes 15 min whereas any incident should be detected in the minute. Let's imagine that an incident is detected at 4 p.m. and can be solved by modifying three fuses. The incidents will be seen as resolved by the adaptation process at 4 p.m. + 15 min * 3 = 4:45 p.m. In this case, the delay of the action is due to the execution time that is not immediate.

Example 2: Reduction of amps limit in smart grids[1] In its smart grid project, Creos S.A.envisages controlling remotely amps limits of customers. Customers will have two limits: a fixed one, set at the beginning, and a flexible one, remotely managed. The action to remotely change amps limits will be performed through specific plugs, such as one for electric vehicles. Even if the action is near instant, due to how power consumption is collected, its impacts would not be visible immediately. Indeed, data received by Creos S.A.corresponds to the total energy consumed since the installation. From this information, only the average of consumed data for the last period can be computed.

In Figure 2.1, we depict a scenario that shows the delay between the action is executed and the impacts are measured. Each time point represents one minute, with the consumption at this moment.

Let's imagine a customer who has his or her limit set to 40 amps[2] and consumes near this limit. We consider that data are sent every 15 min. After receiving data

[1]This example is based on randomly generated data. As this action is not yet available on the Luxembourg smart grid, we miss real data. However, it reflects an hypothesis shared with our partner.

[2]The user cannot consume more than 40 amps at a precise time t_i.

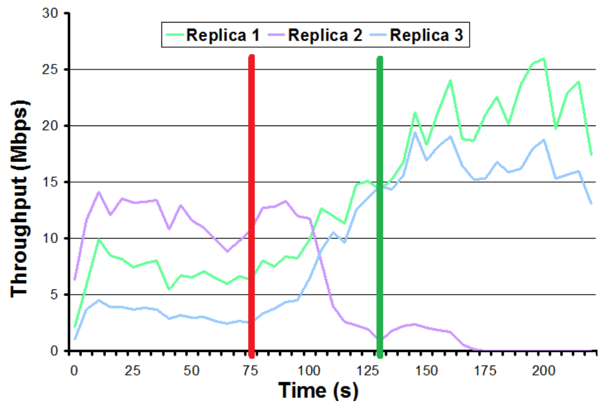


Figure 2.2: Figure extracted from [**DBLP:conf/nsdi/WangBR11**]. The red bar depicted the moment when Replica 2 stop receiving new connections. The green one represents the moment where all the rules in the load balancer stop considering R2. Despite these two actions, the throughput of the machine does not drop to 0 due to existing and active connections.

sent t_{15} and processing them, the adaptation process detects an overload and decides to reduce the limits to 20 amps for the customer. However, considering the delay for data to be collected and the one to send data[3], the action is received and executed at t_{20}. At t_{30}, new consumption data is sent, here equals 22 amps. Here, there are two situations. First, this reduction was enough to fix the overload. Even in this idealistic scenario, the adaptation process must wait at worst 15 min (t_{30} - t_{15}) to see the resolution (without considering the communication time). Second, this reduction was not enough - as the adaptation process considered that the consumption data will be at worst 20 amps and here it is 22. Before seeing the incident as solved and knowing that the decision fixed the incident, the adaptation process should wait for new data, sent at t_{45}, *i.e.*, around 30 min ($t_{45} - t_{15}$) after the detection.

In this case, the delay of this action can be explained by the inertia in the average of the consumption.

Example 3: Switching off a machine from a load balancer An example based on cloud infrastructure of delayed actions is to remove a machine from a

[3]Reminder: the smart grid is not built upon a fast network such a fiber network.

load balancer, for example during a scale down operation. Scale down operations allows cloud managers to reduce allocated resources for a specific task. It is used either to reduce the cost of the infrastructure or to reallocate them to other tasks. In [**DBLP:conf/nsdi/WangBR11**], Wang *et al.,* present a load-balancing algorithm. In their evaluation, they present the figure depicted in Figure 2.2 that shows the evolution of the throughput after the server Replica 2 (R2) is removing from the load balancer. The red bar shows the moment where R2 stop receiving new connection and the green the moment where it is removed from the load balancer algorithm. However, despite these actions have been taken, R2 should finish the ongoing tasks that it is executing. This explains why the throughout is progressively decreasing to 0 and there is a delay of around 100s between the red bars and the moment where R2 stop being active.

This example shows a delayed action due to the time required by the system to handle the new configuration.

Example 4: Modifying home temperature through a smart home system Smart home systems have been implemented in order to manage remotely a house or to perform automatically routines. For example, it allows users to close or open blinds from their smartphones. Based on instruction temperatures, smart home systems manage the heating or cooling system to reach them at the desired time. However, heating or cooling a house is not immediate, it can take several hours before the targeted temperature is reached. Plus, if the temperature sensor and the heating or cooling system are not placed nearby, the new temperature can take time before being measured. This can be explained due to the temperature inertia plus the delay for the temperature to be propagated.

Through these four examples, we show that delayed actions can be found in different kinds of systems, from CPS to cloud infrastructure. However, not only knowing that an action is running is important but also knowing its expecting effect. We detail this point in the following section.

The need to consider effects In the previous section, we show the existence of delayed actions. One may argue that action statuses are already integrated into the knowledge. For example, the OpenStack Watcher framework stores them in a database[4], accessible through an API. However, for the best of our knowledge Watcher does not store the expecting effects of each action. While the adaptation process knows what action is running, it does not know what it should expect from them.

Considering our example based on the modification of fuses, if the system knows that the technician is modifying fuse states, it does not know what would be the effects. In this case, when the adaptation process analyzes the system context it may wonder: what will be the next grid configuration? How the load will be balanced? Will the future configuration fix all the current incidents? If the effects are not considered by the adaptation process, then it may take suboptimal decisions.

Let's exemplify this claim through a scenario based on the fuse example (*cf.* Example 1). As explained before, the overload detected at 4 p.m. takes around 45 min to be fixed. The system marks this incident as "being resolved". In addition to this information, the knowledge contains another one saying that it is being solved by modifying three fuses. However, during the resolution stage, a cable is also being overloaded. The adaptation process has two solutions. It can either wait for the end of the resolution of the first incident to see if both overloaded elements will be fixed or it takes other actions without considering the ongoing actions and their impacts. Applying the first strategy may make the resolution of the second incident late, whereas the second one may generate a suboptimal sequence of actions. For example, the second modifications may undo what has been done before or both actions may be conflicting.

Conclusion Actions, like fuse modification in a smart grid or removing a server from a load balancer, generated during by adaptation processes could take time upon completion. Moreover, the expected effects resulting from such action is reflected in the context representation only after a certain delay. One 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

[4] https://docs.openstack.org/watcher/latest/glossary.html#watcher-database-definition

with support for past and future planned actions and their expected effects over time, we can highly enhance reasoning processes and avoid empirical tuning.

The research question that motivates our work is thus: how to enable reasoning over unfinished actions and their expected effects?

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 [**DBLP:journals/computer/MorinBJFS09**; **DBLP:journals/computer/BlairBF09**]. 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.

Diagnosis support

Faced with growingly complex and large-scale software systems (e.g. smart grid systems), we can all agree that the presence of residual defects becomes unavoidable [**DBLP:conf/icse/BarbosaLMJ17**; **DBLP:conf/icse/MongielloPS15**; **DBLP:conf/icse/H** Even with a meticulous verification or validation process, it is very likely to run into an unexpected behavior that was not foreseen at design time. Alone, existing formal modeling and verification approaches may not be sufficient to anticipate these failures [**DBLP:conf/icse/TaharaOH17**]. As such, complementary techniques need to be proposed to locate the anomalous behavior and its origin in order to handle it in a safe way.

As there might be many probable causes behind an abnormal behavior, developers usually perform a set of diagnosis routines to narrow down the scope or origin of the failure. One way to do so is by investigating the satisfaction of its requirements and the decisions that led to this system state, as well as their timing [**DBLP:conf/iceccs/BencomoWSW12**]. In this perspective, developers may set up a set of systematic questions that would help them understand why and how the system is behaving in such a way. These questions may comprise:

- what goal(s) the system was trying to reach by executing a tactic a?
- what were the circumstances used by a decision d and its expected impact on the

context?

- what decision(s) influenced the system's context at a time t?

Bencomo *et al.,* [**DBLP:conf/iceccs/BencomoWSW12**] argue that comprehensive explanation about the system behavior contributes drastically to the quality of the diagnosis, and eases the task of troubleshooting the system behavior. To enable this, we believe that adaptive software systems should be equipped with traceability management facilities to link the decisions made to their **(i) circumstances, that is to say, the history of the system states and the targeted requirements, and (ii) the performed actions with their impact(s) on the system**. In particular, an **adaptive system should keep a trace of the relevant historical events**. Additionally, it should be able to **trace the goals intended to be achieved by the system to the adaptations and the decisions that have been made, and vice versa**. Finally, in order to enable developers to interact with the system in a clear and understandable way, appropriate abstraction to **enable the navigation of the traces and their history should also be provided**. Unfortunately, suitable solutions to support these features are under-investigated.

Existing approaches [**hassel13**; **heinrich14**; **ehlers11**; **DBLP:conf/icse/MendoncaAR14**; **DBLP:conf/icse/CasanovaGSA14**; **DBLP:conf/icse/IftikharW14a**] are accompanied by built-in monitoring rules and do not allow to interact with the underlying system in a simple way. Moreover, they do not keep track of historical changes as well as causal relationships linking requirements to their corresponding adaptations. Only flat execution logs are stored.

Background

Before formalizing and modeling decisions and their circumstances, we abstract common concepts implied in an adaptation process. We refer to these concepts as the knowledge.

General concepts of adaptation process

Similar to the definition provided by Kephart [**DBLP:journals/computer/KephartC03**], IBM defines adaptive systems as "a computing environment with the ability to manage itself and **dynamically adapt** to change in accordance with **business policies and objectives**. [These systems] can perform such activities based on **situations they observe or sense in the IT environment** [...]" [**computing2006architectural**].

Based on this definition, we can identify three principal concepts involved in adaptation processes. The first concept is *actions*. They are executed in order to perform a dynamic adaptation through actuators. The second concept is **business policies and objectives**, which is also referred to as the **system requirements** in the domain of (self-)adaptive systems. The last concept is the observed or sensed **situation**, also known as the **context**. The following subsections provide more details about these concepts.

Context

In this thesis, we use the widely accepted definition of context provided by Dey [**DBLP:journals/puc**
"Context is **any information that can be used to characterize** the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and [the system], including the user and [the system] themselves". In this section, we list the characteristics of this information based on several works found in the literature [**DBLP:conf/pervasive/HenricksenIR02**; **chong2007context**; **DBLP:conf/seke/0001FNMKT14**; **bettini2010survey**; **DBLP:journals/co**
We use them to drive our design choices of our Knowledge meta-model (cf. Section *TODO*: **Add ref**).

Volatility Data can be either **static** or **dynamic**. Static data, also called frozen, are data that will not be modified, over time, after their creation [**DBLP:conf/pervasive/HenricksenIR0**
DBLP:journals/comsur/MakrisSS13; **bettini2010survey**; **chong2007context**]. For example, the location of a machine, the first name or birth date of a user can be identified as static data. Dynamic data, also referred to as volatile data, are data that will be modified over time.

Temporality In dynamic data, sometimes we may be interested not only in storing the latest value, but also the previous ones [**DBLP:conf/seke/0001FNMKT14**; **DBLP:conf/pervasive/HenricksenIR02**; **chong2007context**]. We refer to these data as **historical** data. Temporal data is not only about past values, but also future ones. Two kinds of future values can be identified, **predicted** and **planned**. Thanks to machine learning or statistical methods, dynamic data values can be **predicted**. **Planned** data are set by a system or a human to specify planned modification on the data.

Uncertainty One of the recurrent problems facing context-aware applications is the data uncertainty [**DBLP:conf/dagstuhl/LemosGMSALSTVVWBBBBCDDEGGGGIKKLMN** **DBLP:conf/pervasive/HenricksenIR02**; **DBLP:journals/comsur/MakrisSS13**; **bettini2010survey**]. Uncertain data are not likely to represent the reality. They contain a noise that makes it deviate from its original value. This noise is mainly due to the inaccuracy and imprecision of sensors. Another source of uncertainty is the behavior of the environment, which can be unpredictable. All the computations that use uncertain data are also uncertain by propagation.

Source According to the literature, data sources are grouped into two main categories, either sensed (measured) data or computed (derived) data [**DBLP:journals/comsur/PereraZC** **chong2007context**].

Connection Context data entities are usually linked using three kinds of connections: conceptual, computational, and consistency [**DBLP:conf/pervasive/HenricksenIR02**; **bettini2010survey**]. The conceptual connection relates to (direct) relationships between entities in the real world (e.g. smart meter and concentrator). The computational connection is set up when the state of an entity can be linked to another one by a computation process (derived, predicted). Finally, the consistency connection relates entities that should have consistent values. For instance, temperature sensors belonging to the same geographical area.

Requirement

Adaptation processes aim at modifying the system state to reach an optimal one. All along this process, the system should respect the **system requirements** established ahead. Through this paper, we use the definition provided by IEEE [**iso2017systems**]: "(1) Statement that translates or expresses a need and its associated **constraints** and **conditions**, (2) **Condition or capability that must be met or possessed** by a

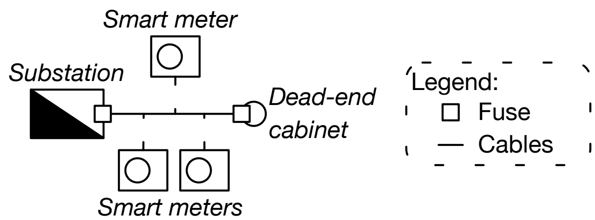


Figure 2.3: Simplified version of a smart grid

system [...] to satisfy an agreement, standard, specification, or other formally imposed documents".

Although in the literature, requirements are categorized as functional or non-functional, in this paper we use a more elaborate taxonomy introduced by Glinz [**DBLP:conf/re/Glinz07**]. It classifies requirements in four categories: functional, performance, specific quality, and constraint. All these categories share a common feature: they are all temporal. During the life-cycle of an adaptive system, the developer can update, add or remove some requirements [**DBLP:conf/icse/ChengA07**; **pandey2010effective**].

Action

In the IEEE Standards [**iso2017systems**], an action is defined as: "**process of transformation** that **operates upon data** or other types of inputs to create data, produce outputs, or **change the state** or condition of the subject software".

Back to adaptive systems, we can define an action as a process that, given the context and requirements as input, adjusts the system behavior. This modification will then create new data that correspond to an output context. In the remainder of this paper, we refer to output context as impacted context, or simply impact(s). Whereas requirements are used to add preconditions to the actions, context information is used to drive the modifications. Actions execution have a start time and a finish time. They can either succeed, fail, or be canceled by an internal or external actor.

Use case scenario

In order to provide a readable and understandable example of the formalism, we give a simplified version of the use case presented in Section 1.2.

Excerpt of a smart grid Figure 2.3 shows a simplified version of a smart grid with one substation, one cable, three smart meters and one dead-end cabinet. Both the substation and the cabinet have one fuse each. The meters regularly send consumption data at the same timestamp. For this example, we consider one requirement: minimizing the number of overloads. To achieve so, among the different actions, two actions are taken into account in this example: decreasing or increasing the amps limits of smart meters.

Scenario The system starts at t_0 with the actions, the requirements and all element of the context that remain fixed: the grid installation. Meters send their values at t_1, t_2 and t_3. Based on these data, the load on cables and substation is computed. On t_1, an overload is detected on the cable, which breaks the requirement. At the same time point, the system decides to reduce the load of all smart meters. The impact of these actions will be measured at t_2 and t_3, *i.e.,* the consumption will slowly reduce until the cable is no longer overloaded from t_3.

Knowledge formalization

As discussed previously, we consider knowledge to be the association of context information, requirements, and action information, all in one global and unified model. While context information captures the state of the system environment and its surroundings, the system requirements define the constraints that the system should satisfy along the way. Actions, on the other hand, are meant to reach the goals of the system.

In this section, we provide a formalization of the knowledge used by adaptation processes based on a temporal graph. Indeed, due to the complexity and interconnectivity of system entities, graph data representation is an appropriate way to represent the knowledge. Augmented with a temporal dimension, temporal graphs are then able to symbolize the evolution of system entities and states over time. We benefit from the well-defined graph manipulation operations, namely temporal graph pattern matching and temporal graph relations to represent the traceability links between the decisions made and their circumstances.

Before describing this formalism, we describe the semantics used for the temporal

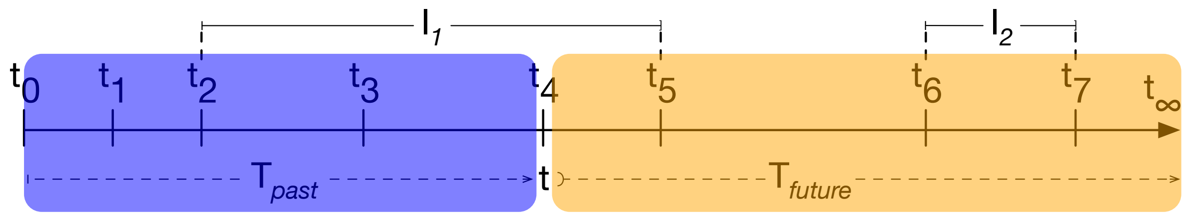


Figure 2.4: Time definition used for the knowledge formalism

axis. Then, we exemplify the knowledge formalism using the Luxembourg smart grid use case, detailed in Section 2.1.3.

Formalization of the temporal axis

The formalism described below has been made with two goals in mind. First, the definition of the time space should allow the distinction between past and future. Doing this distinction enable the differentiation between measured data and predicted (or planned data). Second, it should permit the definition of the life cycle of an element of the knowledge, which can be seen as a succession of states with a validity period that should not overlap each other.

Time space T is considered as an ordered discrete set of time points non-uniformly distributed. As depicted in Figure 2.4, this set can be divided into 3 different subsets $T = T_{past} \cup \{t\} \cup T_{future}$, where:

- T_{past} is the subdomain $\{t_0;t_1;...;t_{current-1}\}$ representing graph data history starting from t_0, the oldest point, until the current time, t, excluded.
- $\{t\}$ is a singleton representing the current time point
- T_{future} is subdomain $\{t_{current+1};\ldots;t_\infty\}$ representing future time points

The three domains depend completely on the current time {t} as these subsets slide as time passes. At any point in time, these domains never overlap: $T_{past} \cap \{t\} = \emptyset$, $T_{future} \cap \{t\} = \emptyset$, and $T_{past} \cap T_{future} = \emptyset$. The definition of these three subsets reaches the first goal.

In addition, there is a right-opened time interval $I \in T \times T$ as $[t_s, t_e)$ where $t_e - t_s > 0$. In English words, it means that the interval should represent at least one time point and should follow the time order. For any $i \in I$, $start(i)$ denotes its lower bound and

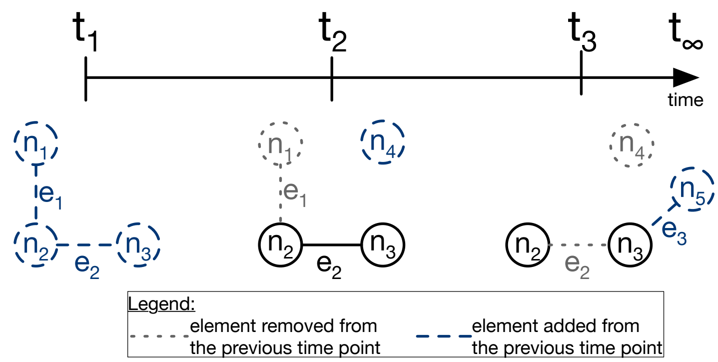


Figure 2.5: Evolution of a temporal graph over time

$end(i)$ its upper bound. As detailed in Section 2.2.2, these intervals are used to define the validity period for each node of the graph (our second goal).

Figure 2.4 displays an example of a time space $T_1 = \{t_0, t_1, t_2, t_3, t_4, t_5, t_6, t_7\}$. Here, the current time is $t = t_4$. According to the definition of the past subset (T_{past}) and the future one (T_{future}), there is: $T_{past1} = \{t_0, t_1, t_2, t_3\}$ and $T_{future1} = \{t_5, t_6, t_7\}$. Two intervals have been defined on T_1, namely I_1 and I_2. The first one starts at t_2 and ends at t_5 and the last one is defined from t_6 to t_7. As shown with I_1, an interval could be defined on different subsets, here it is on all of them (T_{past}, t, and T_{future}).

Formalism

Graph definition First, let K be an adaptive process over a system knowledge represented by a graph such as $K = (N, E)$, comprising a set of nodes N and a set of edges E. Nodes represent any element of the knowledge (context, actions, *etc.*) and edges represent their relationships. Nodes have a set of attribute values: $\forall n \in N, n = (id, P)$, where P is the set of key-value attributes. An attribute value has a type (numerical, boolean, ...). Every relationship $e \in E$ can be considered as a couple of nodes with a label $(n_s, n_t, label) \in N \times N$, where n_s is the source node and n_t is the target node.

Adding the temporal dimension In order to augment the graph with a temporal dimension, the relation V^T is added. So now the knowledge K is defined as a temporal graph such as $K = (N, E, V^T)$.

A node is considered valid either until it is removed or until one of its attributes value changes. In the latter case, a new node with the updated value is created. Whilst, an edge is considered valid until either its source node and target node are valid, or until the edge itself is removed. Otherwise, nodes and edges are considered invalid. The temporal validity relation is defined as $V^T : N \cup E \rightarrow I$. It takes as a parameter a node or an edge ($k \in N \cup E$) and returns a time interval ($i \in I$, *cf.* Section **??**) during which the graph element is valid.

Figure 2.5 shows an example of a temporal graph K_1 with five nodes (n_1, n_2,n_3, n_4, and n_5) and three edges (e_1, e_2, and e_3) over a lifecycle from t_1 to t_3. In this way, K_1 equals $(\{n_1, n_2, n_3, n_4, n_5\}, \{e_1, e_2, e_3\}, V_1^T)$. Let's assume that the graph is created at t_1. As n_1 is modified at t_2, its validity period starts at t_1 and ends at t_2: $V_1^T(n_1) = [t_1, t_2)$. n_2 and n_3 are not modified; their validity period thus starts at t_1 and ends at t_∞: $V_1^T(n_2) = V_1^T(n_3) = [t_1, t_\infty)$. Regarding the edges, the first one, e_1, is between n_1 and n_2 and the second one, e_2 from n_2 to n_3. Both are created at t_1. As n_1 is being modified at t_2, its validity period goes from t_1 to t_2: $V_1^T(e_1) = [t_1, t_2)$. e_2 is deleted at t_3. Its validity period is thus equal to: $V_1^T(e_2) = [t_1, t_3)$.

Lifecycle of a knowledge element One node represents the state of exactly one knowledge element during a period named the validity period. The lifecycle of a knowledge element is thus modeled by a unique set of nodes. By definition, the validity periods of different nodes cannot overlap. A same time period cannot be represented by two different nodes, which could create inconsistency in the temporal graph.

To keep track of this knowledge element history, the Z^T relation is added to the graph formalism: $K = (N, E, V^T, Z^T)$. It serves to trace the updates of a given knowledge element at any point in time. This relation can also be seen as a temporal identity function which takes as parameters a given node $n \in N$ and a specific time point $t \in T$, and returns the corresponding node at that point. Formally, $Z^T : N \times T \rightarrow N$.

In order to consider this new relation in the example presented in Figure 2.5, the definition of K_1 is modified to $K_1 = (\{n_1, n_2, n_3, n_4, n_5\}, \{e_1, e_2, e_3\}, V_1^T, Z_1^T)$ In Figure 2.5, let's imagine that n_1, n_4, and n_5 represent the same knowledge element k_e. The lifecycle of k_e is thus:

- n_1 for period $[t_1, t_2)$,

- n_4 for period $[t_2, t_3)$,
- n_5 for period $[t_3, t_\infty)$.

Let t'_1 be a timepoint between t_1 and t_2. When one wants to resolve the node representing the knowledge element at t'_1, she or he gets n_1 node, no matter of the node input (n_1, n_4, or n_5): $Z_1^T(n_4, t_1) = n_1$. On the other hand, applying the same relation with another node (n_2 or n_3) returns another node. For example, if n_2 and n_3 do not belong to the same knowledge element, then it will return the node given as input, for example $Z_1^T(n_2, t_1) = n_2$.

Knowledge elements stored in nodes Nodes are used to store the different knowledge elements: context, requirements and actions. The set of nodes N is thus split in three subsets: $N = C \cup R \cup A$ where C is the set of nodes which store context information, R a set of nodes for requirement information and A a set of nodes for action information.

Actions define processes that indirectly impact the context: they will change the behavior of the system, which will be reflected in the context information. Requirements are also processes that are continuously run over the system in order to check the specifications. Here, the purpose of the A and R subset is not to store these processes but to list them. It can be thought as a catalogue of actions and requirements, with their history.

Using a high-level overview, these processes can be depicted as: taking the knowledge as input, perform tasks, and modify this knowledge as output. As detailed in the next two paragraphs, action executions and requirement analysis can be formalized by relations.

Temporal queries for requirements At the current state, the formalism of the knowledge K does not contain any information regarding the requirement analysis. To overcome this, system requirements analysis R_A are added such as $K = (N, E, V^T, Z^T, R_A)$. R_A is a set of couples composed of patterns $P_{[t_j,t_k]}(K)$ and requirements R over these patterns: $R_P = P \cup R$.

$P_{[t_j,t_k]}$ denotes a temporal graph pattern, where t_j and t_k are the lower and upper bound of the time interval respectively. $P_{[t_j,t_k]}$ is the result of a function which takes the knowledge and an interval as input: $P_{[t_j,t_k]} : K \times I$. The time interval can be either fixed (absolute), *i.e.,* both bounds are precisely defined, or sliding (relative), *i.e.,* the upper bound is computed from the lower bound. For example, $P_{[t_0,t_4]}$ is considered as fixed and $P_{[t_0,t_0+4]}$ is considered as relative. Each element of the pattern should be valid for at least one timepoint: $\forall\ p \in P_{[t_j,t_k)}, V^T(e) \cap [t_j, t_k) \neq \emptyset$. Patterns can be seen as temporal subgraphs of K, with a time limiting constraint coming in the form of a time interval.

Temporal relations for actions Like for R_A, the knowledge K needs to be augmented with action executions A_E: $K = (N, E, V^T, Z^T, R_A, A_E)$. Actions executions A_E can be regarded as a couple (A, A_F), where A is the action that is executed and A_F a set of relations or isomorphisms mapping a source temporal graph pattern $P_{[t_j,t_k]}$ to a target one $P_{[t_l,t_m]}$, $A_F : K \times I \to K \times I$.

The left-hand side of the A_F relation depicts the temporal graph elements over which an action is applied. Every relation may have a set of application conditions. They describe the circumstances under which an action should take place. These application conditions are either positive, should hold, or negative, should not hold. Application conditions come in the form of temporal graph invariants. The side effects of these actions are represented by the right-hand side.

Finally, we associate to A_E a temporal function E_{A_E} to determine the time interval at which an action has been executed. Formally, $E_{A_E} : A_E \to I$.

Temporal relations for decisions Finally, the knowledge formalism needs to include the last, but not the least, element: decisions made by the adaptation, $K = (N, E, V^T, Z^T, R_A, A_E, D)$ While the source of relations in D represents the state before the execution of an action, the target shows its impact on the context. Its intent is **to trace back impacts of action executions to the decisions they originated from**.

A decision present in D is defined as a set of actions executed, *i.e.,* a subset of A_E, combined with a set of requirement analysis, *i.e.,* a subset of R_A. Formally, $D = \{\ A_D \cup R_D \mid A_D \subseteq A_E, R_A \subseteq R_P\}$. We assume that each action should result from only one decision: $\forall a \in A, \forall d1, d2 \in D \mid a \in d1 \wedge a \in d2 \to d1 = d2$.

The temporal function E_{A_E} is extended to decisions in order to represent the execution time: $E_{A_E} : (A \cup D) \rightarrow I$. For decision, the lower bound of the interval corresponds to the lowest bound of the action execution intervals. Following the same principle, the upper bound of the interval corresponds to the uppermost bound of the action execution intervals. Formally, $\forall d \in D \rightarrow E_{A_E}(d) = [l, u)$, where $l = \min\limits_{a \in A_d}\{E_{A_E}(a)[start]\}$ and $u = \max\limits_{a \in A_d}\{E_{A_E}(a)[end]\}$.

Sum up Knowledge of an adaptive system can be formalism with a temporal graph such as $K = (N, E, V^T, Z^T, R_A, A_E, D)$, wherein:

- N is a set of nodes to represent the different information (context, actions and requirements)
- E is a set of edges which connects the different nodes,
- V^T is a temporal relation which defines the temporal validity of each element,
- Z^T is a relation to track the history of each knowledge elements,
- R_A is a relation that defines the different requirements processes,
- A_E is a relation that defines the different action processes,
- D is a set of action executions, which result from the same decision, and requirement analysis.

Decisions D can allow adaptation processes to reason over ongoing and future executions of decisions. Moreover, it allows tracing the state of the knowledge before and after the decision has been or is executed, thanks to its A_D component. Plus, it represents which action has been used for this. Thanks to the R_A relation, one can access the requirements at the root of the decision and the state of the knowledge used by this requirement.

In the next section, we exemplify this formalism over our case study.

Application on the use case

In this section we apply the formalism described on the use case presented in Section 2.1.3.

Let K_{SG} be the temporal graph that represents the knowledge of this adaptive system: $K_{SG} = (N_{SG}, E_{SG}, V_{SG}^T, Z_{SG}^T, R_{P_{SG}}, A_{P_{SG}}, D_{SG})$. Figure 2.6 shows the nodes

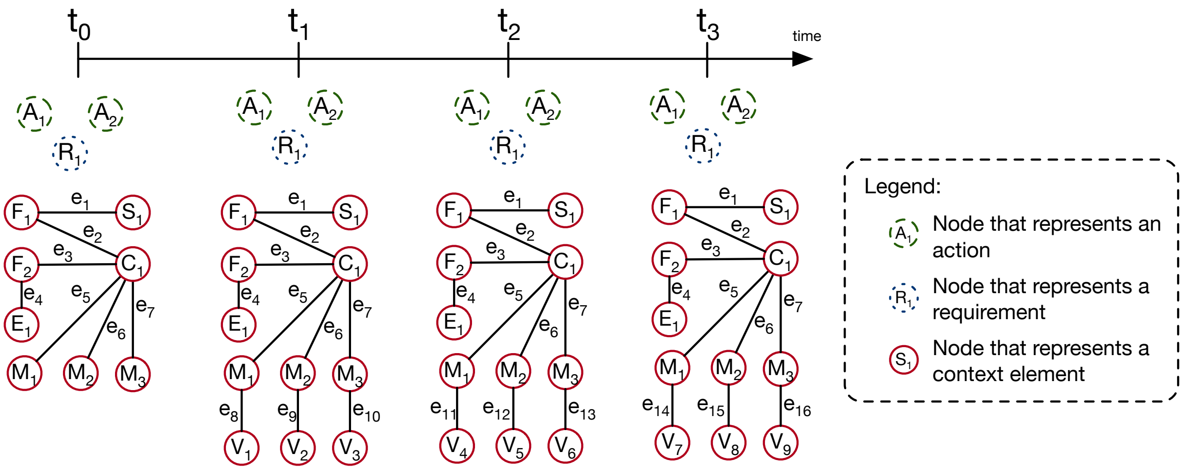
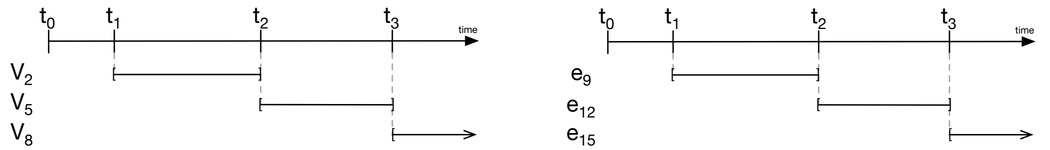


Figure 2.6: Application of the formalism with a temporal graph that represents the knowledge of the smart grid described in Section 2.1.3

and edges of this knowledge.

Description of N_{SG} N_{SG} is divided into three subsets: C_{SG}, R_{SG} and A_{SG}. R_{SG} contains one node, R_1 in Figure 2.6, which represents the requirement of this example (minimizing the number of overloads): $R_{SG} = \{R_1\}$. Two nodes, A_1 and A_2, belong to A_{SG}: $A_{SG} = \{A_1, A_2\}$. They represent the two actions of this example, respectively decreasing and increasing amps limits. Regarding the context C_{SG}, there are three nodes to represent the three smart meters (M_1, M_2, and M_3), one for the substation (S_1), two for the fuses (F_1 and F_2), one for the dead-end cabinet (E_1), one for the cable (C_1) and one node per consumption value received (V_i): $C_{SG} = \{M_1, M_2, M_3, S_1, F_1, F_2, E_1, C_1\} \cup \{V_i | i \in [1..9]\}$.

According to the scenario, except for nodes to store consumption values, the other nodes are created at t_0 and are never modified. Therefore, their validity period starts at t_0 and never ends: $\forall n \in A_{SG} \cup R_{SG} \cup \{M_1, M_2, M_3, S_1, F_1, F_2, E_1, C_1\}, V^T_{SG}(n) = [t_0, t_\infty)$. Considering the consumption values, all the nodes represent the history of the values for the three smart meters. In other words, there are three knowledge elements: the consumption measured for each meter. Let C_i notes the consumption measured by the smart meter M_i. As shown in Figure **??**, there is:



(a) Consumption values C_2

(b) Edges linking the meter node M_2 to its consumption values C_2

Figure 2.7: Validity periods of consumption values and their edges to the smart meter M_2

- C_1 of M_1 is represented by $\{V_1, V_4, V_7\}$,
- C_2 of M_2 is represented by $\{V_2, V_5, V_8\}$,
- C_3 of M_3 is represented by $\{V_3, V_5, V_9\}$.

Taking C_2 as an example, V_2 is the initial consumption value, replaced by V_5 at t_2, itself replaced by V_8 at t_3. Applying the V_{SG}^T on these different values, results are thus:

- $V_{SG}^T(V_2) = [t_1, t_2)$,
- $V_{SG}^T(V_5) = [t_2, t_3)$,
- $V_{SG}^T(V_8) = [t_3, t_\infty)$.

These validity periods are shown in Figure 2.7a. As meters send the new consumption values at the same time, this example can also be applied to C_1 and C_3.

From these validity periods, the Z_{SG}^T can be used to navigate to the different values over time. Let's continue with the same example, C_2. In order to get the evolution of the consumption value C_2, given the initial one, one will use the Z_{SG}^T relation:

- $Z_{SG}^T(V_2, t_{s1}) = \emptyset$, where $t_0 \leqslant t_{s1} < t_1$
- $Z_{SG}^T(V_2, t_{s2}) = V_2$, where $t_1 \leqslant t_{s2} < t_2$
- $Z_{SG}^T(V_2, t_{s3}) = V_5$, where $t_2 \leqslant t_{s3} < t_\infty$.
- $Z_{SG}^T(V_2, t_{s4}) = V_8$, where $t_2 \leqslant t_{s4} < t_\infty$.

Description of $\boldsymbol{E_{SG}}$ In this example, edges are used to store the relationships between the different context elements. For example, the edge between the substation S_1 and the fuse F_1 allow representing the fact that the fuse is physically inside the substation. Another example, edges between the cable C_1 and the meters M_1, M_2 and M_3 represent the fact that these meters are connected to the smart grid through this cable.

One may consider that relations (validity, Z^T, decisions, action executions and requirements analysis) will be stored as edges. But this decision is let to the implementation part of this formalism.

In our model, only consumption values (V_i nodes) are modified over time. Plus, since the scenario does not imply any edge modifications, only those between meters and values are modified. The edge set contains thus sixteen edges: $E_{SG} = \{e_i \mid i \in [1..16]\}$.

By definition, the unmodified edges have a validity period starting from t_0 and never ends: $\forall i \in [1..7], V^T_{SG}(e_i) = [t_0, t\infty)$. The history of the three knowledge elements that represent consumption values do not only impact the nodes which represent the values but also the edges between those nodes and the meters ones:

- C_1 impacts edges between M_1 and V_1, V_4, and V_7, *i.e.*, $\{e_8, e_{11}, e_{14}\}$,
- C_2 impacts edges between M_2 and V_2, V_5, and V_8, *i.e.*, $\{e_9, e_{12}, e_{15}\}$,
- C_3 impacts edges between M_3 and V_3, V_6, and V_9, *i.e.*, $\{e_{10}, e_{13}, e_{16}\}$.

Continuing with C_2 as an example, the initial edge value is e_9 from t_1, which is replaced by e_{12} from t_2, itself replaced by e_{15} from t_2. The validity relation, applied to these edges, thus returns:

- $V^T_{SG}(e_9) = [t_1, t_2) = V^T_{SG}(V_2)$,
- $V^T_{SG}(e_{12}) = [t_2, t_3) = V^T_{SG}(V_5)$,
- $V^T_{SG}(e_{15}) = [t_3, t_\infty) = V^T_{SG}(V_8)$,

These validity periods are depicted in Figure 2.7b. As they are driven by those of consumption values (V_2, V_5, and V_8), they are equal.

As for nodes, the Z^T_{SG} relation can navigate over time through these values. For example, to get the history of the edges between the consumption value C_2 and the meter represented by M_2, one can apply the Z^T_{SG} relation as follows:

- $Z^T_{SG}(e_9, t_{s1}) = \emptyset$, where $t_0 \leqslant t_{s1} < t_1$
- $Z^T_{SG}(e_9, t_{s2}) = e_9$, where $t_1 \leqslant t_{s2} < t_2$,
- $Z^T_{SG}(e_9, t_{s3}) = e_12$, where $t_2 \leqslant t_{s3} < t_3$,
- $Z^T_{SG}(e_9, t_{s4}) = e_15$, where $t_3 \leqslant t_{s4} < t_\infty$.

Description of D_{SG}, $A_{E_{SG}}$, and $R_{A_{SG}}$ As described in the scenario (cf. Section 2.1.3), the requirement analysis detects that t_1 the requirement is broken. The adaptation process will thus apply the "decreasing amps limits" action on the three meters. Following Example 2 detailed in Section 2.1.1, we consider that the action will impact the consumption values on the next two measurements: t_2 and t_3.

In the knowledge, we thus have one decision: $D_{SG} = D_1$. This decision has been taken after one requirement analysis, $R_{A_{SG1}}$, that detects no respect of the requirement R_1. To determine if there is an overload, this analysis needs to know the topology and the consumption values. The pattern is thus defined by all nodes related to the grid network and consumption values at $t1$: $P_{1[t_1,t_1+1]} = \{S_1, F_1, F_2, C_1, E_1, M_1, M_2, M_3, V_1, V_2, V_3\}$. So we have: $R_{A_{SG1}} = \{R_1, P_{1[t_1,t_1+1]}\}$.

The knowledge also includes the three action executions: $A_{E_{SG1}}$, $A_{E_{SG2}}$, and $A_{E_{SG3}}$. These actions have been executed on, respectively, M_1, M_2, and M_3. Following the definition, they all contain the action A_1 and similar relation which linked the circumstances to the impacts. The circumstances are the state of the knowledge at t_0, which contain all information of the grid network and the consumption values. We denote them $P_{2[t_1,t_1+1]}$, $P_{3[t_1,t_1+1]}$, and $P_{4[t_1,t_1+1]}$, all equal $P_{1[t_1,t_1+1]}$. The impact contains all consumption values received at t_2 and t_3. Each action impacts the consumption value of the meter that it modifies. For example, $A_{E_{SG2}}$ only impacts values of meter M_2. For this action, the output pattern is thus : $P_{5[t_2,t_3]} = \{V_5, V_8\}$. In summary, $A_{E_{SG1}}$, $A_{E_{SG2}}$, and $A_{E_{SG3}}$ are defined as follows:

- for the action executed on M_1: $A_{E_{SG1}} = (A_1, A_{F1})$, with $A_{F1} : P_{2[t_1,t_1+1]} \rightarrow \{V_4, V_7\}$,
- for the action executed on M_2: $A_{E_{SG2}} = (A_1, A_{F2})$, with $A_{F2} : P_{3[t_1,t_1+1]} \rightarrow \{V_5, V_8\}$,
- for the action executed on M_3: $A_{E_{SG3}} = (A_1, A_{F3})$, with $A_{F3} : P_{4[t_1,t_1+1]} \rightarrow \{V_6, V_9\}$,

The decision described in the scenario is thus equal to: $D_1 = \{R_{A_{SG1}}, A_{E_{SG1}}, A_{E_{SG2}}, A_{E_{SG3}}\}$. At t_2, this decision will still be valid. The adaptation process can thus include it in the adaptation process to reason over the ongoing actions. If at t_3 the cable remains overloaded, then one may use this element to check if the system tried to fix it, how and based on which information.

Modeling the knowledge

In order to simplify the diagnosis of adaptive systems, this thesis proposes a novel metamodel that combines, what we call, design elements and runtime elements. Design elements abstract the different elements involved in knowledge information to assist the specification of the adaptation process. Runtime elements instead, represent the data collected by the adaptation process during its execution. In order to maintain the consistency between previous design elements and newly created ones, instances of design elements (*e.g.,* actions) can be either added or removed. Modifying these elements would consist in removing existing elements and creating new ones. Combining design elements and runtime elements in the same model helps not only to acquire the evolution of system but also the evolution of its structure and specification (e.g. evolution of the requirements of the system). Design time elements are depicted in gray in the Figures 2.8– 2.11. Note that, this thesis does not address how runtime information is collected.

For the sake of modularity, the metamodel has been split into four packages: Knowledge, Context, Requirement and Action. All the classes of these packages have a common parent class that adds the temporality dimention: *TimedElement* class. Before describing the Knowledge (core) package, we detail this element. Then, we introduce in more details the other three packages used by the Knowledge package: Context, Requirement, and Action. In below sections, we use "*Package::Class*" notation to refer to the provenance of a class. If the package is omitted, then the provenance package is this one described by the figure or text.

Parent element: *TimedElement* class

we assume that all the classes in the different packages extend a *TimedElement* class. This class contains three methods: $startTime$, $endTime$, and $modificationsTime$. The first two methods allow accessing the validity interval bounds defined by the previously discussed V^T relation. The last method resolves all the timestamps at which an element has been modified: its history. This method is the implementation of the relation Z^T described in our formalism (cf. Section 2.2.2).

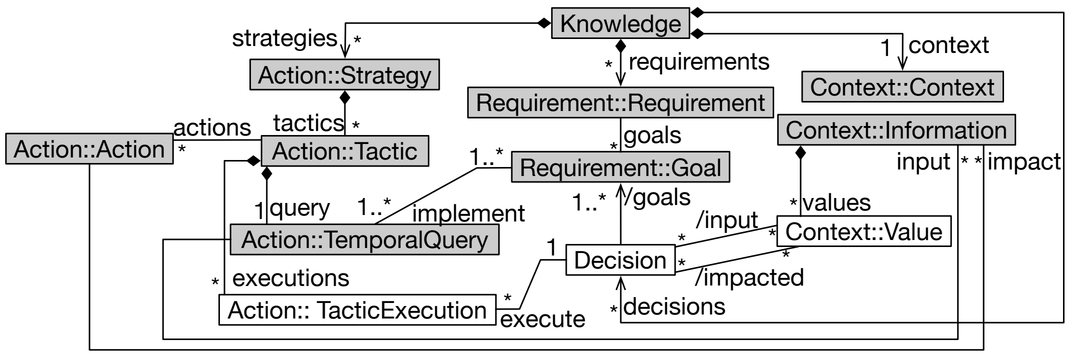


Figure 2.8: Excerpt of the knowledge metamodel

Knowledge metamodel

In order to enable interactive diagnosis of adaptive systems, traceability links between the decisions made and their circumstances should be organized in a well-structured representation. In what follows, we introduce how the knowledge metamodel helps to describe decisions, which are linked to their goals and their context (input and impact). Figure 2.8 depicts this metamodel.

Knowledge package is composed of a *context*, a set of *requirements*, a set of *strategies*, and a set of *decisions*. A decision can be seen as the output of the Analyze and Plan steps in the Monitor, Analyze, Plan, and Execute over knowledge (MAPE-k) loop.

Decisions comprise target *goals* and trigger the execution of one *tactic* or more. A decision has an *input* context and an *impacted* context. The context impacted by a decision (*Decision.impacted*) is a derived relationship computed by aggregating the impacts of all actions belonging to a decision (see Fig. 2.11). Likewise, the *input* relationship is derived and can be computed similarly. In the smart grid example, a decision can be formulated (in plain English) as follows: since the district D is almost overloaded (*input context*), we reduce the amps limit of greedy consumers using the "*reduce amps limit*" *action* in order to reduce the load on the cable of the district (*impact*) and satisfy the "*no overload*" policy (*requirement*).

As all the elements inherit from the *TimedElement*, we can capture the time at which a given decision and its subsequent actions were executed, and when their impact

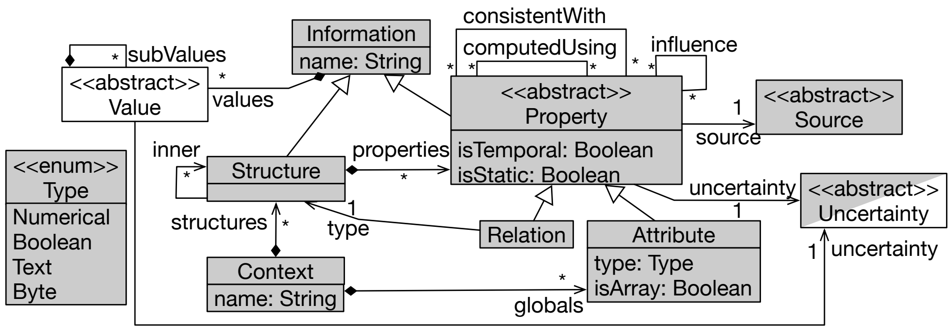


Figure 2.9: Excerpt of the context metamodel

materialized, *i.e.,* measured. Thanks to this metamodel representation, a developer can apprehend the possible causes behind malicious behavior by navigating from the context values to the decisions that have impacted its value (*Property.expected.impact*) and the goals it was trying to reach (*Decision.goals*). In Section *TODO*: **add reference** , we present an example of interactive diagnosis queries applied to the smart grid use case.

Context metamodel

Context models structure context information acquired at runtime. For example, in a smart-grid system, the context model would contain information about smart-grid users (address, names, etc.) resource consumption, etc.

An excerpt of the context model is depicted in Figure 2.9. we propose to represent the context as a set of structures (*Context.structures*) and global attributes (*Context.globals*). A structure can be viewed as a C-structure with a set of properties (*Property*): attributes (*Attribute*) or relationships (*Relation*). A structure may contain other nested structures (*Structure.inner*). Structures and properties have values. They correspond to the nodes described in the formalization section (*cf.* Section 2.2.2). The connection feature described in Section 2.1.2 is represented thanks to three recursive relationships on the Property class: *consistentWith*, *computedUsing* and *influence*. Additionally, each property has a source (*Source*) and an uncertainty (*Uncertainty*). It is up to the stakeholder to extend data with the appropriate source: measured, computed, provided by a user, or by another system (*e.g.,* weather information coming

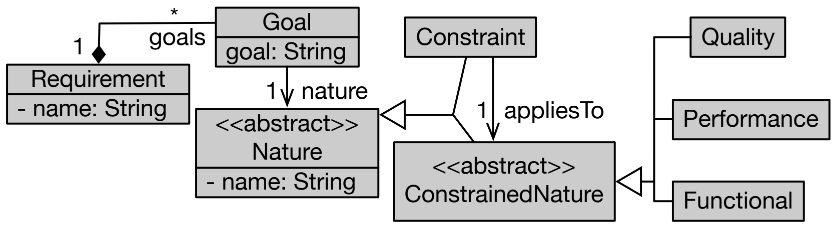


Figure 2.10: Requirement metamodel

from a public API). Similarly, the uncertainty class can be extended to represent the different kinds of uncertainties. Finally, a property can be either historic or static.

Requirement metamodel

As different solutions to model system requirements exist (*e.g.,* KAOS [**dardenne1993goal**], i* [**yu2011modelling**] or Tropos [**DBLP:journals/aamas/BrescianiPGGM04**]), in this metamodel, we abstract their shared concepts. The requirement model, depicted in Figure 2.10, represents the *requirement* as a set of *goals*. Each goal has a *nature* and a textual specification. The nature of the goals adheres to the four categories of requirements presented in Section 2.1.2. One may use one of the existing requirements modeling languages (*e.g.,* RELAX) to define the semantics of the requirements. Since the requirement model is composed solely of design elements, we may rely on static analysis techniques to infer the requirement model from existing specifications. The work of Egyed [**egyed01**] is one solution among others. This work is out of the scope of the paper and envisaged for future work.

In the guidance example, the requirement model may contain a **balanced resource distribution** requirement. It can be split into different goals: (i) *minimizing overloads*, (ii) *minimizing production lack*, (iii) *minimizing production loss.*

Action metamodel

Similar to the requirements metamodel, the actions metamodel also abstracts main concepts shared among existing solutions to describe adaptation processes and how they are linked to the context. Figure 2.11 depicts an excerpt of the action metamodel. we define a strategy as a set of tactics (*Strategy*). A tactic contains a set of actions (*Action*).

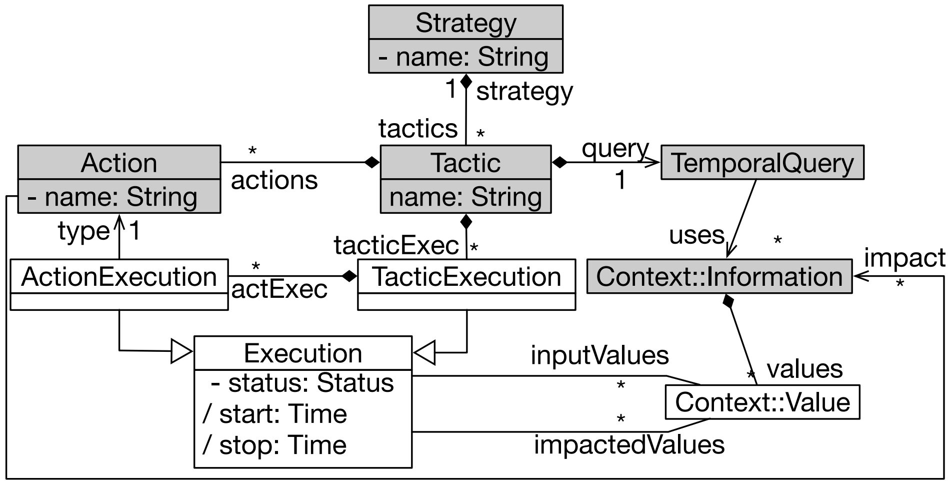


Figure 2.11: Excerpt of the action metamodel

A tactic is executed under a precondition represented as a temporal query (*Temporal-Query*) and uses different data from the context as input. In future work, we will investigate the use of preconditions to schedule the executions order of the actions, similarly to existing formalisms such as Stitch [**DBLP:journals/jss/ChengG12**]. The query can be as complex as needed and can navigate through the whole knowledge model. Actions have impacts on certain properties, represented by the *impacted* reference.

The different executions are represented thanks to the *Execution* class. Each execution has a status to track its progress and links to the impacted context values(*Execution.impactedValues*). Similarly, input values are represented thanks to the *Execution.inputValues* relationship. An execution has *start* and *end* time. Not to confuse with the *startTime* and *endTime* of the validity relation V^T. Whilst the former corresponds to the time range in which a value is valid, the *start* and *stop* time in the class execution correspond to the time range in which an action or a tactic was being executed. The start and stop attributes correspond to the relationL E_{A_E} (see Section 2.2.2). These values can be derived based on the validity relation. They correspond to the time range in which the status of the execution is "*RUNNING*". Formally, for every execution node e, $E_{A_E}(e) = (V(e) \mid e.status = \text{“RUNNING”})$.

Similarly to requirement models, it is possible to automatically infer design elements of action models by statically analyzing actions specification. Since acquiring information about tactics and actions executions happens at runtime, one way to achieve this is by intercepting calls to actions executions and updating the appropriate action model elements accordingly. This is out of the scope of this paper and planned for future work.

Validation

To validate and evaluate our approach, we implemented a prototype publicly available online[5]. This implementation leverages the GreyCat framework[6], more precisely the modeling plugin, which allows designing a metamodel using a textual syntax. Based on this specification, GreyCat generates a Java and a JavaScript API to create and manipulate models that conform to the predefined metamodel. The GreyCat framework handles time as a built-in concept. Additionally, it has native support of a lazy loading mechanism and an advanced garbage collection. This is achieved by dynamically loading and unloading model elements from the main memory when necessary.

The validation of our approach has been driven by the two research questions formulated in the introduction section:

- How to diagnose the self-adaptation process?
- How to enable reasoning over unfinished actions and their expected effects?

To address the first one, we describe how one can use our approach to represent the knowledge of an adaptation process for a smart grid system. Then, we present a code to extract the circumstances and the goals of a decision. For the second one, we present a scenario where a developer can use our approach to reason over unfinished actions and their expected effects. The presented code shows how information can be extracted from our model to enable any reasoning algorithm. Finally, we present a performance evaluation to show the scalability of our approach.

[5]https://github.com/lmouline/LDAS
[6]https://github.com/datathings/greycat

Diagnostic: implementation of the use case

In what follows, we explain how a stakeholder, Morgan, can apply our approach to a smart grid system in order to, first, abstract adaptive system concepts, then, structure runtime data, and finally, query the model for diagnosis purpose. The corresponding object model is depicted in Figure 2.12. Due to space limitation, we only present an excerpt of the knowledge model. An elaborate version is accessible in the tool repository.

Abstracting the adaptive system At design time (t_d), either manually or using an automatic process, Morgan abstracts the different tactics and actions available in the adaptation process. Among the different tactics that Morgan would like to model is "*reduce amps limit*". It is composed of three actions: sending a request to the smart meter (*askReduce*), checking if the new limit corresponds to the desired one (*checkNewLimit*), and notifying the user by e-mail (*notifyUser*). Morgan assumes that the *askReduce* action impacts consumption data (*csmpt*). This tactic is triggered upon a query (*tempQ*) that uses meter (*mt*), consumption (*csmpt*) and customer (*cust*) data. The query implements the "*no overload*" goal: the system shall never have a cable overload. Figure 2.12 depicts a flattened version of the temporal model representing these elements. The tag at upper-left corner of every object illustrates the creation timestamp. All the elements created at this stage are tagged with t_d.

Adding runtime information The adaptation process checks if the current system state fulfills the requirements by analyzing the context. To perform this, it executes the different temporal queries, including *tempQ*. For some reasons, the tempQ reveals that the current context does not respect the "*no overload*" goal. To adapt the smart grid system, the adaptation process decides to start the execution of the previously described tactic (*exec1*) at t_s. As a result, a decision element is added to the model along with a relationship to the unsatisfied goal. In addition, this decision entails the planning of a tactic execution, manifested in the creation of the element *exec1* and its subsequent actions (*notifyU*, *checkLmt*, and *askRed*). At t_s, all the actions execution have an IDLE status and an expected start time. All the elements created at this stage are tagged with the t_s timestamp in Figure 2.12.

At t_{s+1}, the planned tactic starts being executed by running the action *askReduce*.

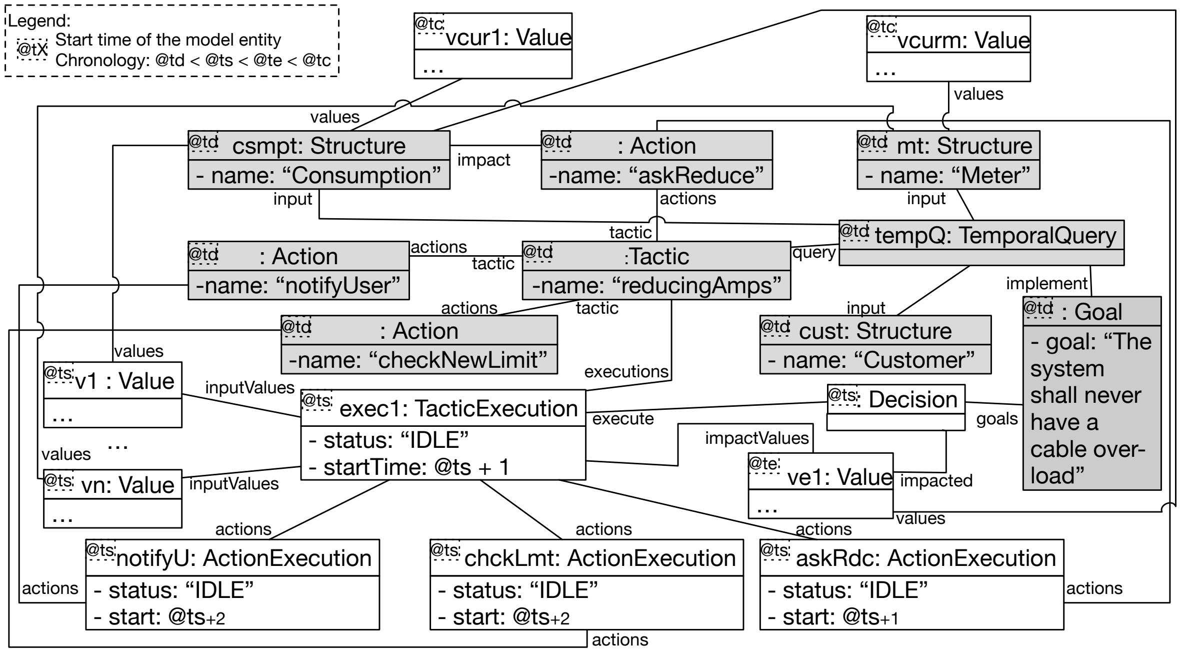


Figure 2.12: Excerpt of the knowledge object model related to our smart grid example

The status of this action turns from *IDLE* to *RUNNING*. Later, at t_{s+2}, the execution of *askReduce* finishes with a *SUCCEED* status and triggers the execution of the actions *notifyUser* and *checkNewLimit* in parallel. The status of *askReduce* changes to *SUCCEED* while the status of *notifyUser* and *checkNewLimit* turns to *RUNNING*. The first action successfully ends at t_{s+3} while the second ends at t_{s+4}. As all actions terminates with a *SUCCEED* status at t_{s+4}, accordingly, the final status of the tactic is set *SUCCEED* and the *stop* attribute value is set to t_e.

Interactive diagnosis query After receiving incident reports concerning regular power cuts, and based on the aforementioned knowledge model, Morgan would be able to query the system's states and investigate why such incidents have occurred. As described in Section **??**, she/he will interactively diagnose the system by interrogating the context, the decisions made, and their circumstances.

The first function, depicted in Listing 2.1, allows to navigate from the currently measured values (*vcur1*) to the decision(s) made. The for-loop and the if-condition are responsible for resolving the measured data for the past two days. Past elements are accessed using the *resolve* function that implements the $\mathcal{Z}^T$ relation (*cf.* Section **??**). After extracting the decisions leading to power cuts, Morgan carries on with the diagnosis by accessing the circumstances of this decision. The code to perform this task is depicted in Listing 2.1, the second function (getCircumstances). Note that the relationship *Decision.input* is the aggregation of *Decision.excecute.inputValues*.

```
// extracting the decisions
Decision[] impactedBy(Value v) {
  Decision[] respD
  for( Time t: v.modificationTimes() ):
    if (t >= v.startTime() - 2 day)
      Value resV = resolve(v,t)
    respD.addAll(from(resV).navigate(Value.impacted))
  return respD
}
// extracting the circumstances of the made decisions
Tuple<Value[], Goal[]> getCircumstance(Decision d) {
  Value[] resValues = from(d).navigate(Decision.input)
  Goal[] resGoals = from(d).navigate(Decision.goals)
  return Tuple<>(resValues, resGoals)
}
```

Listing 2.1: Get the goals used by the adaptation process from executed actions

Reasoning over unfinished actions and their expected effects

By associating the action model to the knowledge model, we aim at enhancing adaptation processes with new abilities to reason. In this section, we present an example of a reasoning algorithm which considers the impacts of running actions. This example is based on our use case (*cf.* Section 1.2).

Let's imagine that the adaptation process detects overloaded cables in the smart grid. To fix this situation, it takes several countermeasures, among which there are fuse state modifications. As detailed in Section 2.1.1, this action is considered as delayed action. Later, another incident is detected, for example, a substation is being overloaded. Before taking any actions, the adaptation process can, thanks to our solution, verify if the running actions will be sufficient to solve this new incident. If not, it can either take additional actions or replan the running one. The algorithm to reschedule current actions or to compute additional actions is out of the scope of this thesis. Here, we present the code to extract the required information from our model.

Checking if the running actions will be sufficient to solve all current issues can also be thought as: will the issue remain with the new context, *i.e.,* after each action have been executed. In our case, it is like verifying if the second overload will still remain with the new topology, which is coming. The adaptation process, therefore, needs to extract the context in the future. To do so, the adaptation process should know the latest timepoint at which the impact will be measured. Listing 2.2 shows the code to get this timepoint. Running, idle and finished actions are accessed thanks to the first two nested loops with the if-condition. We consider that failed and canceled actions have no effects. As finished actions may still have effects, we also consider them. Then we navigate through all impacted values to get their start time, *i.e.,* the beginning of their validity period (V^T relation, *cf.* Section 2.2.2). Doing so, we are sure to get the latest timepoint at which an impact will be measurable.

```
Time latestImpact(Knowledge k) {
  Time latestTime = CURRENT_TIME

  for(Decision d: from(k).navigate(decisions))
    for(TacticExecution te: from(d).navigate(execute))
      if(te.status == "RUNNING" || te.status == "IDLE" || te.status == "SUCCEED")
        for(Value v: from(te).navigate(impactedValues))
          if(v.startTime() > latestTime)
            latestTime = v.startTime()

  return latestTime
}
```

Listing 2.2: Get latest timepoint at which the impact will be measured

Using this timepoint, then the adaptation process can then compute how the grid should be after the actions have been executed. If the system has no prediction mechanism, then the adaptation process can verify how the power will be balanced over the new topology. Otherwise, it can use this prediction feature to compute the expected loads with the coming topology. Using this information, it can verify if all current incidents will be solved by the ongoing actions or not. If not, it may take additional actions or reschedule them.

Listing 2.3 depicts the code to extract all running actions. The nested loops allow accessing all executions made by decision. Then, we filter only those with the "RUNNING" status. The resulting collection should then be given to the scheduling algorithm, which will decide if rescheduling is possible and how.

```
TacticExecution[] runningActions(Knowledge k) {
  TacticExecution[] resA
  for(Decision d: k.decisions) {
    for(TacticExecution te: d.execute) {
      if(te.status == Status.RUNNING) {
        resA.add(te)
      }
    }
  }
  return resA
}
```

Listing 2.3: Extract ongoing actions and their effects

Using our model, developers have two solutions to model a rescheduling operation. Either they modify the actions, which may delete the history of the previous decision, or they mark all running and idle actions as "CANCELED" and create a new decision, with new actions, which update the circumstances and re-use the same requirements.

Performance evaluation

GreyCat stores temporal graph elements in several key/value maps. Thus, the complexity of accessing a graph element is linear and depends on the size of the graph. Note that in our experimentation we evaluate only the execution performance of diagnosis algorithms. For more information on I/O performance in GreyCat, please refer to the original work by Hartmann *et al.,* [**DBLP:conf/seke/0001FJRT17**; **DBLP:phd/basesearch/Hartmann16**].

```
MATCH (input)-[*4]->(output)
WHERE input.id IN [randomly generated set]
RETURN output
LIMIT O
```

Listing 2.4: Traversal used during the experimentations

We consider a diagnosis algorithm to be a graph navigation from a set of nodes (input) to another set of nodes (output). Unlike typical graph algorithms, diagnosis algorithms are simple graph traversals and do not involve complex computations at the node level. Hence, we believe that three parameters can impact their performance (memory and/or CPU): the global size of the graph, the size of the input, and the number of traversed elements. In our evaluation, we altered these parameters and report on the behavior of the main memory and the execution time. The code of our evaluation is publicly available online[7]. All experiments reporting on memory consumption were executed 20 times after one warm-up round. Whilst, execution time experiments were run 100 times after 20 warm-up rounds. The presented results correspond to the mean of all the iterations. We randomly generate graph with sizes (N) ranging from 1 000 to 2 000 000. At every execution iteration, we follow these steps: (1) in a graph with size N, we randomly select a set of I input nodes, (2) then traverse M nodes in the graph, (3) and we collect the first O nodes that are at four hops from the input element. Listing 2.4 describes the behavior of the traversal using Cypher, a well-known graph traversal language.

We executed our experimentation on a MacBook Pro with an Intel Core i7 processor (2.6 GHz, 4 cores, 16GB main memory (RAM), macOS High Sierra version 10.13.2). We used the Oracle JDK version 1.8.0_65.

How performance is influenced by the graph size N? This experimentation aims at showing the impact of the graph size (N) on memory and execution time while performing common diagnosis routines. We fix the size of I to 10. To assure that the behavior of our traversals is the same, we use a seed value to select the starting input elements. We stop the algorithm when we reach 10 elements. Results are depicted in Figure 2.13.

As we can notice, the graph size does not have a significant impact on the execution time of diagnosis algorithms. For graphs with up to 2,000,000 elements, execution time remains between 2 ms and four 4 ms. We can also notice that the memory consumption insignificantly increases. Thanks to the implementation of a lazy loading and a garbage

[7]https://bitbucket.org/ludovicpapers/icac18-eval

collection strategy by GreyCat, the graph size does not influence memory or execution time performance. The increase in memory consumption can be due to the internal indexes or stores that grow with the graph size.

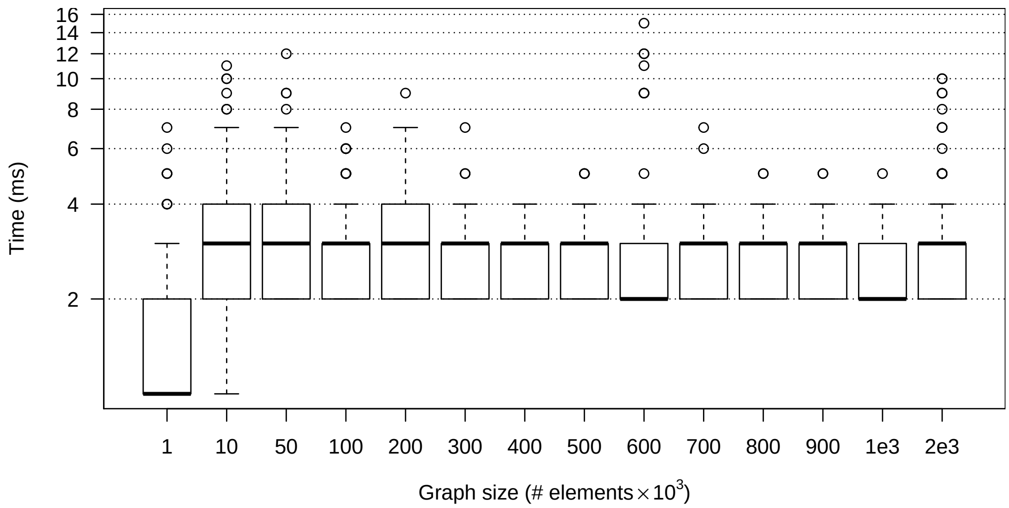
How performance is influenced by the input size (I)? The second experiment aims to show the impact of the input size (I) on the execution of diagnosis algorithms. We fix the size of N to 500 000 and we variate I from 1 000 nodes to 100 000, *i.e.,* from 0.2% to 20% of the graph size. The results are depicted in Figure 2.14 (straight lines).

Unlike to the previous experiment, we notice that the input size (I) impacts the performance, both in terms of memory consumption and execution time. This is because our framework keeps in memory all the traversed elements, namely the input elements. The increase in memory consumption follows a linear trend with regards to N. As it can be noticed, it reaches 2GB for I=100 000. The execution time also shows a similar curve, while the query response time takes around than around 60ms to run for I=1 000, it takes a bit more than 4 seconds to finish for I=100 000. Nonetheless, these results remain very acceptable for diagnosis purposes.

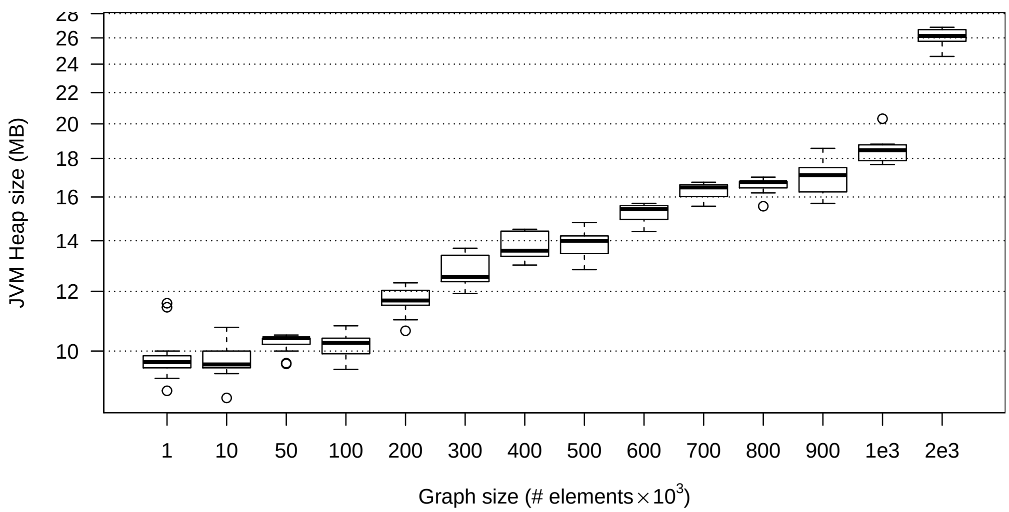
How performance is influenced by the number of traversed elements (M)? For the last experiment, we aim to highlight the impact of the number of traversed elements (M). For this, we fix I and O to 1, and randomly generate a graph with sizes ranging from 1 000 to 100 000. Our algorithm navigates the whole model (M=N). We depict the results in Figure 2.14 (dashed curve). As we can notice, the memory consumption increases in a quasi-linear way. The memory footprint to traverse M = 100 000 elements is around 0.9GB. The progress of the execution time curve behaves similarly, in a quasi-linear way. Finally, the execution time of a full traversal over the biggest graph takes less than 2.5 seconds.

Discussion

By linking context, actions, and requirements using decisions, data extraction for explanation or fault localization can be achieved by performing common temporal graph traversal operations. In the detailed example, we show how a stakeholder could use our approach to define the different elements required by such systems, to structure runtime

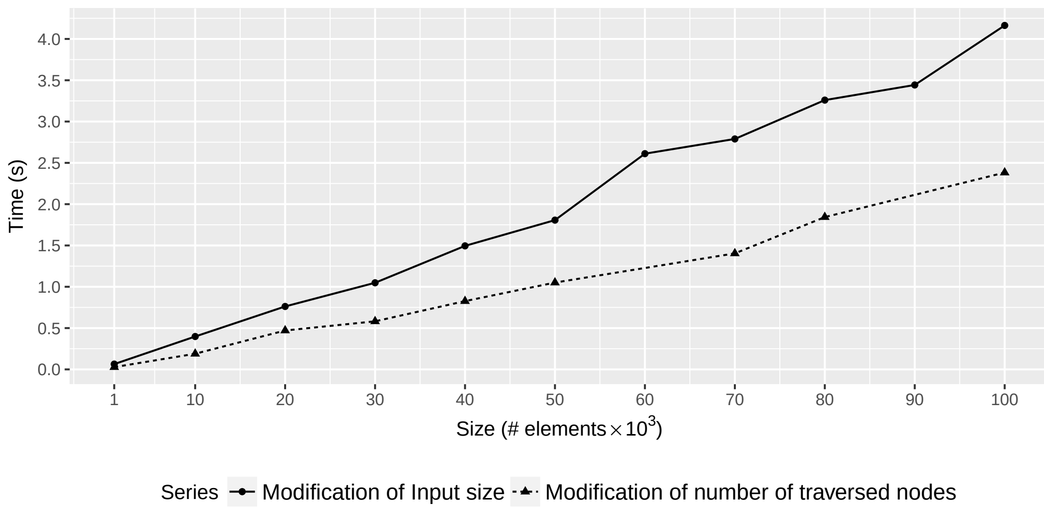


(a) Execution time evolution

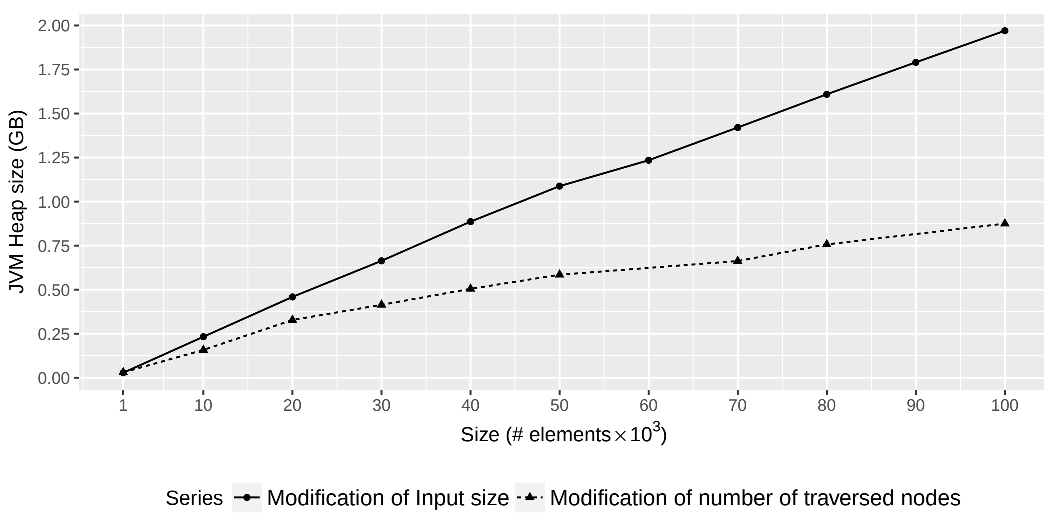


(b) Memory evolution

Figure 2.13: Experimentation results when the knowledge based size increases



(a) Evolution of the execution time



(b) Evolution of the memory consumption

Figure 2.14: Results of experiments when the number of traversed or input elements increases

data, finally, to diagnose the behavior of adaptation processes.

Our implementation allows to dynamically load and release nodes during the execution of a graph traversal. Using this feature, only the needed elements are kept in the main memory. Hence, we can perform interactive diagnosis routines on large graphs with an acceptable memory footprint. However, the performance of our solution, in terms of memory and execution time, is restricted by the number of traversed elements and the number of input elements. Indeed, as shown in our experimentation, both the execution time and the memory consumption grow linearly.

In the Luxembourg smart grid, a district contains approximatively 3 data concentrators and 227 meters[8]. Counting the global datacenter, the network is thus composed of 231 elements. Each meter sends the consumption value every 15 min, being 908 every hours. Plus, there is from 0 to 273 topology modifications in the network. In total, the system generates from 908 to 1,181 new values every hour. If we consider that we have one model element per smart grid entity and one model element per new value, 100,000 model elements correspond thus from $((100,000 - 231) * 1H)/1,181 = 84,5H$ ($\sim$ 3,5 days) to $((100,000 - 231) * 1H)/908 = 109,9H$ ($\sim$ 4,6 days) of data. In other word, our approach can efficiently interrogate up to $\sim$5 days history data in 2.4s of one district.

Threat to validity

Size of the model

Engineer effort to use the solution

Performance of the adaptation process

Conclusion

Adaptive systems are prone to faults given their evolving complexity. To enable interactive diagnosis over these systems, we proposed a temporal data model to abstract and store knowledge elements. We also provided a high-level API to specify and

[8] Previously, our studies uses the data described in [**DBLP:conf/smartgridcomm/0001FKTPTR14**], which corresponded to the all Luxembourg at this date. Since 2014, the smart grid has been more and more deployed. Numbers present in this paper now correspond more to one district.

perform diagnosis algorithms. Thanks to this structure, a stakeholder can abstract and store decisions made by the adaptation process and link them to their circumstances –targeted requirements and used context– as well as their impacts. In our evaluation, we showed that our solution can efficiently handle up to 100 000 elements, in a single machine. This size is comparable to 5 days history of one district in the Luxembourg smart grid.

Throughout this work, we assumed that designers are able to link actions with their expected impacts at design time. However, this is not always true. Some impacts cannot be known in advance. In this perspective, in addition to the future plans already mentioned throughout the paper, we will investigate techniques to identify unknown impacts on the context model, for instance, by studying the use of machine learning techniques. In order to improve the accuracy and correctness of diagnosis routines, another aspect to be considered for future work is handling uncertainty in self-adaptive systems. Understanding the effect of uncertainty on the development of self-adaptive systems and their diagnosis is still an open question. We plan to explore this research direction by answering the following questions: How to represent and express uncertainty in self-adaptive systems at design time? How to efficiently interrogate data with uncertainty in self-adaptive systems, for instance, for troubleshooting purpose?

Abbreviations

MAPE-k Monitor, Analyze, Plan, and Execute over knowledge. 27, *Glossary:* MAPE-k

Glossary

action “Process that, given the context and requirements as input, adjusts the system behavior”, IEEE Standards [**iso2017systems**]. 15

circumstance State of the knowledge when a decision has been taken. 15

context In this document, I use the definition provided by Anind K. Dey [**DBLP:journals/puc/Dey01** “Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and [the system], including the user and [the system] themselves”. 15, 20, 27

decision A set of actions taken after comparing the state of the knowledge with the requirement. 15, 27

knowledge The knowledge of an adaptive system gathers information about the context, actions and requirements. 15–17, 26, 27

MAPE-k A theoretical model of the adaptation process proposed by Kephart and Chess [**DBLP:journals/computer/KephartC03**]. It divides the process in four stages: monitoring, analysing, planning and executing. These four stages share a knowledge. 27, *Abbreviation:* MAPE-k

metamodel Through this thesis, I use the definition of Seidewitz: “A metamodel is a specification model for a class of [system under study] where each [system under study] in the class is it-self a valid model expressed in a certain modeling language.” [**DBLP:journals/software/Seidewitz03**] . 26, 27

requirement "(1) Statement that translates or expresses a need and its associated constraints and conditions, (2) Condition or capability that must be met or possessed by a system [...] to satisfy an agreement, standard, specification, or other formally imposed documents", IEEE Standards [**iso2017systems**]. 15, 27