

Reflecting on the past and the present with temporal graph-based models

Antonio García-Domínguez
SEA, SARI
Aston University, UK
a.garcia-dominguez@aston.ac.uk

Nelly Bencomo
SEA, SARI
Aston University, UK
nelly@acm.org

Luis H. Garcia Paucar
SEA, SARI
Aston University, UK
garciapl@aston.ac.uk

ABSTRACT

Self-adaptive systems (SAS) need to reflect on the current environment conditions, their past and current behaviour to support decision making. Decisions may have different effects depending on the context. On the one hand, some adaptations may have run into difficulties. On the other hand, users or operators may want to know why the system evolved in a certain direction. Users may just want to know why the system is showing a given behaviour or has made a decision as the behaviour may be surprising or not expected. We argue that answering emerging questions related to situations like these requires storing execution trace models in a way that allows for travelling back and forth in time, qualifying the decision making against available evidence. In this paper, we propose temporal graph databases as a useful representation for trace models to support self-explanation, interactive diagnosis or forensic analysis. We define a generic meta-model for structuring execution traces of SAS, and show how a sequence of traces can be turned into a temporal graph model. We present a first version of a query language for these temporal graphs through a case study, and outline the potential applications for forensic analysis (after the system has finished in a potentially abnormal way), self-explanation, and interactive diagnosis at runtime.

CCS CONCEPTS

• **Software and its engineering** → **Software system models; Extra-functional properties; Designing software;** • **Computing methodologies;**

KEYWORDS

Self-explanation, Temporal Graph Models, Runtime models, Self-adaptation

1 INTRODUCTION

In [31], it is argued that self-explanation shown by the running system helps someone diagnosing the behaviour of the system to analyze and trace past actions, helping fix potential faults and fostering the trust of the end users. To enable these capabilities, we argue that self-adaptive systems should be equipped with traceability management facilities and offer temporal links to provide (i) the impacts of the adaptation actions over the quality properties of the system over time and (ii) the history of the decisions of the system and the evidence that supports the decisions made with the environmental conditions observed.

In this paper we offer the first contributions towards allowing the system to support explanations to operators and end users based on a generic meta-model. Specifically, we define a generic meta-model for structuring execution traces of SAS, and show how a sequence of traces can be turned into a temporal graph model. We present a first version of a query language for these temporal graphs based on specific cases related to a case study. Our solution relies on temporal model-based graphs that abstracts decisions, evidence collected and their corresponding estimated impacts on quality properties of the system. We foresee two potential applications of our approach: forensic analysis of SAS once the system has finished, and self-explanation supported by the self-adaptive system at runtime.

The paper is organised as follows. Section 2 presents the basic concepts in self-explanation and temporal graphs needed to understand the rest of the text. Section 3 describes our proposed approach for creating frameworks for reusable self-explanation, and outlines our proof of concept implementations of the key components. Section 4 presents a case study on an existing self-adaptive system, together with a number of time-aware queries targeted at users and developers. Section 5 relates this work to others in the fields of self-explanation and model versioning. Section 6 concludes the paper with some general remarks and our lines of future work.

2 BACKGROUND

This section will present some of the basic concepts that underlie our proposal: the need for self-explanation in self-adaptive system, and our specific choice among the various definitions available in the literature for temporal graphs.

2.1 Self-explanation and diagnosis in self-adaptive systems

Our increasing reliance on software systems has made self-adaptation a expected capability. However, self-adaptation actions may run into problems or unexpected behaviour due to uncertainty in the environment [11]. Therefore, end users may require explanation about the reasons the system is showing the current behaviour and specifically why it has made particular adaptations actions that were not expected. Further, in case of a failure, the operators may perform diagnosis during runtime, or forensic studies after the system has terminated, to therefore identify the origins of the failure. Surprisingly, this area of research has been rather limited with scarce research efforts. We describe some of the few initiatives below.

Early work has been done by Roth-Berghofer et al [7, 28] on Explanation-aware Computing. The main idea was to help designers and engineers to create explanations for users. The explanations

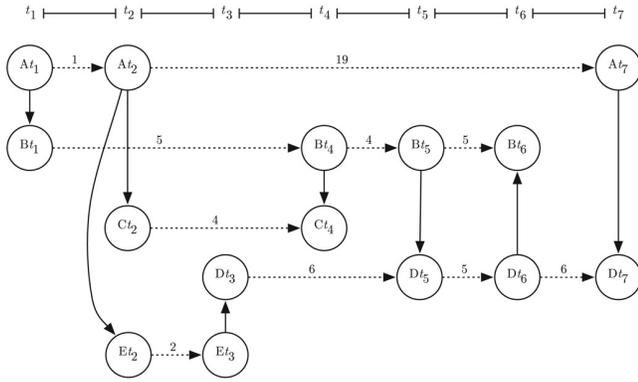


Figure 1: Email exchange example of a temporal graph by Kostakos [21]

should cover why specific services were recommended and how the system infers that the end user will agree, and therefore maintain the end user satisfied by the recommendations. In their work explanation generation was an aim.

More recently, the need of self-explanation in self-adaptive systems was argued in [3, 29, 31]. The authors claim the behaviour of self-adaptive systems is emergent, and means that the behaviour exposed by the running system may be seen as unexpected by its end users or its developers. They further argue that trust in the system and the resolution of the surprising behaviour can only be achieved if a self-adaptive system is also capable of self-explaining itself [29]. In [10] we presented how traceability (i.e. following the life of a requirement) and versioning (i.e. keeping track of how a specific artifact evolved over time) are needed for self-explanation and diagnosis. More recently, in [23], the authors present a temporal model to support interactive diagnosis of adaptive Systems. The authors describe a temporal data model to represent, store and query decisions as well as their relationship with the context, requirements, and adaptation actions. Self-explanation and diagnosis support is still a young research area that needs more research efforts.

2.2 Temporal graphs

A graph is a well-understood concept in computer science: in its most basic form, it is a collection of nodes with edges connecting them, which may be directed or undirected. There is a number of ways to extend the concept of a graph with the time dimension: in this section we will present three, one of which is the base for our proposal.

Kostakos [21] was one of the first to use the term *temporal graph*, as a graph encoding of a temporal dataset of events. Kostakos’ proposal includes an example where the email exchanges between a number of people through time are transformed into a temporal graph like the one in Figure 1 in three steps:

- (1) One node is created per person and point in time when it sent an email: a person *A* would have nodes At_1 , At_2 and so on.

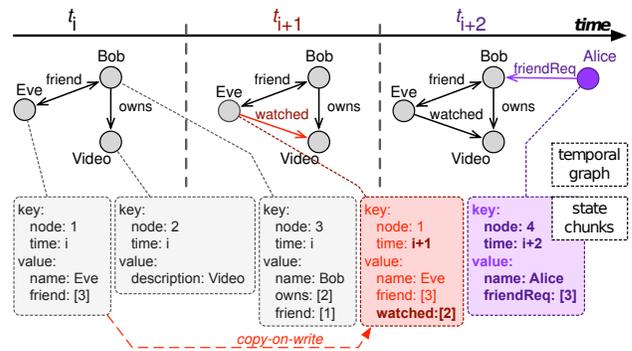


Figure 2: Example of a time-evolving temporal graph, by Hartmann et al. [13]

- (2) Directed edges are used to link the various nodes of a person into a sequence. The edges have weights equal to the time elapsed between the two timepoints.
- (3) Unweighted directed edges are used to link people who exchanged emails at a specific timepoint: for instance, an edge from At_3 to Bt_3 means that at timepoint 3, *A* sent an email to *B*.

This representation lends itself well to variants of traditional graph metrics, such as temporal distance between two people, or the temporal availability of a path from one person to another (i.e. whether there is a chain of emails from one person to another while considering time ordering). It uses discrete time, where timepoints act as timeslices and events are assumed to be instantaneous.

In a later survey of temporal networks by Holme and Sarämäki [15], this type of graph with instantaneous edge activations is called a *contact sequence*, and another type of temporal network is identified: *interval graphs*, where edges are active over a set of time intervals rather than at specific timepoints. Holme and Sarämäki mentioned in the same survey how the use of temporal networks was becoming common across multiple disciplines, and no standardized notation had been set out yet.

Regardless, these two previous works consider temporal graphs to be rearrangements of a sequence of events between persistent entities, which may or may not be instantaneous. In contrast, Hartmann et al. consider temporal graphs as attributed labelled graphs¹ whose state evolves over time [13]. In the most naive approach, one would think of simply storing each version of a time-evolving graph as separate snapshots, and to visit each snapshot as needed. Unfortunately, the space requirements for such a naive solution would skyrocket as we increase the number of timepoints, and the time needed to visit the various versions would raise as well. In the same paper, Hartmann et al. specifically considered Internet of Things devices and cyber-physical systems, where a network of sensors may be picking up readings frequently over a long period of time, at different rates.

Hartmann et al. proposed a more efficient data model and storage mechanism for these temporal graphs, and made it available as the

¹Attributed graphs have key-value pairs in their nodes and edges, and labelled graphs classify nodes and edges into equivalence sets, e.g. “person” nodes and “emailed” edges. Neo4j is a well-known implementation of this data structure.

Greycat open source project². In this data model (shown in Figure 2), the graph is stored as a collection of nodes, which are *conceptual identifiers* that are mapped to specific *state chunks* depending on the *world* and *timepoint* chosen to visit it. Nodes have a lifespan between two specific timepoints, and within that lifespan they may take on a sequence of state chunks. Each state chunk appears at a specific timepoint and overrides any previous state chunk.

In the example in Figure 2, during timepoint $i + 1$ a “watched” edge is created from “Eve” to “Video”, and in $i + 2$ “Alice” enters the graph and posts a “friendReq” to “Bob”. Instead of storing the three full graphs outright, we only create new state chunks for “Eve” and “Alice” as needed, using a *copy-on-write* style. State chunks are keyed by node, time and (in Greycat) by *world*. This third coordinate makes it possible to “fork” the graph into multiple branching paths, which enables what-if analyses.

The approach presented in this paper adopts the data model by Hartmann et al. of an evolving labelled attributed graph. If we can turn the models that the system operates upon into this type of temporal graphs, we could allow the system to reflect on what it has been doing in the long and the short term, and provide clear explanations about its history to the user.

3 PROPOSED APPROACH

Our end goal is to develop a generic and reusable framework to allow self-adaptive systems using the `models@run.time` approach to reflect upon their past execution and to improve the explanations provided to the users about their behaviour. In this section we will describe the various components that we see as necessary to achieve this goal. The next section will present an initial case study for a SAS which must choose between multiple configurations.

3.1 Problem-independent execution trace models

Self-adaptive systems are generally built as feedback loops (e.g. those following the MAPE-K architecture [18]). At each timepoint or *time slice*, observations are made and analysed, then future behaviour is planned, and those plans are executed. Since we want to make the queries on the execution history reusable, the history must be expressed in a language that can be reused across multiple problems (e.g. network management and smart grids). Whether the language could be further reused across multiple types of self-adaptive systems would require further research. It may be necessary to allow extending these metamodels to accommodate algorithm-specific details.

As an example of a potential execution trace metamodel for self-adaptive systems that need to switch between multiple configurations, consider the metamodel shown in Figure 3. At the top level, the LOG for a time slice records the requested non-functional requirements as NFR objects (which have specific satisfaction thresholds between 0 and 1), together with the METRICS to be measured to check their satisfaction, and the alternative ACTIONS that can be taken. These are used in the various DECISIONS that must be taken by the system. The system is pre-configured with a REWARDTABLE linking the satisfaction of certain NFR with certain ACTIONS to a reward value. These rewards may evolve over time.

²<https://github.com/datathings/greycat>

Each DECISION is based on an OBSERVATION of the environment, which produces a set of MEASUREMENTS of the METRICS. In this version of the language we do not include specific values, but rather between which of the various THRESHOLDS the value was. For instance, if we had three thresholds x and y with values 10 and 20, position 0 would be for $x < 10$, position 1 would be $10 \leq x < 20$, and position 2 would be $x \geq 20$. Using these measurements, the system would derive a set of beliefs about the satisfaction of the NFR and the value of the different ACTIONS, and finally pick a specific ACTION from the DECISION.

3.2 Transparent temporal graph storage

The next part of the approach is to store the models themselves in a temporal graph to facilitate querying. In the literature, there are essentially two approaches to integrate graph databases with modelling technologies: changing the storage layer of the system directly (as implemented by NeoEMF [6]), or having an external system watch the existing storage and update the graph when changes are detected (as done by our Hawk [9] tool).

Either option is valid, but we chose Hawk as it had a number of advantages over NeoEMF for this problem. First, using Hawk does not require modifying existing systems: Hawk has a component-based architecture, making it possible to change the database technology, graph updating algorithms and supported model storage locations to fit the situation. In addition, Hawk has been specifically designed to detect the parts of a model that have changed, and only changes the subgraph that is impacted by these changes [1].

In our implementation, we have extended Hawk with the ability to use Greycat as a backend. We have also extended Hawk with a time-aware version of the incremental graph update algorithm, which tells time-aware backends (i.e. Greycat) to “travel in time” to the timepoint when the change has been introduced before applying the detected changes. This allows for the preservation of the original graph at the previous timepoint, making it possible to travel back and forth in time to answer queries about the history of the trace execution model.

3.3 Reusable time-aware query language

Having a convenient way to write queries over the history of the graph is another important ingredient for reusable self-explanation. To simplify adoption, the most direct approach is to start from an existing model querying language (e.g. OCL), and then add the ability to traverse the history of a model element or a type through their lifespans. Our definitions for the history of a model element and a type are as follows:

- The history of a model element starts from the moment it is created, and ends when it is destroyed. Model elements are assumed to have a unique identity, which could be a natural or artificial identifier or its location within the model. There will be a new version of a model element every time its state changes, whether by changing the value of an attribute or the target of one of its references to other model elements.
- Model element types are considered “immortal”, in the sense that they are created at the first timepoint in the graph and last to the virtual “end of time” of the graph. We will have a

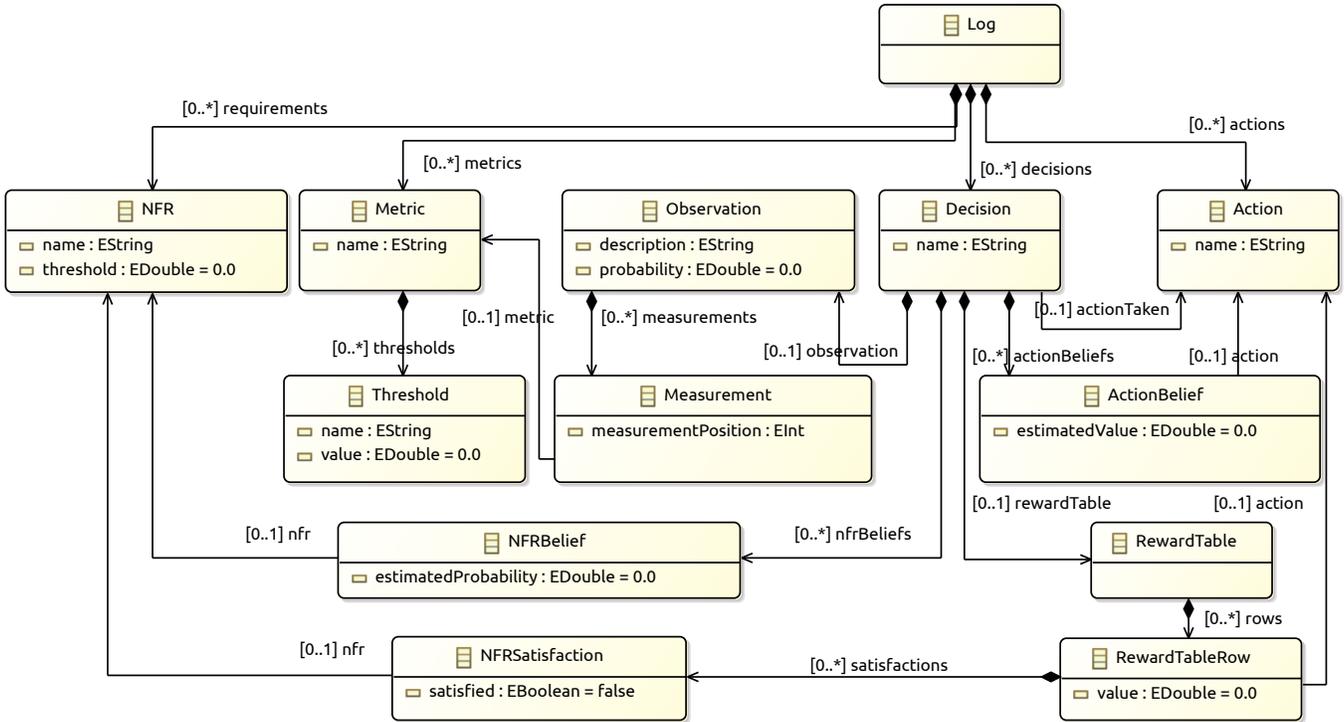


Figure 3: Execution trace metamodel for a decision-based self-adaptive system

new version of a model element type every time an instance of the type is created or destroyed.

For model elements and model element types, we consider these to be the basic time-aware operations that must be supported by the temporal graph backend (e.g. Greycat): i) retrieving all versions, ii) all versions within a range, iii) versions from/up to a certain timepoint (included), iv) earliest/previous/next/latest version, and v) retrieving the timepoint for that version.

Our approach is heavily inspired by the “history” fields proposed by Rose and Segev during their work on temporal object-oriented data models in the 90s [27]. Rose and Segev went further in their proposal, suggesting the availability of per-field histories and a wider variety of predicates covering linear temporal logic. We are considering providing pre-defined versions of these additional facilities on top of the basic primitives above.

For our proof-of-concept implementation, Hawk already had most of the elements, as it came with a number of backend-specific and backend-agnostic query engine components. The most mature query language at the time of writing is a dialect of the Epsilon Object Language (EOL) [20], essentially a mix between JavaScript and OCL. It was a matter of defining a new query engine based on the EOL one with additional support for the previous primitives: Table 1 lists the syntax for these new primitives.

3.4 Reusable visualizations

The last piece in the puzzle would be to have a reusable set of visualizations for a certain class of self-adaptive system. These could be dashboards with the key instants in the self-adaptive

Operation	Syntax
All versions, from newest to oldest	x.versions
Versions within a range	x.getVersionsBetween(from, to)
Versions from a time-point (included)	x.getVersionsFrom(from)
Versions up to a time-point (included)	x.getVersionsUpTo(from)
Earliest / latest version	x.earliest, x.latest
Next / previous version	x.next, x.prev/x.previous
Version timepoint	x.time

Table 1: Implemented time-aware primitives in the Hawk EOL dialect, for a model element or type x

system, to allow users to jump to the main changes in behaviour that were introduced automatically, or a predefined sequence of “why”-form questions for common queries: why was it doing this at that time, why did it stop doing the previous action, why did it reason that was beneficial, why was the reasoning process in such a state, and why was the user configuration like that.

Adding these visualizations to a system should require less effort once we have achieved the definition of a standardized trace metamodel, and have reusable temporal storage and querying capabilities that we can always start from. The visualizations would be backed by time-aware queries, and could be packaged together

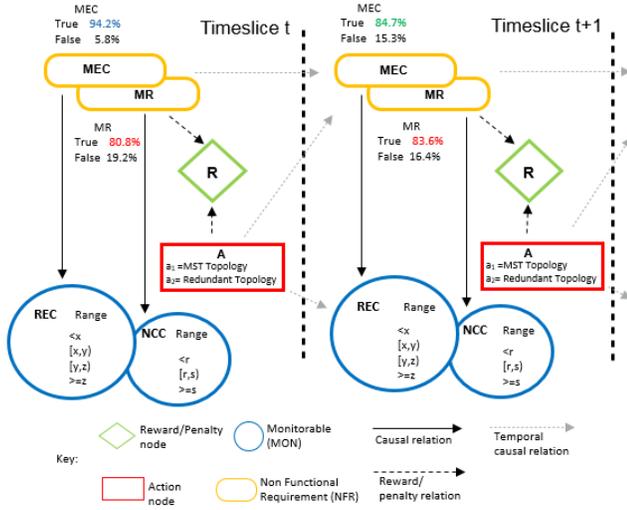


Figure 4: RDM Case Study

with the configuration of the self-adaptive system for a specific problem domain.

Beyond getting the data, another challenge is finding an accessible way to present it, which steps into the realm of human-computer interaction and is outside the scope of this paper. Regardless, at this stage the reusable visualizations remain as a future line of work.

4 CASE STUDY: DECISION-MAKING SAS FOR NETWORKS

As an example to demonstrate the feasibility of the ideas proposed, let us consider the case of the Remote Data Mirroring (RDM) self-adaptive system (SAS). RDM is a technique to protect data against inaccessibility to therefore provide further resistance to data loss [17, 26]. An RDM maintains data availability and prevents data loss by storing copies (i.e., replicates) on servers (i.e., data mirrors) in physically remote locations [8].

Fig. 4 presents the R-POMDP (Relational Partially Observable Markov Decision Process [22]) model of the RDM SAS for a given IT network infrastructure which has been used as the case studies in [2, 25].

The RDM described above has been designed to be configured by using two different topologies: minimum spanning tree (MST) and redundant topology (RT). These two possible configurations allow the RDM selectively activate and deactivate network links to change its overall topology at runtime [8].

The RDM SAS self-adapts reconfiguring itself at runtime according to the changes in its environment, which may include either delayed or dropped messages and network link failures. Each network link in the RDM brings upon an operational cost and has a measurable throughput, latency, and loss rate. The performance and reliability of the RDM are determined by these metrics according to the following trade-off: while RT is more reliable than MST, RT can be prohibitively more expensive than MST in some contexts. Each configuration provides its own levels of reliability and energy costs which are taken into account while estimating the levels of

Listing 1: Excerpt of the original JSON trace execution logs from the Remote Data Mirroring self-adaptive system

```

1 {
2   "0": {
3     "current_belief_mec_true": 0.5,
4     "current_belief_mr_true": 0.25,
5     "current_observation_code": -1,
6     "current_rewards": [
7       [90.0, 45.0, 25.0, 5.0],
8       [100.0, 10.0, 20.0, 0.0]
9     ],
10    "ev.mst": 465.104345236406,
11    "ev.rt": 326.710194366562,
12    "flagUpdatedRewards": 0,
13    "observation_description": "None",
14    "observation_probability": 0.0,
15    "selected_action": "MST"
16  },
17  "1": {
18    "current_belief_mec_true": 0.94, ...
19  }, ...
20 }
    
```

satisfaction of the NFRs observed, the Maximization of Reliability (MR) and the Minimization of Energy Consumption (MEC). As such, and RDM makes decisions about the topologies to use. The operators may find themselves asking the reasons why the RDM SAS has used one topology instead of the other.

The states of these NFRs are not directly observable. Observations about their states are obtained by using monitoring variables (called MON variables). Two MON variables REC="Ranges of Energy Consumption" and NCC="Number of Concurrent Connections" has been specified. In [24], we have shown the requirements specification based on Partially Observable Markov Decision Processes (POMDP) that enables reasoning and decision-making about partial satisfaction of non-functional requirements (NFRs) and their trade-off based on evidence collected at runtime based on the formalism for decision-making under uncertainty provided by POMDPs (See Fig. 4).

4.1 Log preprocessing

In its current implementation, the RDM SAS produces execution traces in JSON format for each time slice, mentioning the observations made, the currently estimated levels of satisfaction of the NFRs, and the preferences currently being applied in the decision process. The JSON log is made available for forensic purposes to debug the system after the fact. Listing 1 shows an excerpt of the log for the first time slice.

Due to time constraints, for this first feasibility study it was decided to collect a large number of JSON logs and transform them into a temporal graph, answering queries away from the system (in an "off-line" fashion). It is planned to revise the RDM SAS in future studies to have it maintain the temporal graph while it is running, so queries can be answered "on-line" for reflection and

self-explanation. All the resources used in this study are available online³, and Hawk is freely available as open source from Github⁴.

The transformation of the logs into a temporal graph was done on a Lenovo Thinkpad X1 Carbon laptop with an Intel i7-6600U CPU running at 2.60GHz, running Ubuntu Desktop 18.04, Linux 4.15.0 and Oracle Java 8u102, allocating 8GB of RAM through `-Xmx8g -Xms8g`. The process required a number of steps:

- (1) The RDM SAS had been previously run in a different machine through a simulation over 1000 time slices, producing a sequence of entries in JSON format which took 536KB.
- (2) The trace execution metamodels of Section 3.1 were implemented in the Eclipse Modelling Framework [30].
- (3) A small Java program (381 lines of code) was created to transform the JSON logs into EMF models conforming to the trace execution metamodels, and store them into a Subversion⁵ (SVN) repository as a sequence of revisions of the same trace execution model file. The SVN repository was produced after 48 seconds, taking up 7.3MB of disk space, and resulted in 888 commits. SVN naturally ignores cases when the model has not changed at all from one time slice to the next.
- (4) Hawk was instructed to index the full history of the SVN repository into a Greycat temporal graph, using its new time-aware updater component. From the second revision onwards, Hawk used its incremental updating capabilities to propagate any differences observed since the previous revision. The Greycat temporal graph over the 888 commits was produced after 21 seconds, taking up 31MB of disk space⁶.

We chose SVN over manually time-stamped files (for example, `slice1.xmi`, `slice2.xmi`, and so on) because the version of the local folder indexing component in Hawk at the time would index all time-stamped files separately, rather than as a single evolving model. The SVN component in Hawk is designed to provide the full history of each file, which produced the results we intended. In the future, we may create a version of the local folder indexing component that understands that files time-stamped according to a certain convention are versions of the same model.

Regardless, ignoring those 112 timepoints when the model did not change did not result in loss of information. Indeed, if we only have SVN revisions for timepoints 1 and 10, that means that the model did not change at all between timepoints 2 and 9. If we ask for the state of the model at timepoint 5, we will see the version at timepoint 1 as we should. Omitting timepoints which did not introduce any changes can result in significant space savings when changes are infrequent, i.e. in a stable system configuration. This also reduces the number of results that we will have to go through in our queries. It can be thought of as a form of compression.

4.2 Time-aware queries for developers

We argue that self-explanation needs to be tailored to the reader. SAS developers and integrators will be interested on a different type of explanations about the system. Particularly, they will often need to verify that certain desirable properties are being met, while

³<https://git.aston.ac.uk/garcia-a/hawk-mrt2018>

⁴<https://github.com/mondo-project/mondo-hawk>

⁵<https://subversion.apache.org/>

⁶A previous version with second-level rather than millisecond-level timestamps required 2.6MB instead. We intend to investigate this further in future studies.

Listing 2: EOL query to check the evolution of belief levels through the lifespan of each action choice

```
1 return RewardTableRow.latest.all
2   .collect(r_row | r_row.versions.size).max();
```

Listing 3: EOL query to check the min/max/average shift in reward values through the life of the SAS

```
1 var rewardShifts = RewardTableRow.latest.all
2   .collect(row | row.getRewardShifts()).flatten();
3
4 return Sequence { rewardShifts.min(),
5   rewardShifts.max(), rewardShifts.average() };
6
7 operation RewardTableRow getRewardShifts(): Sequence {
8   var v = self.versions;
9   if (v.size <= 1) {
10    return Sequence {};
11  } else {
12    return v.subList(0, v.size - 1)
13      .collect(v | v.value - v.prev.value);
14  }
15 }
16 operation Sequence average() { return self.sum() / self.size(); }
```

other times they will want to identify points in time where the self-adaptive system misbehaved.

Listing 2 shows a first example of what can be done for the developers. It allows the RDM SAS developer to check if the internal reward values in the decision algorithm have evolved over time or if they have remained the same. It operates as follows:

- (1) `RewardTableRow.latest` returns the latest version of the `REWARDTABLEROW` type node in the temporal graph. There are only two versions for this node: the one at the beginning of time with no instances, and the second one with all the instances. `REWARDTABLEROWS` are not created or deleted, they are simply modified with different reward values.
- (2) `.all` returns all instances of that type at that point in time.
- (3) `.collect(x | expression)` visits each instance, computing an expression and collecting the results into a new list.
- (4) `r_row.versions.size` returns the number of versions for that instance. This would be the number of times that the reward values have changed.
- (5) `.max()` computes the maximum value over the list with all the numbers of versions of the various `REWARDTABLEROWS`.

Essentially, if the query returns 1 we know that the reward values have remained the same, whereas if it returns 2 or higher we will know that it has changed at some point, and depending on the value we will know how often it happened. For this experiment, the query returned 442 - there was a reward table row that had changed that many times in value.

Developers can easily expand upon the queries to produce more nuanced results. Listing 3 shows a more advanced example that computes some basic descriptive statistics of the reward table rows

Listing 4: EOL query to detect the longest sequence of action thrashing within the SAS

```

1 var decVersions = Decision.latest.all.first.versions;
2 var dvBeforeLast = decVersions.subList(1, decVersions.size);
3 var dvWithObs = dvBeforeLast.collect(dv | Sequence {
4   dv.time, dv.actionTaken.name, dv.supportingObservations()
5 });
6
7 // Find longest sequence of actions supported by 1 observation
8 var longestThrashing : Sequence;
9 var lastStart = -1;
10 for (i in 0.to(dvWithObs.size() - 1)) {
11   var nObs = dvWithObs.at(i).at(2);
12   if (nObs > 1) {
13     if (lastStart > -1 and i - lastStart > longestThrashing.size) {
14       longestThrashing = dvWithObs.subList(lastStart, i);
15     }
16     lastStart = -1;
17   } else if (lastStart = -1) {
18     lastStart = i;
19   }
20 }
21 return longestThrashing;
22
23 // Count supporting observations for a decision
24 operation Decision supportingObservations() : Integer {
25   var timeNextDecision = self.next.time;
26   var result = 0;
27   var observation = self.observation;
28   while (observation.time < timeNextDecision) {
29     observation = observation.next;
30     result += 1;
31   }
32   return result;
33 }

```

over time. This query makes use of EOL context operations to define the “reward shifts” of a specific version of a REWARDTABLE-Row. If we have only one version, it is the empty sequence. If it has more than one version, then it is the sequence of differences between the values of each version and the one immediately before it. For values (0.1, 0.12, 0.11) we would have shifts of (0.02, -0.01). This can give us an idea of whether the reward recalculation is keeping shifts bounded, or if the values are wildly shifting from one timepoint to the next. In the case of the current log, the SAS kept shifts bounded to a symmetric range within ± 0.034 , with an average of -5.31×10^{-9} .

Continuing with the theme of checking if the SAS is behaving appropriately, it may be important to notice situations in which the system may be “thrashing” between two actions, which suggest that the decision process may benefit from a “tolerance interval” where it will not react just yet to an observed situation. Listing 4 shows a query designed to find the longest sequence of actions that are only backed by a single observation, i.e. intervals in which the

Listing 5: EOL query to find cases when observations clash against the understanding of satisficement within the SAS

```

1 var mecBelief = NFRBelief.latest.all.selectOne(
2   nfrb | nfrb.nfr.name='Minimization of Energy Consumption'
3 );
4 var recMeasurement = Measurement.latest.all.selectOne(
5   m | m.metric.name = 'Ranges of Energy Consumption'
6 );
7
8 var contradictions : Sequence;
9 for (v in recMeasurement.versions) {
10   var vMecBelief = mecBelief.travelInTime(v.time);
11   var meets = vMecBelief.estimatedProbability
12     >= vMecBelief.nfr.threshold;
13   if (meets and v.measurementPosition >= 2) {
14     contradictions.add(Sequence {
15       v.time, meets, v.eContainer.description});
16   } else if (not meets and v.measurementPosition <= 1) {
17     contradictions.add(Sequence {
18       v.time, meets, v.eContainer.description});
19   }
20 }
21
22 return contradictions;

```

system kept changing action after each observation. It is a rather complex query, but it can be broken down as follows:

- (1) Lines 1–5 go from the earliest to the second last versions of the only DECISION in the RDM SAS. For each of them, they compute a triplet with the timepoint, the name of the action taken, and the number of supporting observations.
- (2) The number of supported operations is defined by the context operation in lines 24–33, which counts the number of observations that existed before the next version of that decision.
- (3) Lines 8–21 find the longest sequence of triplets with 1 supporting observation. For our trace, the query finds a sequence of 8 timepoints when the SAS is switching back and forth between RT and MST after each observation.

Queries can also be used to find less intuitive scenarios. Being probabilistic, R-POMDP may infer that a certain NFR is not being met even though the current observation may say otherwise: this simulates sensor failures and noise. Listing 5 shows a query which found 18 time slices when the Minimization of Energy Consumption NFR satisfaction did not match the Rate of Energy Consumption measurement. Either the NFR was met even though we were in the high ranges of REC, or the NFR was not met even though we were in the low ranges of REC. As a minor detail, for each version of the measurement we check the belief level at the same timepoint by using `travelInTime` on the belief node.

4.3 Time-aware queries for users

Other queries may be more generally useful to the wider community around the SAS, and could be fed into dashboards. They would

Listing 6: EOL query to compute statistics about NFR satisfaction above their thresholds

```

1 return NFRBelief.latest.all.collect(nfrb | nfrb.stats());
2
3 operation NFRBelief stats() {
4   var versions = self.versions;
5   var nAbove = versions.select(v |
6     v.estimatedProbability >= v.nfr.threshold).size;
7   var nBelow = versions.size - nAbove;
8   return Map { 'name' = self.nfr.name,
9     'above' = nAbove, 'below' = nBelow };
10 }

```

essentially start off from the NFRs and give increasingly more detailed explanations of to what degree they were met, what was done to correct situations when they were not met, and why those corrective actions were chosen.

Listing 6 shows a query which indicates how often the various NFRs were met. The query takes all the NFRBELIEF instances, and visits all their versions, counting how many are above and below their thresholds⁷. We compute a simple triplet with the name of the NFR and the number of times we believed it to be above/below the threshold. Regarding MEC, out of the 888 unique belief levels stated by the SAS, 670 passed the threshold and 218 did not. 665 belief levels passed MR, and 223 did not.

Deeper self-explanation requires looking at how the satisfaction of the MRs evolved over time, and how the system reacted to it. Listing 7 shows a query that will produce a timeline of how the NFRs changed between being met and not met, as shown in Listing 8.

Looking at line 1 of the output, we see that the system started with both MR and MEC unmet, that it stayed like that for 1 observation, and that since it observed that REC was low and NCC was high, it decided to go with RT as an action. Line 2 shows that the system started meeting MR and MEC, but then observed energy usage (REC) to be in the high ranges, so it went into MST. Interestingly, MEC started to fail later on, even though the observed energy usage was not that high: again, this may be due to the probabilistic nature of R-POMDP observed in Listing 5.

In general, the main advantage of the presented approach is that it allows for rapid development and iteration of new queries on the history of the model, making it possible to create the explanations for a category of SAS as required, and then later package them as premade, reusable visualizations.

5 RELATED WORK

This paper is based on a combination of various results from the areas of self-explanation for decision making systems, and model versioning. In this section we will relate the work to several key contributions in these fields.

Listing 7: EOL query to find intervals of MR satisfaction states and reactions by the SAS upon the observations made.

```

1 var vNfrMecB = NFRBelief.latest.all.selectOne(nfrb
2   | nfrb.nfr.name = 'Minimization of Energy Consumption'
3   ).versions.reverse();
4
5 var currentStates = computeStates(vNfrMecB.first.time);
6 var newStates : Map;
7 var results : Sequence;
8 var length = 0;
9 for (v in vNfrMecB) {
10   newStates = computeStates(v.time);
11   if (newStates.equals(currentStates)) {
12     length += 1;
13   } else {
14     var lastDecision = Decision.latest.all.first.travelInTime(v.time);
15     results.add(Sequence { currentStates, length,
16       v.time, // time of last decision taken in this interval
17       lastDecision.observation.description,
18       lastDecision.actionTaken.name // name of action
19     });
20     currentStates = newStates;
21     length = 1;
22   }
23 }
24 return results;
25
26 operation computeStates(instant: Integer): Map {
27   var nfrbs = NFRBelief.latest.all
28     .collect(nfrb | nfrb.travelInTime(instant));
29   var result : Map;
30   for (nfrb in nfrbs) {
31     result.put(nfrb.nfr.name,
32       nfrb.estimatedProbability >= nfrb.nfr.threshold);
33   }
34   return result;
35 }

```

Listing 8: Excerpt of output from Listing 7 about justification of the actions taken by the system.

```

1 [[{Maximization of Reliability=false, Minimization of Energy
2   Consumption=false}, 1, 1532385574820, REC LOWER X
3   AND NCC GREATER S, Redundant Topology],
4   [{Maximization of Reliability=true, Minimization of Energy
5     Consumption=true}, 1, 1532385575022, REC IN Y_Z AND
6     NCC GREATER S, Minimum Spanning Tree Topology],
7   [{Maximization of Reliability=true, Minimization of Energy
8     Consumption=false}, 1, 1532385575166, REC LOWER X
9     AND NCC GREATER S, Minimum Spanning Tree Topology
10  ],
11  ...]

```

⁷Interestingly, this query could easily accommodate dynamic NFR thresholds without any changes, since we visit the NFR through the version of the belief.

5.1 Decision making, self-explanation, interactive diagnosis

The area of research about self-explanation and interactive is still in its infancy. The need for it is exacerbated due to the use of artificial intelligence and machine learning. However, few research initiatives exist. The authors in [3] use goal-based requirements models at runtime to offer self-explanation of how a system is meeting its requirements. Our case study also contemplates the use of runtime goal-based models but supported by POMDPS. Different from the work in [3], our work uses Bayesian learning. Further, new future versions of the temporal graph models will be seen as runtime models to be consulted at runtime [4] to support decision making.

In [23], and as in our case, the authors present a temporal model to support interactive diagnosis of self-adaptive systems. The authors describe a temporal data model to represent, store and query decisions as well as their relationship with the context, requirements, and adaptation actions. So far, we do not include the context or the requirements, however it is part of our future research avenues. They have used their approach in the area of smart grids while we have used RDMs as the case study. While they use Greycat, only we have extended the Hawk model indexer with the ability to use Greycat as a backend. Using a model indexer makes it possible to reason over temporal graphs without the need of making changes to the existing system.

5.2 Model versioning

As a complex artifact developed within teams, keeping track of the various revisions that a model goes through is very important. According to the survey by Brosch et al. [5], versioning approaches can be classified across two orthogonal dimensions: the way they represent the artefacts, and the way they identify, represent and merge differences between versions. Artifact representations can be *text-based* as in most well-known tools (e.g. Subversion or Git), where they are seen as a collection of lines, or can be *graph-based* as a collection of nodes and edges, potentially with attributes and labels. Merging two versions developed in parallel from a common ancestor can be done in two ways: by comparing their states, or by combining the operations that were applied on the common ancestor in each side.

In terms of tools, many practitioners use simple and mature text-based version control systems (VCS) to keep track of their models (e.g. Git), and they use standalone state-based model comparison and merging tools (e.g. EMF Compare⁸). Others accept the additional complexity for the sake of additional functionality and use dedicated model repositories, which handle model revisions in terms of model elements and their references. Some well-known examples are Eclipse Connected Data Objects⁹, which stores model revisions inside a relational database (usually combined with a relational database), or EMFStore [19], which actually uses a collection of XMI files. EMFStore is interesting in that it keeps both the states of the various revisions, and the individual changes that were applied between those revisions, so it may use those for merging.

In the last few years, there has been increasing interest in having time-awareness as a native capability of the modelling framework itself. In 2012, Holmes et al. implemented a copy-on-write versioning scheme for models at an element level using UUIDs [16]. Hartmann showed in 2014 [14] a first version of the Kevoree Modelling Framework that supported reusable versioning of individual model elements, with the ability to travel back and forth in time. This would eventually evolve to their standalone Greycat temporal graph database. Our proposal takes this element-level versioning idea as a base, and proposes a general approach to use it for self-explanation across a variety of SAS, adding a generic metamodel for execution traces and a easy-to-use, database-agnostic query language. We have also integrated Greycat with a model indexer, making it possible to reason over temporal graphs without needing to re-engineer existing systems.

Another recent work in the area of temporal graph stores for models is ChronoSphere, developed by Haeusler et al. [12]. Similar to Greycat, it is also based on a key-value store where the key combines the timepoint and the element identifier. Unlike Greycat, which is a “pure” temporal graph database, the authors report capabilities as a model repository, supporting branches (without merges, for now), transactions, and some capabilities for metamodel evolution. The authors also mention the application of ChronoSphere in an “industrial IT Landscape Management tool” called Txture for model-based visualizations. We intend to evaluate ChronoSphere as an alternative to Greycat in future versions of our approach, in terms of performance and feature set.

One last idea that Borsch et al. identified in their survey as an open research area was *intention-aware model versioning*, where merges could be simplified by encoding what the modelers wanted to accomplish with their changes. We have not found many research initiatives in the area since then, but we find that the idea of encoding the intentions of a change in the model could certainly be relevant and useful for self-explanation. For future work, we are considering model versioning approaches where the self-adaptive model-based system would encode their intentions upon the various changes. These intentions could also be indexed by Hawk into the temporal graph, and could be accessed from EOL queries.

6 CONCLUSIONS AND FUTURE WORK

In this work, we have described the key requirements for a reusable framework for self-explanation in self-adaptive systems: a generic and extensible execution trace metamodel, a temporal graph to store these traces, a time-aware query language that allows to reason about the history of the models, and a set of reusable visualizations the main types of self-adaptive systems in the wild. We have provided proof-of-concept implementations for the first three. We have also demonstrated different queries aimed at explaining the self-adaptive nature of the systems to developers and end users.

The present work can be considered as a first step towards that reusable framework, suggesting several lines of future work. First, it would be useful to have a taxonomy of the various types of queries that different audiences may ask of a self-adaptive system, and the different levels of detail that we could use for our answers. Some of those queries may cross over multiple types of SAS, while others may be specific to a class of SAS. Once we determine which queries

⁸<https://www.eclipse.org/emf/compare/downloads/>

⁹<https://www.eclipse.org/cdo/>

are the most valuable and reusable, we would develop visualizations based on them.

The trace execution metamodel shown in the paper captured most concepts used by the RDM SAS, but it may require further refinement to cover other SAS. We need to apply the approach to a wider range of SAS and to allow the extension of the trace execution metamodel with “profiles” for other types of self-adaptation approaches. This is a similar approach to UML and the use of profiles for specific domains.

The query language is based on a set of basic primitives around versions, but it lacks the richness of a more formal model such as linear temporal logic, with richer predicates such as “always”, “never” or “eventually”.

We also envision a different application of temporal graphs to produce better simulation models for the development of self-adaptive systems. If we kept track of the actions and their impacts in a self-adaptive system “deployed in the wild”, we could produce a better probability matrix between NFR satisfaction levels, actions and observations. Further, we envision that the temporal graph models will act as runtime models to support self-explanation, interactive diagnosis and even decision-making. The language will provide a way how to access and change the runtime model during execution.

REFERENCES

- [1] Barmpis, K., Shah, S., Kolovos, D.S.: Towards Incremental Updates in Large-Scale Model Indexes. In: Taentzer, G., Bordeleau, F. (eds.) *Modelling Foundations and Applications*, pp. 137–153. No. 9153 in *Lecture Notes in Computer Science*, Springer International Publishing (Jul 2015)
- [2] Bencomo, N., Belaggoun, A., Issarny, V.: Dynamic decision networks to support decision-making for self-adaptive systems. In: (SEAMS) (2013)
- [3] Bencomo, N., Welsh, K., Sawyer, P., Whittle, J.: Self-explanation in adaptive systems. In: 17th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2012, Paris, France, July 18-20, 2012. pp. 157–166 (2012). <https://doi.org/10.1109/ICECCS.2012.34>
- [4] Blair, G., France, R.B., Bencomo, N.: Models@ runtime. *Computer* **42**, 22–27 (10 2009). <https://doi.org/10.1109/MC.2009.326>
- [5] Brosch, P., Kappel, G., Langer, P., Seidl, M., Wieland, K., Wimmer, M.: An Introduction to Model Versioning. In: Bernardo, M., Cortellessa, V., Pierantonio, A. (eds.) *Formal Methods for Model-Driven Engineering*, vol. 7320, pp. 336–398. Springer Berlin Heidelberg, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30982-3_10
- [6] Daniel, G., Sunyé, G., Benelallam, A., Tisi, M., Vernageau, Y., Gómez, A., Cabot, J.: NeoEMF: A multi-database model persistence framework for very large models. *Science of Computer Programming* **149**, 9–14 (Dec 2017). <https://doi.org/10.1016/j.scico.2017.08.002>
- [7] Forcher, B., Agne, S., Dengel, A., Gillmann, M., Roth-Berghofer, T.: Semantic logging: Towards explanation-aware DAS. In: 2011 International Conference on Document Analysis and Recognition, ICDAR 2011, Beijing, China, September 18-21, 2011. pp. 1140–1144 (2011). <https://doi.org/10.1109/ICDAR.2011.230>
- [8] Fredericks, E.M.: Mitigating uncertainty at design time and run time to address assurance for dynamically adaptive systems. Michigan State University. PhD Thesis. (2015)
- [9] García-Domínguez, A., Barmpis, K., Kolovos, D.S., Wei, R., Paige, R.F.: Stress-testing remote model querying APIs for relational and graph-based stores. *Software & Systems Modeling* pp. 1–29 (Jun 2017). <https://doi.org/10.1007/s10270-017-0606-9>
- [10] García-Domínguez, A., Bencomo, N.: Non-human modelers: Challenges and roadmap for reusable self-explanation. In: *Software Technologies: Applications and Foundations - STAF 2017 Collocated Workshops*, Marburg, Germany, July 17-21, 2017, Revised Selected Papers. pp. 161–171 (2017). https://doi.org/10.1007/978-3-319-74730-9_14
- [11] Giese, H., Bencomo, N., Pasquale, L., Ramirez, A.J., Inverardi, P., Wätzoldt, S., Clarke, S.: *Living with Uncertainty in the Age of Runtime Models*, pp. 47–100. Springer International Publishing, Cham (2014). https://doi.org/10.1007/978-3-319-08915-7_3
- [12] Haeusler, M., Trojer, T., Kessler, J., Farwick, M., Nowakowski, E., Breu, R.: Combining Versioning and Metamodel Evolution in the ChronoSphere Model Repository. In: *SOFSEM 2018: Theory and Practice of Computer Science*. pp. 153–167. *Lecture Notes in Computer Science*, Edizioni della Normale, Cham (Jan 2018). https://doi.org/10.1007/978-3-319-73117-9_11
- [13] Hartmann, T., Fouquet, F., Jimenez, M., Rouvoy, R., Traon, Y.L.: Analyzing Complex Data in Motion at Scale with Temporal Graphs. In: *Proceedings of the 29th International Conference on Software Engineering & Knowledge Engineering (SEKE'17)*. pp. 596–601 (Jul 2017). <https://doi.org/10.18293/SEKE2017-048>
- [14] Hartmann, T., Fouquet, F., Nain, G., Morin, B., Klein, J., Barais, O., Le Traon, Y.: A Native Versioning Concept to Support Historized Models at Runtime. In: Dingel, J., Schulte, W., Ramos, I., Abrahão, S., Infran, E. (eds.) *Model-Driven Engineering Languages and Systems*, vol. 8767, pp. 252–268. Springer International Publishing, Cham (2014). https://doi.org/10.1007/978-3-319-11653-2_16
- [15] Holme, P., Saramäki, J.: Temporal networks. *Physics Reports* **519**(3), 97–125 (Oct 2012). <https://doi.org/10.1016/j.physrep.2012.03.001>
- [16] Holmes, T., Zdun, U., Dustdar, S.: Automating the Management and Versioning of Service Models at Runtime to Support Service Monitoring. In: 2012 IEEE 16th International Enterprise Distributed Object Computing Conference. pp. 211–218. IEEE, Beijing, China (Sep 2012). <https://doi.org/10.1109/EDOC.2012.32>
- [17] Ji, M., Veitch, A.C., Wilkes, J., et al.: Seneca: remote mirroring done write. In: *USENIX Annual Conference*. pp. 253–268 (2003)
- [18] Kephart, J.O., Chess, D.M.: The vision of autonomic computing. *Computer* **36**(1), 41–50 (Jan 2003). <https://doi.org/10.1109/MC.2003.1160055>
- [19] Koegel, M., Helming, J.: EMFStore: A Model Repository for EMF Models. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*. pp. 307–308. ICSE '10, ACM, New York, NY, USA (2010). <https://doi.org/10.1145/1810295.1810364>
- [20] Kolovos, D.S., Paige, R.F., Polack, F.: The Epsilon Object Language (EOL). In: *Model Driven Architecture - Foundations and Applications*, Second European Conference, ECMDA-FA 2006, Bilbao, Spain, July 10-13, 2006, *Proceedings*. pp. 128–142 (2006). https://doi.org/10.1007/11787044_11
- [21] Kostakos, V.: Temporal graphs. *Physica A: Statistical Mechanics and its Applications* **388**(6), 1007–1023 (Mar 2009). <https://doi.org/10.1016/j.physa.2008.11.021>
- [22] Monahan, G.E.: State of the Art—A Survey of Partially Observable Markov Decision Processes: Theory, Models, and Algorithms. *Management Science* **28**(1), 1–16 (Jan 1982). <https://doi.org/10.1287/mnsc.28.1.1>
- [23] Mouline, L., Benelallam, A., Fouquet, F., Bourcier, J., Barais, O.: A temporal model for interactive diagnosis of adaptive systems. In: 2018 IEEE International Conference on Autonomic Computing, ICAC 2018, Trento, Italy, September 3-7, 2018 (2018)
- [24] Paucar, L.H.G., Bencomo, N.: RE-STORM: mapping the decision-making problem and non-functional requirements trade-off to partially observable Markov decision processes. In: *Proceedings of the 13th International Conference on Software Engineering for Adaptive and Self-Managing Systems*. pp. 19–25. SEAMS '18, ACM, New York, NY, USA (2018). <https://doi.org/10.1145/3194133.3195537>
- [25] Paucar, L.H.G., Bencomo, N., Fung Yuen, K.K.: Juggling Preferences in a World of Uncertainty. *RE NEXT*, Lisbon. (2017)
- [26] Ramirez, A., Cheng, B., Bencomo, N., Sawyer, P.: Relaxing claims: Coping with uncertainty while evaluating assumptions at run time. *MODELS* (2012)
- [27] Rose, E., Segev, A.: TOODM: A Temporal Object-Oriented Data Model with Temporal Constraints. Tech. Rep. LBL-30678; CONF-9110316–1, Lawrence Berkeley Lab., CA (United States) (Apr 1991). <https://www.osti.gov/scitech/biblio/5969182>
- [28] Roth-Berghofer, T., Tintarev, N., Leake, D.B. (eds.): *Explanation-aware Computing*, Papers from the 2011 IJCAI Workshop, Barcelona, Spain, July 16-17, 2011 (2011)
- [29] Sawyer, P., Bencomo, N., Whittle, J., Letier, E., Finkelstein, A.: Requirements-aware systems: A research agenda for RE for self-adaptive systems. In: *Proceedings of the 2010 18th IEEE International Requirements Engineering Conference*. pp. 95–103. RE '10, IEEE Computer Society, Washington, DC, USA (2010). <https://doi.org/10.1109/RE.2010.21>
- [30] Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: *EMF: Eclipse Modeling Framework*. Addison Wesley, Upper Saddle River, NJ, 2 edition edn. (Dec 2008)
- [31] Welsh, K., Bencomo, N., Sawyer, P., Whittle, J.: Self-explanation in adaptive systems based on runtime goal-based models. *Trans. Computational Collective Intelligence* **16**, 122–145 (2014). https://doi.org/10.1007/978-3-662-44871-7_5