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

A. García-Domínguez, N. Bencomo, Luis H. García Paucar

MRT'18, 14 October 2018

# Motivation

# Gaining trust on a self-adaptive system (SAS)

**Emergent behaviour in self-adaptive systems**

- Self-adaptive systems build a model of how their part of the world works, and how they can best meet their goals
- This results in emergent behaviour: it may be correct and meet the goals, but it could seem not immediately “obvious” to users or even developers

**Good self-explanation is needed**

- Developers can use it to debug the system and check if it meets certain safety criteria
- Users can ask questions and gain more confidence about why the SAS did something at a certain moment

# Reflection in self-adaptive systems

**Natural "compression" in most systems**

- Many SAS work by building a neural network / Bayesian network / other model from observations and decisions.
- In a way, this packs the history of the system into knowledge, but it may be "lossy", i.e. impossible to retrieve the original system state.
- It may be based on uncertain information: what if we kept track of how uncertain it was?
- What if we could look back at an explicit history of the system?

**Adding reflective capabilities to systems**

- What could we do if the system could ask explicit questions about its own past behaviour and its consequences?
- Could we make the system make better decisions?
- Could we make the system make more understandable decisions?

# Types of traces

**Traditional approach: textual output as we go along**

- Plain logging framework, just printing whatever we decide
- Easy to implement, easy for devs to read — hard to parse!

**Better: structured traces**

- Generate an XML/JSON snapshot of system state by timeslice
- Slightly more work, but computer-friendly
- What about history, though?

**We want trace models that are...**

- Based on a complete and reusable metamodel
- Stored in a way that allows "travelling in time"
- Answering questions about system history concisely
- Presenting answers in an accesible way

# Proposal

# Problem-independent trace models

**Many self-adaptive systems...**

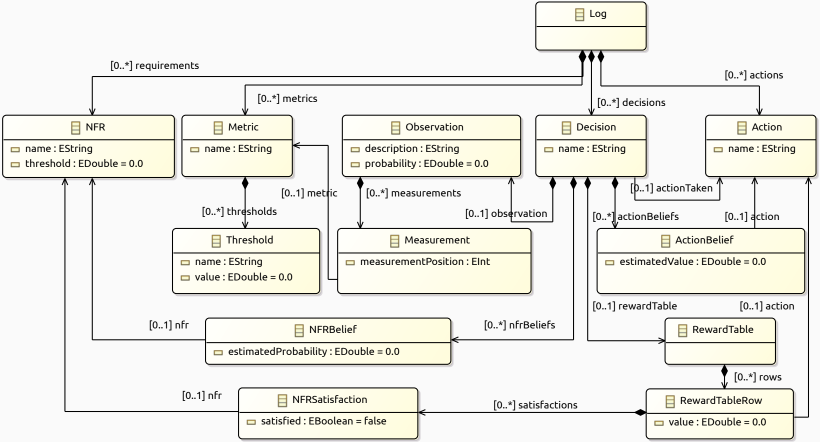
- Quantize time into *slices*
- Make observations and analyse them
- Plan future behaviour
- Execute plans

This is common in those based on the MAPE-K architecture.

**Can we define a domain-specific language for their state of mind?**

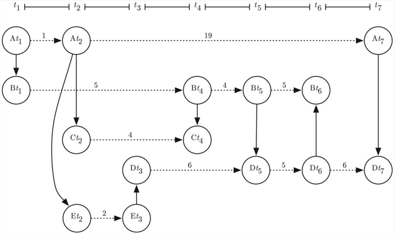
- We have a first version
- So far, tried it only on one SAS, though
- See next slide!

# Problem-independent trace metamodel

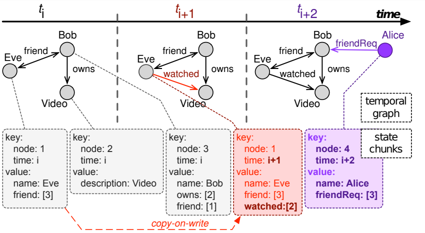


The SAS is trying to achieve NFRs by making a Decision among several Actions, guided by Observations of certain Metrics.

# What is a temporal graph?



Contact sequence (Kostakos)

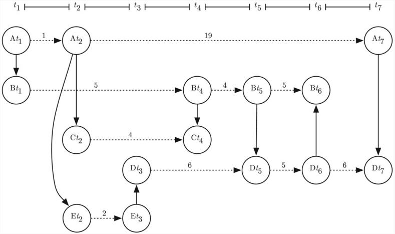

$t_i$
$t_{i+1}$
$t_{i+2}$
time
Bob
Eve
friend
owns
Video
Bob
Eve
friend
watched
owns
Video
Bob
friendReq
Alice
Eve
friend
watched
owns
Video
temporal graph
state chunks
key: node: 1 time: i value: name: Eve friend: [3]
key: node: 2 time: i value: description: Video
key: node: 3 time: i value: name: Bob owns: [2] friend: [1]
key: node: 1 time: i+1 value: name: Eve friend: [3] watched:[2]
key: node: 4 time: i+2 value: name: Alice friendReq: [3]
copy-on-write

Efficient storage (Hartmann)

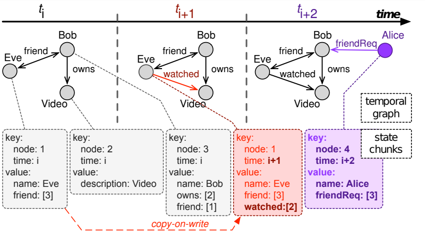
## Conflicting definitions

- Kostakos thinks about graphs of “contacts” between entities
- Hartmann focuses on efficient storage of a versioned graph

# What is a temporal graph?



Contact sequence (Kostakos)



$t_i$
$t_{i+1}$
$t_{i+2}$
time
Bob
Eve
friend
owns
Video
Bob
Eve
friend
watched
owns
Video
Bob
Eve
friend
watched
owns
Video
friendReq
Alice
temporal graph
state chunks
key: node: 1 time: i value: name: Eve friend: [3]
key: node: 2 time: i value: description: Video
key: node: 3 time: i value: name: Bob owns: [2] friend: [1]
key: node: 1 time: i+1 value: name: Eve friend: [3] watched:[2]
key: node: 4 time: i+2 value: name: Alice friendReq: [3]
copy-on-write

Efficient storage (Hartmann)

## Conflicting definitions

- Kostakos thinks about graphs of “contacts” between entities
- Hartmann focuses on efficient storage of a versioned graph

## In this paper...

We use the vision of temporal graphs from Hartmann, in its Greycat TGDB implementation.

# Transparent temporal graph storage

### Object-oriented models are time-evolving graphs

- Objects = nodes (labelled by type)
- References = edges
- Nodes and edges have attributes (object/reference fields)
- Overall: model = *labelled attributed graph* that evolves over time

### Hawk + Greycat: our implementation

- Hawk is our open-source heterogeneous model indexing framework: `https://github.com/mondo-project/mondo-hawk`
- Watches over the models and mirrors all history into a Greycat temporal graph, applying changes incrementally
- SAS models can be used as-is if based on EMF

# Reusable time-aware query language

**So far...**

- We have a representation of our SAS' state of mind
- We keep track of the full history of this representation
- We want to start asking questions — how do we express them?

**Our approach**

- Extend an existing query language, with good tool support
- Hawk already supported the Epsilon Object Language ($\sim$ OCL + JS)
- More info: `http://eclipse.org/epsilon`
- Added temporal extensions based on Rose and Segev's work in the 90s with *history objects* for object-oriented databases

# Time-aware primitives for the Hawk EOL dialect

### Model element history

- Limited lifespan
- Instance-based identity
- New version when state changes

### Type history

- Unlimited lifespan
- Name-based identity
- New version when instances are created or deleted

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

# Reusable visualizations

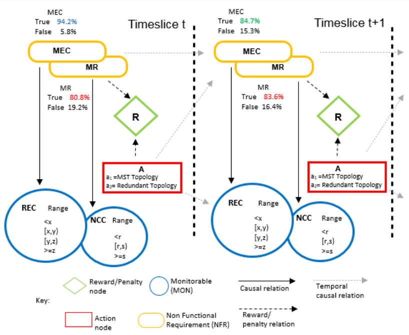
**Potential examples**

- Key instants in the SAS, with links to main changes in behaviour
- Predefined “why” questions for common queries:
  - Why was it doing this at this time?
  - Why did it stop doing the previous action?
  - Why was this change considered good?
  - What was the evaluation process like?
  - Why was the evaluation process configured that way?
- Visualizations would be bundled with the reusable representation and the appropriate time-aware queries

This is still work in progress!

# Case study

# Case study: Remote Data Mirroring



## Concept

SAS that protects against data loss by storing copies on servers.

## Configuration

- 2 topologies: min. spanning tree (MST), redundant (RT)
- 2 NFRs: max reliability (MR), min energy consumption (MEC)
- 2 monitoring variables: energy use, # of connections

## Goal

Switch between MST and RT to meet MR and MEC.

# Turning JSON traces into a temporal graph

```
{"0": {
  "current_belief_mec_true": 0.5, "current_belief_mr_true": 0.25,
  "current_observation_code": −1,
  "current_rewards": [[90.0, 45.0, 25.0, 5.0], [100.0, 10.0, 20.0, 0.0]],
  "ev.mst": 465.104345236406, "ev.rt": 326.710194366562,
  "flagUpdatedRewards": 0,
  "observation_description": "None", "observation_probability": 0.0,
  "selected_action": "MST"
 },
 "1": {
  "current_belief_mec_true": 0.94, ...
 },... }
```

**Steps taken**

1. Collected JSON traces from existing RDM SAS for 1000 time slices
2. Transformed traces to execution trace models
3. Turned sequence of trace models into a Subversion repository
4. Told Hawk to index all model revisions into a Greycat TGDB

# Turning JSON traces into a temporal graph

```
{"0": {
  "current_belief_mec_true": 0.5, "current_belief_mr_true": 0.25,
  "current_observation_code": −1,
  "current_rewards": [[90.0, 45.0, 25.0, 5.0], [100.0, 10.0, 20.0, 0.0]],
  "ev.mst": 465.104345236406, "ev.rt": 326.710194366562,
  "flagUpdatedRewards": 0,
  "observation_description": "None", "observation_probability": 0.0,
  "selected_action": "MST"
 },
 "1": {
  "current_belief_mec_true": 0.94, ...
 },... }
```

### Current study limitations

- No changes were made to the SAS in this study
- The current case study focuses on *forensic analysis*

# Turning JSON traces into a temporal graph

```
{"0": {
  "current_belief_mec_true": 0.5, "current_belief_mr_true": 0.25,
  "current_observation_code": −1,
  "current_rewards": [[90.0, 45.0, 25.0, 5.0], [100.0, 10.0, 20.0, 0.0]],
  "ev.mst": 465.104345236406, "ev.rt": 326.710194366562,
  "flagUpdatedRewards": 0,
  "observation_description": "None", "observation_probability": 0.0,
  "selected_action": "MST"
 },
 "1": {
  "current_belief_mec_true": 0.94, ...
 },... }
```

**Ideally...**

- The SAS would simply work off from a model indexable by Hawk, and Hawk + SAS would operate concurrently
- The SAS could ask Hawk about the history of the model to make history-aware decisions

# Queries for developers

- Self-explanations must be tailored to the target audience:

  **Developer** Wants to know things about the system

  **User** Wants to know if the NFRs were met, and how

  **System** Wants to find similar patterns to current observations, what went well, and not so well

- Self-explanations must focus on the intended use.
- As an example, for forensic analysis:
  - Is a certain desired property holding over time?
  - If not, when did the system misbehave?
  - Why did it misbehave then?

# Queries for developers: did reward values change?

Let's start with a simple one:

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

1. RewardTableRow.latest returns the latest version of the type node, with all the available instances (they are created once and never deleted),

# Queries for developers: did reward values change?

Let’s start with a simple one:

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

1. RewardTableRow.latest returns the latest version of the type node, with all the available instances (they are created once and never deleted),
2. .all returns all instances of the type at that point in time,

# Queries for developers: did reward values change?

Let’s start with a simple one:

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

1. RewardTableRow.latest returns the latest version of the type node, with all the available instances (they are created once and never deleted),
2. .all returns all instances of the type at that point in time,
3. .collect visits each instance and produces a sequence of results,

# Queries for developers: did reward values change?

Let’s start with a simple one:

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

1. RewardTableRow.latest returns the latest version of the type node, with all the available instances (they are created once and never deleted),
2. .all returns all instances of the type at that point in time,
3. .collect visits each instance and produces a sequence of results,
4. .versions.size measures how many times the rewards changed,

# Queries for developers: did reward values change?

Let's start with a simple one:

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

1. RewardTableRow.latest returns the latest version of the type node, with all the available instances (they are created once and never deleted),
2. .all returns all instances of the type at that point in time,
3. .collect visits each instance and produces a sequence of results,
4. .versions.size measures how many times the rewards changed,
5. .max() returns the number of times the most active reward table changed.

The most active reward table changed a total of 442 times in our traces.

# Queries for developers: distribution of shifts in reward values

```
var rs = RewardTableRow.latest.all.collect(row | row.getRewardShifts()).flatten();
return Sequence { rs.min(), rs.max(), rs.average() };

operation RewardTableRow getRewardShifts(): Sequence {
  var v = self.versions;
  if (v.size <= 1) { return Sequence {}; }
  else { return v.subList(0, v.size − 1).collect(v | v.value − v.prev.value); }
}
operation Sequence average() { return self.sum() / self.size(); }
```

1. RewardTableRow.latest.all returns all reward table rows,

# Queries for developers: distribution of shifts in reward values

```
var rs = RewardTableRow.latest.all.collect(row | row.getRewardShifts()).flatten();
return Sequence { rs.min(), rs.max(), rs.average() };

operation RewardTableRow getRewardShifts(): Sequence {
  var v = self.versions;
  if (v.size <= 1) { return Sequence {}; }
  else { return v.subList(0, v.size − 1).collect(v | v.value − v.prev.value); }
}
operation Sequence average() { return self.sum() / self.size(); }
```

1. RewardTableRow.latest.all returns all reward table rows,
2. .collect visits each instance and produces a sequence of results,

# Queries for developers: distribution of shifts in reward values

```
var rs = RewardTableRow.latest.all.collect(row | row.getRewardShifts()).flatten();
return Sequence { rs.min(), rs.max(), rs.average() };

operation RewardTableRow getRewardShifts(): Sequence {
  var v = self.versions;
  if (v.size <= 1) { return Sequence {}; }
  else { return v.subList(0, v.size − 1).collect(v | v.value − v.prev.value); }
}
operation Sequence average() { return self.sum() / self.size(); }
```

1. RewardTableRow.latest.all returns all reward table rows,
2. .collect visits each instance and produces a sequence of results,
3. .versions returns all versions from newest to oldest,

# Queries for developers: distribution of shifts in reward values

```
var rs = RewardTableRow.latest.all.collect(row | row.getRewardShifts()).flatten();
return Sequence { rs.min(), rs.max(), rs.average() };

operation RewardTableRow getRewardShifts(): Sequence {
  var v = self.versions;
  if (v.size <= 1) { return Sequence {}; }
  else { return v.subList(0, v.size − 1).collect(v | v.value − v.prev.value); }
}
operation Sequence average() { return self.sum() / self.size(); }
```

1. RewardTableRow.latest.all returns all reward table rows,
2. .collect visits each instance and produces a sequence of results,
3. .versions returns all versions from newest to oldest,
4. .prev compares adjacent values up to the second oldest version,

# Queries for developers: distribution of shifts in reward values

```
var rs = RewardTableRow.latest.all.collect(row | row.getRewardShifts()).flatten();
return Sequence { rs.min(), rs.max(), rs.average() };

operation RewardTableRow getRewardShifts(): Sequence {
  var v = self.versions;
  if (v.size <= 1) { return Sequence {}; }
  else { return v.subList(0, v.size - 1).collect(v | v.value - v.prev.value); }
}
operation Sequence average() { return self.sum() / self.size(); }
```

1. RewardTableRow.latest.all returns all reward table rows,
2. .collect visits each instance and produces a sequence of results,
3. .versions returns all versions from newest to oldest,
4. .prev compares adjacent values up to the second oldest version,
5. .min()/.max()/.average() compute descriptive statistics. 16 lines of EOL check that the SAS kept shifts bounded to ±0.034.

# Queries for developers: more examples

**Thrashing between two actions?**

- Is the SAS constantly changing actions?
- 33 lines of EOL later, we found a sequence of 8 timeslices in which the SAS was switching between RT and MST.
- Would this be acceptable behaviour for a SAS?

# Queries for developers: more examples

**Thrashing between two actions?**

- Is the SAS constantly changing actions?
- 33 lines of EOL later, we found a sequence of 8 timeslices in which the SAS was switching between RT and MST.
- Would this be acceptable behaviour for a SAS?

**Unintuitive inferences?**

- Does the SAS think at some point that a NFR is not being met, even if the observation says it is within range?
- Wrote query in 22 lines of EOL — there are 18 time slices in which the SAS thought the MEC NFR was not being met, even though there is low energy usage.
- Is this obvious? We may need to look at neighbouring observations to find out.

# Queries for users: enabling exploration

**Differences from developer-oriented queries**

- These are problem-centric, rather than solution-centric
- Were my NFRs met? If not, what was done about it, and why?
- Queries may need to be organised in a way that promotes exploration

# Queries for users: enabling exploration

**Differences from developer-oriented queries**

- These are problem-centric, rather than solution-centric
- Were my NFRs met? If not, what was done about it, and why?
- Queries may need to be organised in a way that promotes exploration

**Initial dashboard-style query: overall system health**

- We defined a query in 10 lines that measures number of versions in which each NFR is met and unmet
- MEC: SAS had 670 belief levels as “met” out of 888
- MR: SAS had 665 belief levels as “met” out of 888

# Queries for users: timeline views

Listing 1: Excerpt of output from query

```
[[{Maximization of Reliability=false, Minimization of Energy Consumption
    =false}, 1, 1532385574820, REC LOWER X AND NCC GREATER S, Redundant
     Topology],
[{Maximization of Reliability=true, Minimization of Energy Consumption=
    true}, 1, 1532385575022, REC IN Y_Z AND NCC GREATER S, Minimum
    Spanning Tree Topology],
[{Maximization of Reliability=true, Minimization of Energy Consumption=
    false}, 1, 1532385575166, REC LOWER X AND NCC GREATER S, Minimum
    Spanning Tree Topology],
...]
```

- Queries bring together beliefs and observations at decision points:
    - Beliefs can be simplified to yes/no if deemed useful.
    - Observations can be translated from the solution domain (e.g. “code 3”) back to the problem domain (“high use of energy”).
- We can simplify the timeline to points where the decision changed.
- The paper shows a 35-line query that does this.

# Conclusion and future work

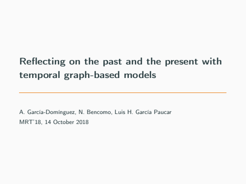
### Key points

- Text logs, structured traces not enough for good self-explanation
- For reusable self-explanation and reflection in SASs, we need:
  - A reusable trace metamodel
  - A transparent way to store versioned models as temporal graphs
  - A reusable time-aware querying language
  - A set of reusable visualizations based on the trace metamodel
- We showed the approach on a small case study (RDM)

### Future lines of work

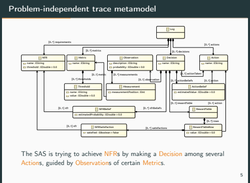
- Build visualizations!
- Create a taxonomy of typical questions to ask from a SAS
- Try out the trace metamodel on more SAS, and allow "profiling"
- Extend the time-aware primitives to cover linear temporal logic
- Leverage time-awareness for better simulations and decision-making

# Thank you!



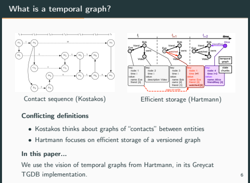
## Reflecting on the past and the present with temporal graph-based models

A. García-Domínguez, N. Bencomo, Luis H. García Paucar
MRT'18, 14 October 2018



### Problem-independent trace metamodel

The SAS is trying to achieve NFRs by making a Decision among several Actions, guided by Observations of certain Metrics.



### What is a temporal graph?

Contact sequence (Kostakos)

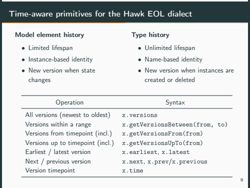
Efficient storage (Hartmann)

**Conflicting definitions**

- Kostakos thinks about graphs of "contacts" between entities
- Hartmann focuses on efficient storage of a versioned graph

**In this paper...**

We use the vision of temporal graphs from Hartmann, in its Greycat TGDB implementation.



### Time-aware primitives for the Hawk EOL dialect

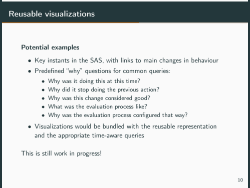
**Model element history**

- Limited lifespan
- Instance-based identity
- New version when state changes

**Type history**

- Unlimited lifespan
- Name-based identity
- New version when instances are created or deleted

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

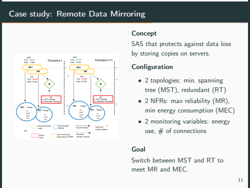


### Reusable visualizations

**Potential examples**

- Key instants in the SAS, with links to main changes in behaviour
- Predefined "why" questions for common queries:
  - Why was it doing this at this time?
  - Why did it stop doing the previous action?
  - Why was this change considered good?
  - What was the evaluation process like?
  - Why was the evaluation process configured that way?
- Visualizations would be bundled with the reusable representation and the appropriate time-aware queries

This is still work in progress!



### Case study: Remote Data Mirroring

**Concept**

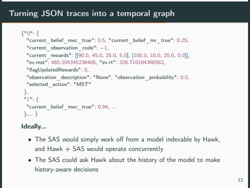
SAS that protects against data loss by storing copies on servers.

**Configuration**

- 2 topologies: min. spanning tree (MST), redundant (RT)
- 2 NFRs: max reliability (MR), min energy consumption (MEC)
- 2 monitoring variables: energy use, # of connections

**Goal**

Switch between MST and RT to meet MR and MEC.

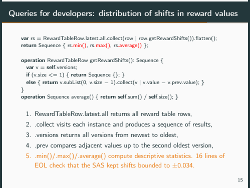


### Turning JSON traces into a temporal graph

```
{"0": {
  "current_belief_mec_true": 0.5, "current_belief_mr_true": 0.25,
  "current_observation_code": −1,
  "current_rewards": [[90.0, 45.0, 25.0, 5.0], [100.0, 10.0, 20.0, 0.0]],
  "ev.mst": 465.104345236406, "ev.rt": 326.710194366562,
  "flagUpdatedRewards": 0,
  "observation_description": "None", "observation_probability": 0.0,
  "selected_action": "MST"
 },
 "1": {
  "current_belief_mec_true": 0.94, ...
 },... }
```

**Ideally...**

- The SAS would simply work off from a model indexable by Hawk, and Hawk + SAS would operate concurrently
- The SAS could ask Hawk about the history of the model to make history-aware decisions

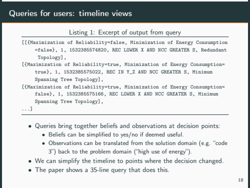


### Queries for developers: distribution of shifts in reward values

```
var rs = RewardTableRow.latest.all.collect(row | row.getRewardShifts()).flatten();
return Sequence { rs.min(), rs.max(), rs.average() };

operation RewardTableRow getRewardShifts(): Sequence {
  var v = self.versions;
  if (v.size <= 1) { return Sequence {}; }
  else { return v.subList(0, v.size − 1).collect(v | v.value − v.prev.value); }
}
operation Sequence average() { return self.sum() / self.size(); }
```

1. RewardTableRow.latest.all returns all reward table rows,
2. .collect visits each instance and produces a sequence of results,
3. .versions returns all versions from newest to oldest,
4. .prev compares adjacent values up to the second oldest version,
5. .min()/.max()/.average() compute descriptive statistics. 16 lines of EOL check that the SAS kept shifts bounded to ±0.034.



### Queries for users: timeline views

Listing 1: Excerpt of output from query

```
[[{Maximization of Reliability=false, Minimization of Energy Consumption
    =false}, 1, 1532385574820, REC LOWER X AND NCC GREATER S, Redundant
     Topology],
[{Maximization of Reliability=true, Minimization of Energy Consumption=
    true}, 1, 1532385575022, REC IN Y_Z AND NCC GREATER S, Minimum
    Spanning Tree Topology],
[{Maximization of Reliability=true, Minimization of Energy Consumption=
    false}, 1, 1532385575166, REC LOWER X AND NCC GREATER S, Minimum
    Spanning Tree Topology],
...]
```

- Queries bring together beliefs and observations at decision points:
  - Beliefs can be simplified to yes/no if deemed useful.
  - Observations can be translated from the solution domain (e.g. "code 3") back to the problem domain ("high use of energy").
- We can simplify the timeline to points where the decision changed.
- The paper shows a 35-line query that does this.