Automated Provenance Graphs for models@run.time

Owen Reynolds
180200041@aston.ac.uk
SEA research group, Aston University
Birmingham, UK

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

Nelly Bencomo
n.bencomo@aston.ac.uk
SEA research group, Aston University
Birmingham, UK

ABSTRACT

Software systems are increasingly making decisions autonomously by incorporating AI and machine learning capabilities. These systems are known as self-adaptive and autonomous systems (SAS). Some of these decisions can have a life-changing impact on the people involved and therefore, they need to be appropriately tracked and justified: the system should not be taken as a black box. It is required to be able to have knowledge about past events and records of history of the decision making. However, tracking everything that was going on in the system at the time a decision was made may be unfeasible, due to resource constraints and complexity. In this paper, we propose an approach that combines the abstraction and reasoning support offered by models used at runtime with provenance graphs that capture the key decisions made by a system through its execution. Provenance graphs relate the entities, actors and activities that take place in the system over time, allowing for tracing the reasons why the system reached its current state. We introduce activity scopes, which highlight the high-level activities taking place for each decision, and reduce the cost of instrumenting a system to automatically produce provenance graphs of these decisions. We demonstrate a proof of concept implementation of our proposal across two case studies, and present a roadmap towards a reusable provenance layer based on the experiments.

CCS CONCEPTS

• **Software and its engineering** → **Abstraction, modeling and modularity**; • **Computing methodologies** → *Model development and analysis*; • **Applied computing** → *Evidence collection, storage and analysis.*

KEYWORDS

Provenance, PROV-DM, autonomous decision-making, self-explanation, runtime models.

ACM Reference Format:
Owen Reynolds, Antonio García-Domínguez, and Nelly Bencomo. 2020. Automated Provenance Graphs for models@run.time. In *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20 Companion), October 18–23, 2020, Virtual Event, Canada.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3417990.3419503

1 INTRODUCTION

Autonomous decision making is on the rise with Artificial Intelligence (AI) and machine learning (ML) enabling software systems to perform an increasing number of real world tasks. A report by Crawford et al. raises a vast number of issues with the social impacts of AI [7]. For example, the report raises important questions about algorithmic accountability and impact assessments, outlining the rise of legislation, which seek to make AI vendors and consumers assess the impact of their applications on people's lives.

Self-adaptive autonomous systems (SAS) exhibit autonomous decision making. SAS are built to change their behaviour according to fluctuations in the environment under uncertainty, and as such they can exhibit emergent behaviours with unintended consequences [31]. Even if the system is successfully achieving its goal, the user may still have some reservations based on its potentially erratic behaviour. Enabling a system so it can track and keep a history of its actions would be one way to expose reasons for behaviour.

Self-explainable systems contribute solutions to the challenges in providing assurances for self-adaptive systems [2]. Further, explanations and findings for bad decisions are important for informing the corrections that may be required, such as reversing incorrect decisions or fixing a defect in a system. Beyond explaining incorrect decisions, confirming a system decision is correct for the right reasons should be considered as well.

This paper proposes an architecture to collect data from a system at runtime that can be used to seek explanations for its actions. The architecture can be added to a system to enable the production of data which will assist in explaining runtime behaviours. A proof of concept implementation is presented and demonstrated in two case studies: a toy example based on the Fibonacci sequence, and a traffic light controller for a traffic simulator. Both systems under study are based on runtime models that abstract the internal state of the system and guide their decisions. By monitoring the interactions with the system models, data is produced and stored to create a history of the systems' actions that can be explored.

The paper is presented in the following sections. The underlying ideas about model versioning, provenance and runtime models are provided in Section 2. Section 3 describes the various components involved in the architecture and their relations. Section 4 presents a proof of concept implementation on top of existing technologies. Two case studies using this implementation are shown in Section 5. Finally, Section 6 closes the paper with a discussion of what has been achieved so far and outlines future lines of research.

2 BACKGROUND

2.1 Tracking how models change over time

The need for tracking changes on models is seen in different domain areas. In the area of model-driven software development, creating large systems requires teams of developers working together, in the same way as code-centric development [6]. These teams need to be able to concurrently work on different parts of the model and integrate those parts back into a unified model. Therefore, a versioning system that can record and integrate these concurrent revisions is required.

One approach for model versioning is to use traditional Version Control Systems (VCS) such as Git [28]. The models are persisted to files, and these files are tracked by the VCS over their various revisions. However, a traditional VCS generally compares and merges versions using line-based tools, which are better suited for general-purpose languages than models. Dedicated model repositories such as Eclipse Connected Data Objects (CDO) allow for storing models directly into databases [8], solving some of the scalability issues with monolithic file-based models. They can also integrate with model-specific comparison tools like EMF Compare [9].

In either case, these approaches are typically limited to recalling past versions, and they do not provide facilities for searching versions of interest. As they generally store snapshots of the models, recovering the sequence of changes that were made requires expensive model comparison processes. Rather than having model-wide revisions, it may be more useful to keep a history of every model element separately across time. This is an approach implemented by *temporal graph databases* such as Greycat [18]. Each node (model element) in the graph has its own history, and it is inexpensive to travel back and forth over it. This simplifies writing queries that span time, but it still does not track the reasons for the changes.

A filmstrip model aims at describing a sequence of system state transitions [16]. Examples of filmstrips for models are presented by Hilken and Gogolla [20]. A filmstrip is a sequence of model snapshots, connected by a description of the operation that produced the change. Hilken and Gogolla showed how OCL could be applied to the filmstrip model to query its evolution over time. However, the proposal did not tackle potential scalability issues due to the high cost of keeping full snapshots for each change process.

We argue that the scalable approaches to track model changes lack causal information that can be used to inform explanations for a change. The filmstrip approach is close to what is required but lacks a clear vision for how the approach could scale to long-running systems (e.g. by leveraging Greycat/CDO). A combined approach that can scale as well as keep casual information would be ideal for these explanations.

2.2 Tracking why models change using provenance

Existing model versioning approaches track how models changed, but they do not explicitly represent *why* they changed, which is important for explanations. Such a representation would need to consider who made the change due to which concern, and what information was consulted during the change. This is a problem studied within the field of *data provenance*. In their systematic review [24], Pérez defines provenance to be "all the information and relationships that contributed to the existence of a piece of data". Provenance was first related to works of art, but the concepts have been applied to other use cases such as scientific experiments [19].

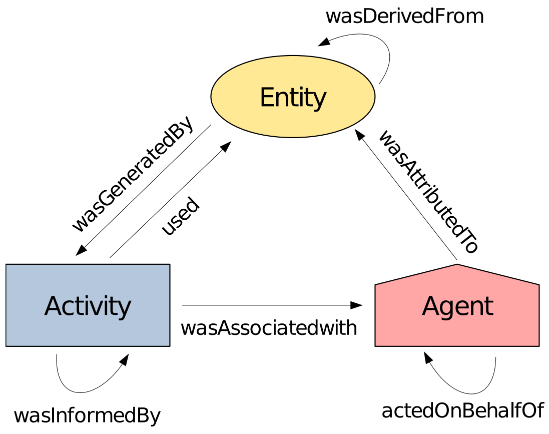


Figure 1: W3C PROV

The Open Provenance Model (OPM) is an ontology that has been created to record provenance information [22], in a way that promotes sharing and reuse. OPM is the base for the W3C PROV provenance ontology. PROV provides several notations, one being the PROV Data Model (PROV-DM) [17]. The model provides three classes of objects and seven relationships for representing provenance graphs (Figure 1). An `Agent` denotes a user or system that has some form of responsibility. An `Activity` is an act performed by an agent which may act on entities. An `Entity` represents a physical or abstract thing manipulated by the system.

The references connecting these objects describe their relationships. An activity has *"used"* an entity, which may produce an entity which *"wasGeneratedBy"* an activity. When two activities share an entity that *"wasGeneratedBy"* one and *"used"* by the other, one activity *"wasInformedBy"* by the other. An agent *"wasAssociatedWith"* an activity which it was responsible for. This responsibility can be passed to another agent who *"actedOnBehalfOf"* when performing an activity. Entities created in these activities can be connected with the responsible agent as the creation *"wasAttributedTo"* them. Finally, entity versions can be related with *"wasDerivedFrom"*.

Data can be collected from complex systems and stored using provenance. An example of such a framework is presented for tracking interactions between an application and operating systems [14]. Activities within the applications are not exposed as the monitoring only seeks to capture interactions with system-calls and resources of an operating system. The results of this provenance based approach to diagnostics helps locate points for instrumentation.

There are some specific applications which have used provenance to explain behaviours. Kohwalter et al. applied provenance graphs to capturing game telemetry, to therefore improve the understanding of player behaviours [21]. By augmenting the Unity game engine to produce W3C PROV-N, and alternative notation to PROV-DM [30]. The author concluded the provenance knowledge aided in the detection of gameplay issues. An informed reason for the issues helped developers to improve the gameplay.

In closing, the literature shows that while standardised formalisations for provenance do exist, these do not seem to have caught on in the modelling community. By integrating Activity-Agent-Entity provenance graphs (such as those in PROV-DM) with existing model versioning approaches could allow for a reusable and extensible way to represent the reasons for changes in models. However, it raises questions about who would produce such data in the case of manual, design-time modelling. Automated changes done to a model by a running system could be instrumented to create these graphs; however: the next section will discuss this possibility and its applications.

2.3 Provenance at runtime

Model-driven engineering (MDE) puts models at the centre of the design and development of the system [26], raising the level of abstraction at which they happen. Runtime models use this power of abstraction during the execution of the system [5]. This allows systems to be self-aware about their modelled concerns, such as architectural aspects [12, 23] and requirements [4]. Runtime models can be implemented using EMF and CDO as Seybold et al. have demonstrated [27], or using goal models as in [4, 25].

Many self-aware systems operate as feedback loops, and MAPE-K is a common architecture for those [1, 3, 15]. The MAPE-K loop is divided into the four phases of Monitoring, Analysis, Planning, and Execution, operating on top of shared Knowledge, which is often a runtime model.

Runtime models can also be used to support self-explanation. Different from our previous work [11], in this paper we use model versioning and provenance to support understanding of the decision making process. Runtime models present an opportunity to capture the evolution of a system using model versioning. However, there is a need for more than periodic or episodic capturing of a model to provide explanations. To have a system explain the *why* behind its changes, provenance tells us what information should be collected. When a system changes its runtime model, it can track the Activity-Agent-Entity triple that caused the change.

3 PROPOSAL FOR REUSABLE AUTOMATED PROVENANCE GRAPHS

We propose an approach to implementing the automated generation of provenance graphs using runtime models and an established standard PROV-DM, which can underpin explanations of decision-making. The following requirements were identified:

- The approach must provide a way to instrument the system with *data loggers* to collect significant changes to its state as part of high-level activities. Changes can be considered significant if they impact the *system runtime model* (kept in a *model store*) that abstracts the key parts of the system state, and the system codebase can be annotated to relate those changes to high-level activities.
- The instrumentation that is added to the system should have minimal impact on performance and no impact on behaviour.
- The *monitor* will collect a large volume of data; therefore, a more scalable approach is required. Filtering unnecessary details and pruning old data is also needed.

The requirements have informed the architecture proposed. The architecture is shown in Figure 2, and consists of:

- **Activity scopes** allow the identification of the current activity being performed by the system. They essentially wrap around a block of code in the original system, associating all model accesses with a high-level activity named by the system developer (e.g. "monitor road activity" or "update traffic lights"). Activity scopes can be nested, as activities can be broken down into subactivities.
- **Instrumented objects** are used between a system and its runtime model. They produce *Object telemetry* as a system interacts with its runtime model. The *Object telemetry* data details the model parts and values with the interaction type that occurred (read/write).
- The **observer** collects the data from the instrumented objects, and combines it with the currently running activity, creating a sequence of *activity-aware model accesses*. It is responsible for maintaining the stack of nested activities.
- The **curator** takes the activity-aware accesses of the observer, and updates the **history store** based on them. The history store will normally be a large model repository, leveraging a scalable model persistence solution. It is structured as a sequence of *chronopoints*, which collect data about the behaviour of the system between two points in time. A chronopoint starts from a *snapshot* of the system's runtime model, and it has a provenance graph of all the accesses performed since then and until the end of the chronopoint.
The provenance graphs follow the W3C PROV entity-activity-agent structure, as shown in Section 2.2 and Figure 1. The curator adds *inferred relationships* of various types, creating additional causal connections between activities and entities.

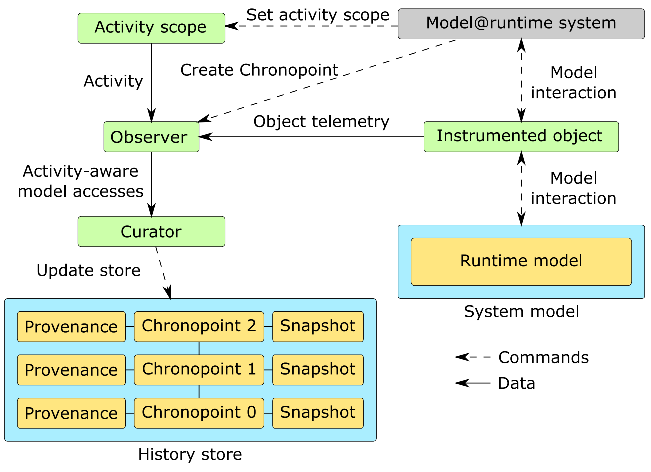


Figure 2: Architecture for a reusable provenance layer

This architecture allows for the state of the system at a certain point in time to be recreated by starting from the snapshot and replaying the provenance graph. At the same time, it is possible to discard old history when it becomes irrelevant for the study of the system since the various chronopoints are independent of each other. It is also possible to apply storage management techniques, such as being able to discard snapshots or set retention periods.

Likewise, the separation of the history store from the storage of the runtime model allows for more flexibility. For instance, some runtime models could remain entirely in memory as the system runs, while the history store would use a database-backed solution for longer-term storage. This separation also makes it possible to implement the runtime model and the history store in different modelling technologies.

4 PROOF OF CONCEPT IMPLEMENTATION

In order to demonstrate the high-level approach outlined in Figure 2, an initial proof-of-concept implementation was developed using the Java programming language. The Eclipse Modelling Framework was chosen as the base technology layer [29], due to its wide availability and the familiarity of the authors with it. The main components were implemented as follows:

- **Activity scopes** essentially need to associate blocks of code in the system with the high-level activities performed by each agent. Since activities need to be pushed and popped from a stack in the observer, reliably detecting when we enter and leave the block (even in the presence of errors) is a must. For these reasons, Java *try-with-resource* blocks were used. These blocks allow for automatically allocating a resource upon entry (i.e. pushing our entry onto the activity stack), and freeing it upon exit (popping the entry).
- For the **instrumented objects**, the implementation takes advantage of the fact that all model instances in EMF extend from a common `EObject` class, and that all accesses can be made to go through the `eGet` and `eSet` methods. By overriding these methods in a new `LoggingEObject` class, and reconfiguring the EMF code generator to use it as the base class for all our model instances, it is possible to capture all model accesses with minimal modifications to the system.
- We want to ensure that there is exactly one instance of the **observer** in the system to which all instrumented objects report, which keeps track of the currently running activity through a stack. A Singleton provides an easy solution for creating a single observer.
- If the model is accessed as part of an activity, the observer will let the **curator** know about it. The curator is yet another singleton, which takes the activity-aware model access and uses it to expand the provenance graph of the currently active chronopoint. The curator is also responsible for starting new chronopoints, creating their initial snapshots of the runtime model. The specific approach depends on the case study: a copy of the runtime model may be kept in the history store, or the runtime model may be versioned, and then only a link to the appropriate version will have to be kept.

  The provenance graphs are EMF models conforming to the metamodel in Figure 3. The curator creates links between the execution of an activity and the model entities it reads and writes. Reads are mapped to "activity used entity" relationships. Writes are mapped to "entity wasGeneratedBy activity" relationships, where the activity has generated a new version of that model entity. In addition to these *primary* links, a number of *inferred* links are also created: i) "activityA wasInformedBy activityB" indicates that the first activity used an entity that the second activity generated, and ii) "entityA wasDerivedFrom entityB" indicates that the left entity is a newer version of the right entity.
- The chronopoints, snapshots, and provenance graphs are kept in the **history store**. The history store needs to use a scalable model persistence solution, as it may potentially grow quite large. In this proof of concept implementation, Eclipse Connected Data Objects (CDO) model repositories were used [8]. CDO has several useful features, such as its support for *lazy loading*, which allows keeping in memory only the part of the model we are currently traversing, so that models larger than available memory can be managed.

5 CASE STUDIES

In order to evaluate the proposal, the proof of concept implementation was applied to two different case studies. The first case study is a toy example (based on Fibonacci numbers) which demonstrates the approach and allows us to examine an entire provenance graph. The second case study is based on a simplification of an agent-based application that controls the traffic lights in a simulation of a 4-way junction, which presents a more complex runtime model and internal logic than the Fibonacci case study.

In both cases, the system was implemented first, and then the provenance layer from Section 4 was added. Each system has its own EMF-based runtime models, and their structures have been kept as they were. The general steps for each system were: i) the base class of all model entities is changed to the instrumented base class, ii) the system instantiates the observer and curator on initialisation, iii) the system calls the observer at certain times to create chronopoints, and iv) the code is instrumented with activity scopes, delimiting the various system-specific high-level activities.

5.1 Fibonacci: exploring the provenance graph

The objective of the first case study is to produce meaningful provenance graphs that a human can understand. To do this, we took a program that calculated the Fibonacci sequence and augmented it with a runtime model to act as a source of behavioural data. Fibonacci provides a simple sequence of activities that uses a small number of variables and actions that can be clearly defined. We hypothesised the provenance graphs produced would be human-readable and easily interpreted when visualised.

Implementation. The runtime model is kept entirely in memory, and the history store is kept in CDO. Snapshots are omitted in this simple case study, as the focus is on the provenance graphs. We only keep a sequence of chronopoints with their provenance graphs. Referring back to Figure 2, the models@run.time-based system would be the calculator, and the runtime model would consist of the various variables involved in the calculation.

The observer is now created on startup, which creates the curator as well, and the main `fib()` routine was changed as in Listing 1. Two lines in red (3 and 26) ask the observer to set up a new chronopoint. The lines in blue correspond to the activity scopes, implemented as *try-with-resource* blocks. An activity scope is implemented through the "acquisition" of an `ActivityScope` with strings for the name of the activity and the actor, which is released automatically upon leaving the block. In this case, there is only one agent: the main thread of execution or "Thread1".

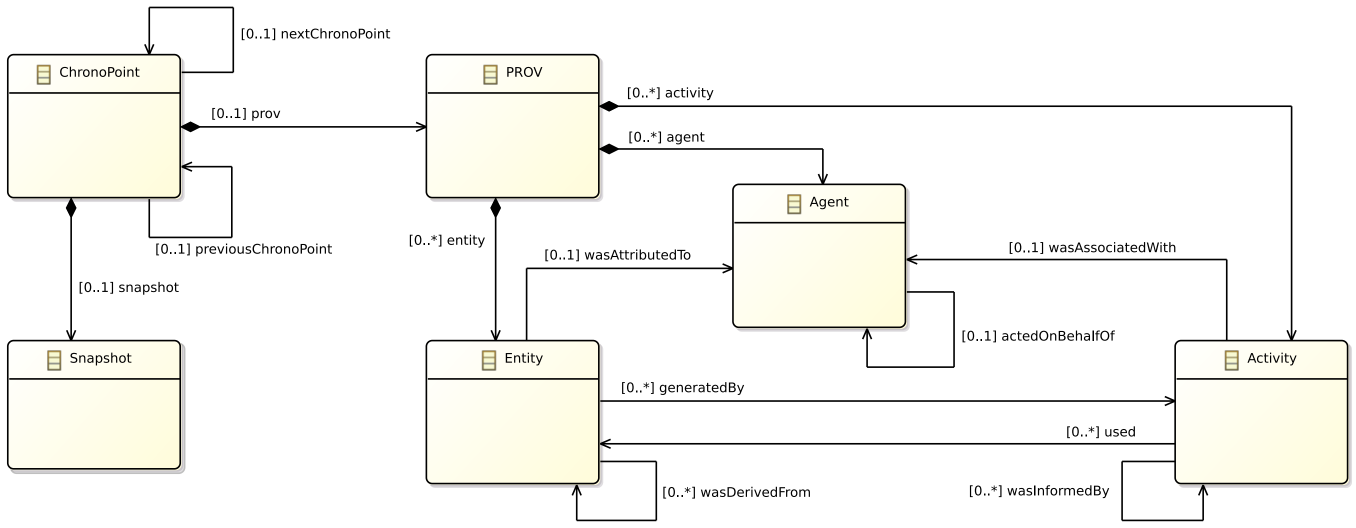


Figure 3: Provenance graph metamodel

Listing 1: Modified Fibonacci code

```
public static void fib() {
  Stringy Console; Inty A, B, C;
  myObserver.newChronopoint();
  try(var sc0 = new ActivityScope("init","Thread1")){
      Console = GPLDataFactory.createStringy();
      A = GPLDataFactory.createInty();
      B = GPLDataFactory.createInty();
      C = GPLDataFactory.createInty();
      try(var sc1 = new ActivityScope("init-names","Thread1")){
        A.setName("A"); B.setName("B"); C.setName("C");
        Console.setName("Console");
      }
      try(var sc1 = new ActivityScope("init-values","Thread1")){
        A.setValue(0); B.setValue(1); C.setValue(0);
        Console.setValue(null);
  }}
  try(var sc0 = new ActivityScope("Loop condition","Thread1")){
    do {
      try(var sc1 = new ActivityScope("A + B","Thread1"))
        { C.setValue(A.getValue()+B.getValue()); }
      try(var sc1 = new ActivityScope("B to A","Thread1"))
        { A.setValue(B.getValue()); }
      try(var sc1 = new ActivityScope("C to B","Thread1"))
        { B.setValue(C.getValue()); }
      // ... continued ...
      myObserver.newChronopoint();
    } while (A.getValue() < 255);
}}
```

Findings. After being fitted with the monitoring components, the system execution is unaffected. The Fibonacci algorithm outputs the sequence to the console as expected. Using CDO Explorer in the Eclipse IDE, it is only possible to examine individual provenance graph nodes via forms. To better illustrate the structure of the graph in practice, an automated model-to-text transformation was created to render a part of the provenance graph through Graphviz [10]. Entity and activity nodes have a sequence number prefixed to their labels based on creation order, and "then" relationships have been created to guide Graphviz on how to order the nodes. Colours are also applied to separate entities, agents and activities.

Figure 4 shows the activity and entity interactions for one pass of the loop in Listing 1. Agents are not shown, as this system has only one agent. The visualisation shows all the nodes and relationships in the provenance graph. The activity labels are visible in the blue rectangle nodes. Interaction with the runtime model creates entity nodes as yellow ovals labelled with the object, attribute and value accessed. While traversing the activity nodes on the graph shown in Figure 4, the following statements can be deduced.

```
[0] A loop condition test used the value of A.
[1] intyA and intyB are used in an "add" activity
    to generate a value stored in intyC.
[2] intyB is used to generate a value stored in intyA.
[3] intyC is used to generate a value stored in intyB.
    This activity was was informed by activity [1].
[4] The value and name in intyC is used to
    generate StringyConsole as an update.
    This activity was informed by activity [1].
[5] The value in StringyConsole is displayed.
    This activity was informed by activity [4].
```

With a more careful selection of activity labels, these statements could be produced through a model transformation. In larger systems this would only be advisable if sections of provenance could be isolated: for example, a query based approach could be used to produce smaller graphs for creating a narrative.

As mentioned in Section 4, the curator not only adds the *"used"* and *"wasGeneratedBy"* primary provenance relationships, but also infers the *"wasInformedBy"* and *"wasDerivedFrom"* relationships. In an initial version of the case study, the inferred provenance

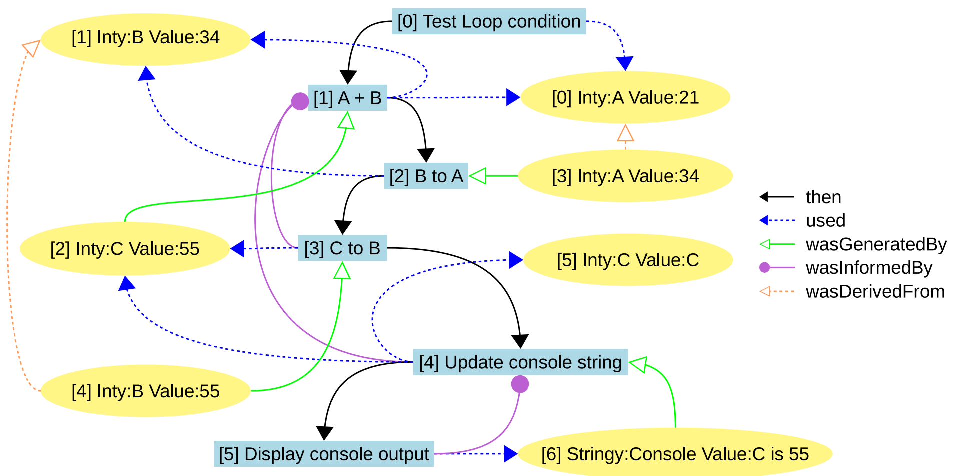


Figure 4: Provenance graph from the Fibonacci sequence

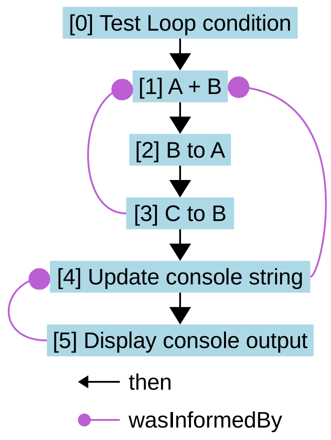


Figure 5: Activities only

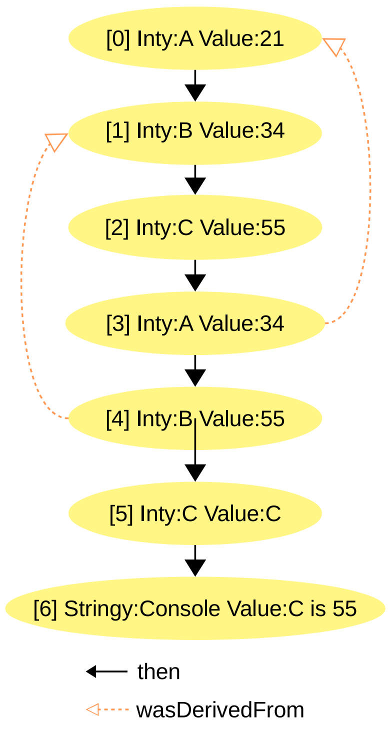


Figure 6: Entities only

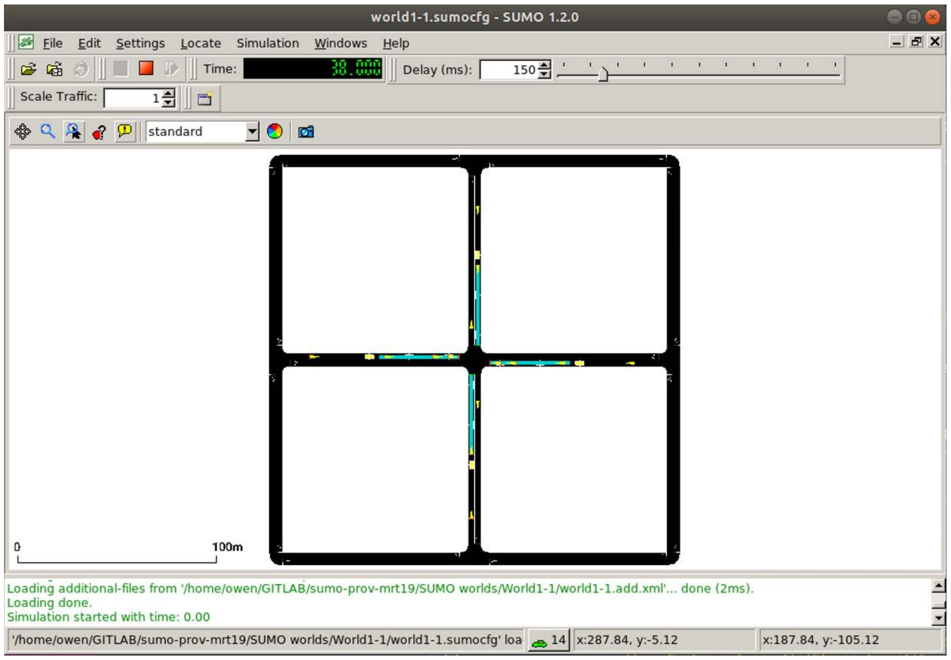
relationship *"wasDerivedFrom"* was initially defined as "an entity contributed to another entity", and it was computed by connecting the entity in "activity used entityA" to the entity in "entityB wasGeneratedBy activity". This produced many relationships, with a very dense graph that was difficult to understand when visualised with Graphviz. It was then that *"wasDerivedFrom"* was changed so it linked each entity to its previous version. The dense graph being difficult to understand could be a visualisation tool problem and not an issue with how the relationship is being applied.

Alternative transformations with different visualisations of the provenance graph were evaluated as well. For instance, it is possible to leave out the inferred edges (*"wasInformedBy"* and *"wasDerivedFrom"*) and slightly simplify the graph. Leaving out one of the node types can produce more significant simplifications: Figure 5 excludes the entity nodes, and Figure 6 excludes the activity nodes. These alternative visualisations tell us that an interactive visualisation tool that can selectively hide or show parts of a provenance graph on demand could be useful.

5.2 SUMO - Traffic controller

The previous case study was a simple example, designed so that the entire provenance graph could be examined at once. In the next case study, we examine the work involved in adapting an existing larger system, and demonstrate how the provenance graphs can provide further insight into the behaviour of the system.

System description. The system under study is a traffic controller within a simulation implemented in SUMO (Simulation of Urban



```
Loading additional-files from '/home/owen/GITLAB/sumo-prov-mrt19/SUMO worlds/World1-1/world1-1.add.xml'... done (2ms).
Loading done.
Simulation started with time: 0.00
'/home/owen/GITLAB/sumo-prov-mrt19/SUMO worlds/World1-1/world1-1.sumocfg' loa  14  x:287.84, y:-5.12  x:187.84, y:-105.12
```

Figure 7: SUMO simulating a 4-way traffic junction

Mobility [13]). SUMO is a simulator for road networks and traffic flows. The simulation models a 4-way intersection with traffic flows controlled by traffic lights (shown in Figure 7). Four flows of traffic are specified with some uncertainty in the driver behaviours, e.g. turning and braking. In the simulation, the traffic lights run a pre-programmed time sequence of light phases, and the environment does not directly influence this sequence. Our system under study is an external controller that manipulates this lighting sequence.

Our external self-adaptive controller monitors a series of lane area detectors on the lanes approaching the junction. The self-adaptive behaviour is a perception of how many stationary cars constitute a traffic jam. This level rises and falls over time, based on the number of lane area detectors meeting a threshold of stationary cars. Should several lane area detectors qualify as jammed, the controller may attempt to change the phase timer on the traffic lights. If this is successful, the current lighting phase ends and the traffic flow changes in an attempt to clear the jam condition. To be clear, this example controller is not intended to solve traffic management at junctions. However, the example strategy can be investigated and possibly improved using the monitoring system.

The traffic controller operates on a runtime model inspired by the MAPE-K, conforming to the metamodel shown in Figure 8. First, the Monitor phase synchronises the system with the SUMO simulation. The `MonitorControls` specify what needs to be read from SUMO, and the results are placed into the `MonitorResults`, `TrafficLight` and `LaneAreaDetector` objects that represent the monitoring results, and the traffic lights and LADs in the simulation.

The Analysis phase checks the internal models evaluating them against the `AnalysisControls`, which include the thresholds for considering if a lane area detector is jammed. Based on that comparison, it updates the `AnalysisResults` with its interpretations.

The Plan stage applies some strategies and then updates the `PlanToExecute`. The plan may decide to change the threshold for jamming, or that the current light phase should end.

Finally, the Execute stage tries to run those plans, which may fail because of hard limits that prevent unwanted rapid back-and-forth "thrashing" between states, and records the `ExecutionResults`.

Changes in infrastructure. In this case study, snapshots are implemented through CDO model versioning, where the model goes through a sequence of revisions. These revisions are introduced in a transactional manner: committing the transaction results in a new model revision being created.

The traffic manager model for the controller is placed in a CDO repository which is connected into the monitoring system. The observer was modified so that the `newChronopoint()` method integrated with the versioned CDO store for the runtime model. When the observer triggers a commit in CDO, it reports the timestamp and location of the snapshot to the curator. This information is then kept in a chronopoint as a surrogate of the snapshot.

Applying activity scopes. Unlike the Fibonacci example explained earlier, the traffic controller is implemented as a MAPE-K loop, calling a series of functions for various tasks. Each iteration of the loop becomes a chronopoint, contain a provenance graph of all the activities performed in a single step of the simulation.

The approach to implementing the activity scopes started with some broad activity scope allocations. Each phase of the MAPE loop was enclosed in a single scope, which resulted in a graph containing a few activity objects that are densely connected with too many entities. It was evident that further subdivisions of the code into activities was needed, as otherwise, the graph is difficult to understand lacking activity detail.

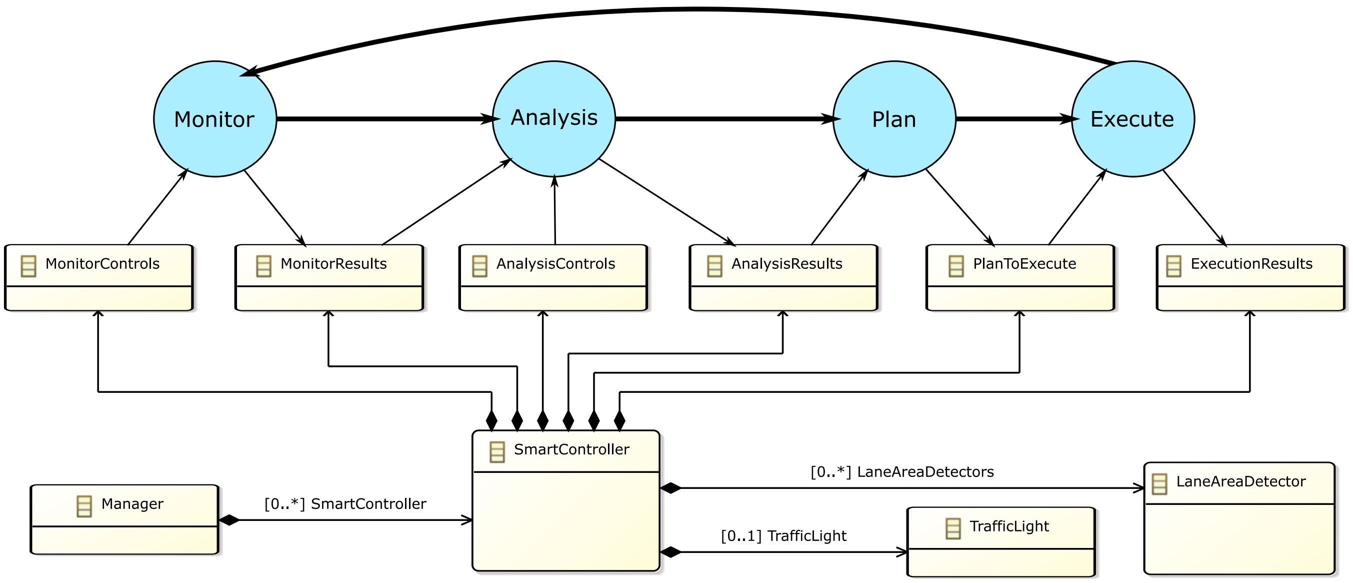


Figure 8: MAPE over Traffic Controller Model

A second phenomenon was spotted on this initial markup, where an inferred relationship *"wasInformedBy"* had been created on the Build Model activity that looped back to itself. This meant the activity had generated an entity and then used it for itself. During the initialisation of the controller, this is likely to happen as models for traffic lights, and lane area detectors are created then populated. We can, therefore, conclude that any activity that is self-informing may be divided into further sub-activities in some cases.

Activities based on loops can be represented differently based on the placement of the activity scope. A scope encapsulating a loop provides a single activity node, which may have several entities. Conversely, placing the scope inside of a loop presents a more linear representation of multiple activities, each with fewer entities. These loop representations need further investigation to discover meaningful differences in the approaches.

Graph complexity. The graphs for this case study are noticeably more complex than those in the previous study. The provenance graph in each of the 15 chronopoints of the Fibonacci case study only had 6 activity nodes and 7 entity nodes. However, in the SUMO case, each tick of the simulation (which represents around 1 second of real-world time) results in a chronopoint with around 18 activity nodes and 30 entity nodes. Unlike the Fibonacci calculation, the programs behaviour changes causing more or fewer node. In this case study, a typical simulation run of SUMO is around 500 ticks.

There is always the option to trade off taking some snapshots and create longer provenance graphs. An example use cases for this might be that a system may idle for long periods, adding little to a provenance graph over this time. In this case study, we found that taking snapshots every 10 ticks of the simulation produced 179 activity nodes and 100 entity nodes. Unsurprisingly, the number of activities increased proportionally. However, the entities did not increase as rapidly due to entity reuse. The increase in entity-to-activity references could complicate visualisation.

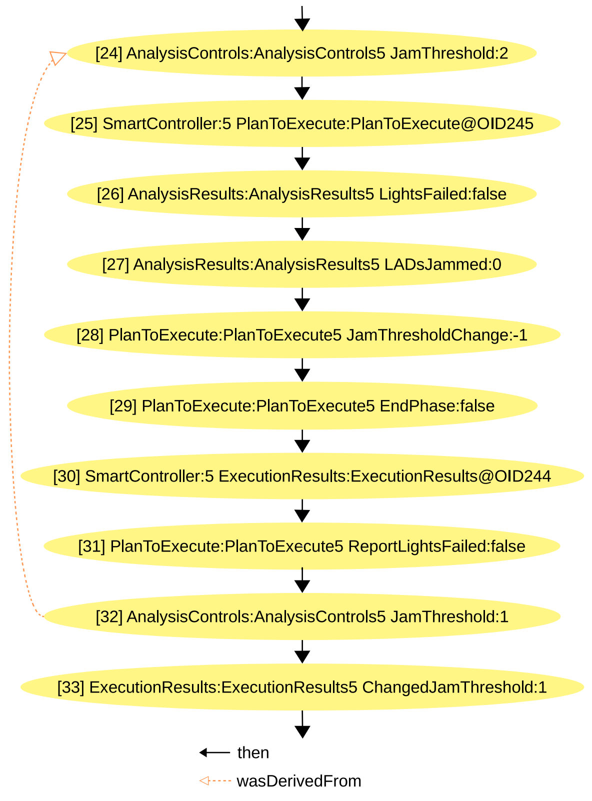

[24] AnalysisControls:AnalysisControls5 JamThreshold:2
[25] SmartController:5 PlanToExecute:PlanToExecute@OID245
[26] AnalysisResults:AnalysisResults5 LightsFailed:false
[27] AnalysisResults:AnalysisResults5 LADsJammed:0
[28] PlanToExecute:PlanToExecute5 JamThresholdChange:-1
[29] PlanToExecute:PlanToExecute5 EndPhase:false
[30] SmartController:5 ExecutionResults:ExecutionResults@OID244
[31] PlanToExecute:PlanToExecute5 ReportLightsFailed:false
[32] AnalysisControls:AnalysisControls5 JamThreshold:1
[33] ExecutionResults:ExecutionResults5 ChangedJamThreshold:1
then
wasDerivedFrom

Figure 9: Entity-only provenance for Jam Threshold being lowered

Checking expected behaviour. Exploring the data from SUMO is challenging with the limited tools available at this time. However, as

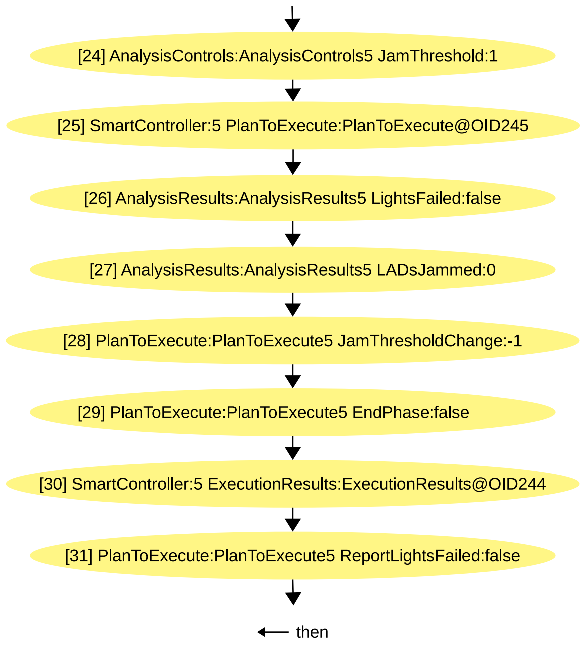

[24] AnalysisControls:AnalysisControls5 JamThreshold:1
[25] SmartController:5 PlanToExecute:PlanToExecute@OID245
[26] AnalysisResults:AnalysisResults5 LightsFailed:false
[27] AnalysisResults:AnalysisResults5 LADsJammed:0
[28] PlanToExecute:PlanToExecute5 JamThresholdChange:-1
[29] PlanToExecute:PlanToExecute5 EndPhase:false
[30] SmartController:5 ExecutionResults:ExecutionResults@OID244
[31] PlanToExecute:PlanToExecute5 ReportLightsFailed:false
then

Figure 10: As Figure 9, but Jam Threshold now at minimum level

the provenance graphs are exported from CDO to Graphviz files, it is possible to search through the 500 full graph text representations. With some prior knowledge of the system, it is possible to search for the presence of nodes in a graph to confirm expected behaviours. In the first few graphs of the set, it is possible to find the relaxing of the jam threshold, which is expected behaviour for an empty junction. From the entity-only graph, this is seen as a *"wasDerivedFrom"* connection between an earlier and later entity for the jam threshold (Figure 9). Figure 10 only shows a single entity representing the jam threshold as the value is unchanged, having reached the minimum value (1). However, *[28]PlanToExecute* (Figure 10) is showing a further attempt to reduce the jam threshold. In this instance, the condition is guarded against so as not to cause a problem.

Finding incorrect behaviour. Our ability to browse graphs is limited, thus exploring graphs to find faults is difficult. Therefore a reverse approach to a fault investigation is performed. We looked for symptoms our approach might show for a known fault as these are the clues someone with a tool might find. A runaway fault was created by removing the reset for the `LADsJammed` counter. Meaning that the counter is always exceeding the threshold to trigger a traffic light phase end, and it never recovers from this.

In this trivial example, the snapshot details might be considered sufficient, but using the provenance graph an exact moment between snapshots can be accessed. Performing a search against the graphs for the entity representing the `LADsJammed` counter, it is possible to find the moment the counter passed the threshold of 4 (Figure 11). Using a Graphviz viewer, the full provenance graph was examined to find the entity node for the threshold. The lane area detector analysis activity can be found, and the threshold and jam length entities can be seen showing the values used. The preceding lane area detector analysis activity can also be found, which did not increment `LADsJammed`, evident from the lack of a *"wasDerivedFrom"* on the entity. This activity can then be traced to planning activities for an end phase and threshold change. The execution of the plan is traced, with an attempt to end the phase and increase the threshold, but it is unsuccessful due to protections against thrashing.

The thrashing behaviour with attempting to end the lighting phase is also present in the graphs. Due to space constraints, Figure 11 omitted a relationship that showed this: a *"wasInformedBy"* link between a monitoring activity and the executing activity. Using a text search this relationship can be found in multiple graph files. Searching 500 graph files for a working and runaway system revealed a significant difference: a working system had 96 occurrences of *"wasInformedBy"*, compared to 479 in the runaway system.

6 CONCLUSION

This paper presents the work to date on an approach to automating the production of provenance for runtime models. Our contribution is an initial architecture based on runtime models and provenance to support explanations for runtime behaviours. The results of applying this architecture in two case studies support the ideas behind this approach. Model versioning and provenance can record runtime behaviours that contribute to an explanation.

A working proof of concept has been created that can be used against runtime models built in EMF with CDO versioning. The resulting provenance graphs for the case studies have shown they can represent runtime behaviours. A toy Fibonacci calculator can produce a provenance graph that represents the expected design-time behaviour. Graphs produced by the traffic controller could be searched for indicators of behaviour, such as ending a lighting phase too frequently. The provenance graphs provide causal information for the state of a runtime model at a point in time.

There are more questions to be answered beyond this proof of concept. Future works include investigating methods to analyse the provenance graphs. While these graphs may scale in terms of storage and structure, there are issues with trying to visualise large graphs without dedicated tools. Visualisation is only one approach to processing these graphs and other analysis techniques need to be explored. The text search in the SUMO case study suggests that querying will scale further.

Other research questions relate to the application of the approach against other types of systems. The approach to explaining behaviours with provenance is specifically demonstrated here for runtime self-adaptive system. However, the approach might also be used against more complex distributed or multi-threaded systems where tracing causation can be difficult.

Acknowledgements: The work was partially funded by the Leverhulme Trust Research Fellowship (Grant RF-2019-548) and the EPSRC Research Project Twenty20Insight (Grant EP/T017627/1).

REFERENCES

[1] P. Arcaini, E. Riccobene, and P. Scandurra. 2015. Modeling and Analyzing MAPE-K Feedback Loops for Self-Adaptation. In *Proceedings of SEAMS 2015*. 13–23. https://doi.org/10.1109/SEAMS.2015.10
[2] Nelly Bencomo, Sebastian Götz, and Hui Song. 2019. Models@run.time: a guided tour of the state of the art and research challenges. *Software & Systems Modeling* (2019). https://doi.org/10.1007/s10270-018-00712-x

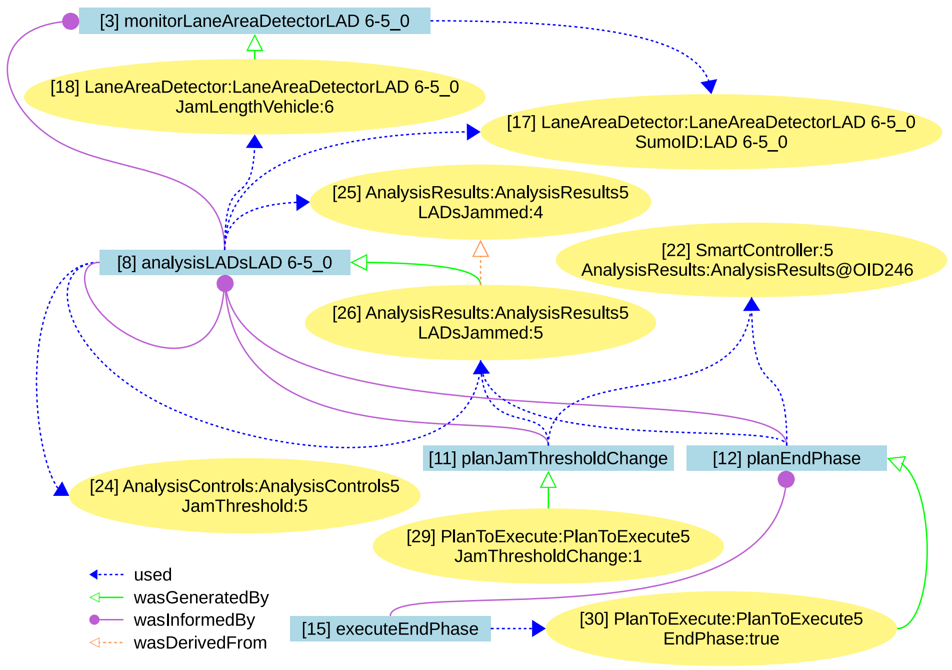

[3] monitorLaneAreaDetectorLAD 6-5_0
[18] LaneAreaDetector:LaneAreaDetectorLAD 6-5_0 JamLengthVehicle:6
[17] LaneAreaDetector:LaneAreaDetectorLAD 6-5_0 SumoID:LAD 6-5_0
[25] AnalysisResults:AnalysisResults5 LADsJammed:4
[8] analysisLADsLAD 6-5_0
[22] SmartController:5 AnalysisResults:AnalysisResults@OID246
[26] AnalysisResults:AnalysisResults5 LADsJammed:5
[11] planJamThresholdChange
[12] planEndPhase
[24] AnalysisControls:AnalysisControls5 JamThreshold:5
[29] PlanToExecute:PlanToExecute5 JamThresholdChange:1
used
wasGeneratedBy
wasInformedBy
wasDerivedFrom
[15] executeEndPhase
[30] PlanToExecute:PlanToExecute5 EndPhase:true

Figure 11: Cropped graph for runaway condition

[3] Nelly Bencomo and Luis Hernán García Paucar. 2019. RaM: Causally-Connected and Requirements-Aware Runtime Models using Bayesian Learning. In *Proc. of MODELS 2019*. IEEE, 216–226. https://doi.org/10.1109/MODELS.2019.00005

[4] Nelly Bencomo, Jon Whittle, Peter Sawyer, et al. 2010. Requirements reflection: requirements as runtime entities. In *Proceedings of ICSE 2010*. ACM, 199–202. https://doi.org/10.1145/1810295.1810329

[5] Gordon Blair, Nelly Bencomo, and Robert B. France. 2009. Models@ run.time. *Computer* 42, 10 (Oct 2009), 22–27. https://doi.org/10/bkpbtk

[6] Petra Brosch, Gerti Kappel, Philip Langer, et al. 2012. *An Introduction to Model Versioning*. Vol. 7320. Springer Berlin Heidelberg, 336–398. https://doi.org/10.1007/978-3-642-30982-3_10

[7] Kate Crawford, Roel Dobbe, Theodora Dryer, et al. 2019. *AI Now 2019 Report*. Technical Report. AI Now Institute. https://ainowinstitute.org/AI_Now_2019_Report.html Date last checked: February 28th, 2020.

[8] Eclipse Foundation. 2019. CDO Model Repository. https://www.eclipse.org/cdo/ Date last checked: February 25th, 2020.

[9] Eclipse Foundation. 2020. EMF Compare homepage. https://www.eclipse.org/emf/compare/ Date last checked: February 25th, 2020.

[10] John Ellson, Emden Gansner, Yifan Hu, et al. 2020. Graphviz - Graph Visualization Software. https://graphviz.org/ Date last checked: August 26th, 2020.

[11] A. García-Domínguez, N. Bencomo, J. M. Parra Ullauri, and L. H. García Paucar. 2019. Querying and annotating model histories with time-aware patterns. In *Proc. of MODELS 2019*. IEEE, 194–204. https://doi.org/10.1109/MODELS.2019.000-2

[12] David Garlan and Bradley R. Schmerl. 2004. Using Architectural Models at Runtime: Research Challenges. In *Proceedings of EWSA 2004 (LNCS, Vol. 3047)*. Springer, 200–205. https://doi.org/10.1007/978-3-540-24769-2_15

[13] German Aerospace Center. 2019. SUMO homepage. http://sumo.sourceforge.net/ Date last checked: February 25th, 2020.

[14] Eleni Gessiou, Vasilis Pappas, Elias Athanasopoulos, et al. 2012. Towards a Universal Data Provenance Framework Using Dynamic Instrumentation. In *Information Security and Privacy Research*. Vol. 376. Springer Berlin Heidelberg, 103–114. https://doi.org/10.1007/978-3-642-30436-1_9

[15] Holger Giese, Nelly Bencomo, Liliana Pasquale, et al. 2011. Living with Uncertainty in the Age of Runtime Models. In *Models@run.time - Foundations, Applications, and Roadmaps*. Lecture Notes in Computer Science, Vol. 8378. Springer, 47–100. https://doi.org/10.1007/978-3-319-08915-7_3

[16] Martin Gogolla, Lars Hamann, Frank Hilken, Mirco Kuhlmann, and Robert France. 2014. From Application Models to Filmstrip Models: An Approach to Automatic Validation of Model Dynamics. In *Modellierung 2014*. Gesellschaft für Informatik e.V., Bonn, 273–288.

[17] Paul Groth and Luc Moreau. 2013. *PROV-Overview*. Working Group Note. W3C. https://www.w3.org/TR/prov-overview/ Date last checked: February 25th, 2020.

[18] Thomas Hartmann, François Fouquet, Matthieu Jimenez, Romain Rouvoy, and Yves Le Traon. 2017. Analyzing Complex Data in Motion at Scale with Temporal Graphs. In *Proceedings of SEKE 2017*. https://doi.org/10.18293/SEKE2017-048

[19] Melanie Herschel, Ralf Diestelkämper, and Houssem Ben Lahmar. 2017. A survey on provenance: What for? What form? What from? *The VLDB Journal* 26 (10 2017). https://doi.org/10.1007/s00778-017-0486-1

[20] F. Hilken and M. Gogolla. 2016. Verifying Linear Temporal Logic Properties in UML/OCL Class Diagrams Using Filmstripping. In *2016 Euromicro Conference on Digital System Design (DSD)*. 708–713. https://doi.org/10.1109/DSD.2016.42

[21] T. Costa Kohwalter, L. Gresta Paulino Murta, and E. Walter Gonzalez Clua. 2017. Capturing Game Telemetry with Provenance. In *2017 16th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*. 66–75. https://doi.org/10.1109/SBGames.2017.00016

[22] Luc Moreau, Ben Clifford, Juliana Freire, et al. 2011. The Open Provenance Model core specification (v1.1). *Future Generation Computer Systems* 27, 6 (Jun 2011), 743–756. https://doi.org/10.1016/j.future.2010.07.005

[23] Brice Morin, Olivier Barais, Grégory Nain, and Jean-Marc Jézéquel. 2009. Taming Dynamically Adaptive Systems using models and aspects. In *Proceedings of ICSE 2009*. IEEE, 122–132. https://doi.org/10.1109/ICSE.2009.5070514

[24] Beatriz Pérez, Julio Rubio, and Carlos Sáenz-Adán. 2018. A systematic review of provenance systems. *Knowledge and Information Systems* 57, 3 (Dec 2018), 495–543. https://doi.org/10/gf8q84

[25] Peter Sawyer, Nelly Bencomo, Jon Whittle, et al. 2010. Requirements-Aware Systems: A Research Agenda for RE for Self-adaptive Systems. In *Proceedings of RE 2010*. IEEE Computer Society, 95–103. https://doi.org/10.1109/RE.2010.21

[26] E. Seidewitz. 2003. What models mean. *IEEE Software* 20, 5 (2003), 26–32. https://doi.org/10.1109/ms.2003.1231147

[27] Daniel Seybold, Jörg Domaschka, Alessandro Rossini, et al. 2016. Experiences of models@run-time with EMF and CDO. In *Proceedings of SLE 2016*. ACM Press, 46–56. https://doi.org/10.1145/2997364.2997380

[28] Software Freedom Conservancy. 2020. Git project homepage. https://git-scm.com/ Date last checked: February 25th, 2020.

[29] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2009. *EMF: Eclipse Modeling Framework 2.0* (2nd ed.). Addison-Wesley Professional. ISBN: 978-0-321-33188-5.

[30] Unity Technologies. 2020. Unity project homepage. https://unity.com/frontpage Date last checked: February 25th, 2020.

[31] Kris Welsh, Nelly Bencomo, Pete Sawyer, and Jon Whittle. 2014. *Self-Explanation in Adaptive Systems Based on Runtime Goal-Based Models*. Springer, 122–145. https://doi.org/10.1007/978-3-662-44871-7_5