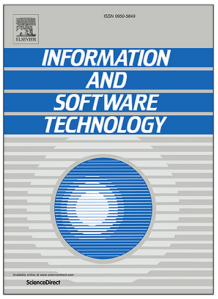
Journal Pre-proof



Cronista: A multi-database automated provenance collection system for runtime-models

Owen Reynolds, Antonio García-Domínguez, Nelly Bencomo

PII: S0950-5849(21)00149-X
DOI: https://doi.org/10.1016/j.infsof.2021.106694
Reference: INFSOF 106694

To appear in: *Information and Software Technology*

Received date : 19 February 2021
Revised date : 1 July 2021
Accepted date : 20 July 2021

Please cite this article as: O. Reynolds, A. García-Domínguez and N. Bencomo, *Cronista*: A multi-database automated provenance collection system for runtime-models, *Information and Software Technology* (2021), doi: https://doi.org/10.1016/j.infsof.2021.106694.

Highlights (for review)

- Cronista logs the provenance of changes to the runtime model used by a system to a history model, with the aim to assist in diagnosing system faults and behaviours.
- Applying Cronista's automated provenance collection approach has negligible impact on the target system's execution times whether storing provenance graphs on CDO or JanusGraph.
- We show how to investigate a seeded defect with provenance graphs stored in CDO and JanusGraph, and find that the declarative style in the Gremlin language of JanusGraph is easier to learn.

Cronista: a multi-database automated provenance collection system for runtime-models

OWEN REYNOLDS, ANTONIO GARCÍA-DOMÍNGUEZ, and NELLY BENCOMO, SEA research group, EPS, Aston University, United Kingdom

Context: Decision making by software systems that face uncertainty needs tracing to support understandability, as accountability is crucial. While logging has been essential to support explanations and understandability of behaviour, several issues still persist, such as the high cost for managing large logs, not knowing what to log, and the inability of logging techniques to relate events to each other or to specific occurrences of high-level activities in the system.

Objective: *Cronista* is an alternative to logging for systems that act on top of runtime models. Instead of targeting the running systems, *Cronista* automatically collects the provenance of changes made to the runtime models, which aim at leveraging high-level representations, to produce more concise historical records. The provenance graphs capture causal links between those changes and the activities of the system, which are used to investigate issues.

Method: *Cronista* 's architecture is described with the current design and the implementation of its high-level components for single-machine, multi-threaded systems. *Cronista* is applied to a traffic-simulation case study.The trade-offs of two different storage solutions are evaluated, i.e. the CDO model repositories, and JanusGraph graph databases.

Results: Integrating *Cronista* into the case study requires only minor code changes. *Cronista* collected provenance graphs for the simulations as they ran, using both CDO and JanusGraph. Both solutions highlighted the cause of a seeded defect in the system. For the longer executions, both CDO and JanusGraph showed negligible overhead on the simulation times. Querying and visualisation tools were more user-friendly in JanusGraph than in CDO.

Conclusion: *Cronista* demonstrates the feasibility of recording fine-grained provenance for the evolution of runtime models, while using it to investigate issues. User convenience and resource requirements need to be balanced. The paper present how the available technologies studied offer different trade-offs to satisfy the balance required.

CCS Concepts: • **Software and its engineering → System modeling languages**; **Integration frameworks**; **Model-driven software engineering**.

Additional Key Words and Phrases: Provenance, runtime-models, multi-threading, self-adaptation, self-explanation

1 INTRODUCTION

The increasing complexity of software systems entails uncertainty, making it difficult to determine the causes of behaviour at runtime [13]. An example of this can be seen in systems that offer concurrent activities, which present uncertainty about the order of events and interaction of activities that may occur at runtime [9]. In other cases, a system may need to make a decision over incomplete information, and it may be difficult to discern afterwards why that decision was made [5, 46].

Algorithmic accountability and the right to explanation is an important topic for software developers and society in general [47]. Explaining system behaviour requires runtime data to support the explanations [46, 52]. Event logging is a typical approach to collecting runtime data, where logs are therefore analysed to establish sequences of events or states [50]. Analyses of system behaviour can be supported by event logs [39]. However, analysis of logging data can be difficult and resource-intensive due to size-related issues. Structuring log files eases analysis of large logs [26]: by organising data (e.g., into typed columns), analysis can be simplified through sorting and filtering.

Authors' address: Owen Reynolds, 180200041@aston.ac.uk; Antonio García-Domínguez, a.garcia-dominguez@aston.ac.uk; Nelly Bencomo, n.bencomo@aston.ac.uk, SEA research group, EPS, Aston University, Birmingham, United Kingdom.

Current approaches to logging can be time consuming, as they are hard to build and refine [54]. Structured log files assist with analysis, but seem to offer little assistance when implementing logging. Part of the challenge with logging is knowing which events to log, and what content to use as a description [17]. Log files are sequential by nature, which presents additional challenges when trying to represent concurrent activities. Developers are left to overcome those problems by themselves, such as indicating time intervals or relationships between events.

A runtime model [7] provides an abstraction of the runtime system at a certain level, which discards details not relevant for its scope. Such a runtime model is available to the system itself to perform analysis [10]. The system is, therefore, self-aware of those aspects represented and abstracted by that runtime model [31]. By logging the changes to the model, rather than the changes to the system, we can abstract away details and therefore reduce the volume of the logs. Knowing the *provenance* of the changes [41], (i.e. who made those changes and for what reasons), creates the needed causal links that can help answer questions about the system behaviour. Such questions can be answered using *provenance graphs* (Section 2.3), which relate system *entities* to the *agents* who produced them, and the *activities* they were performing at the time of the change.

We argue that runtime models combined with provenance graphs can create a logging-based infrastructure that solves some of the present issues with structured logs. For each change, it will be possible to track who did it, and which activity caused it, within a concrete time interval. This approach to creation of logs can, therefore, be automated using a provenance metamodel as a base, with system descriptions derived from the runtime model. Keeping the structure of the provenance graph independent from that of the runtime models would allow for reusability. In this paper, we present *Cronista*, a system that captures the changes of a runtime model into a provenance graph, which can, later on, be queried to support explanation of causes for behaviour. We use *Cronista* on a multi-threaded system whose execution is based on runtime models.

This paper extends an early version of our work [42], making the following additional contributions:

- The system architecture has been refined, separating the implementation details of the storage of the history model into a new third type of component: a *history model store*.
- This capability has been leveraged to expand the traffic simulation case study in a new direction, studying the trade-offs between the CDO model repository and the JanusGraph graph database when storing the provenance graphs created by the *Cronista* curator. We compare their relative costs in time and space (using containers to measure costs in a more holistic manner), and their relative capabilities and ease of use for investigating issues and visualising the provenance graph.
- The discussion of the curator and observer has been updated and expanded with details about the design of the messages and tracking transient versions of model elements that are not reflected in the model storage.

This paper is structured as follows. Section 2 sets out the research background, outlining the concepts that underlie our research. Section 3 presents the architecture and design of *Cronista*. Section 4 applies the approach to a traffic controller, states the research questions and justifies the selection of technologies and their configurations. Section 5 presents the concrete results obtained using the case study for each research question Section 6 lists the internal and external threats to validity of the previous results. Related work is presented in Section 7. Finally, Section 8 presents the conclusions, and outlines the areas of future work.

2 BACKGROUND

The ideas underpinning the work emerge from several areas, which we present as background. As such, we introduce autonomous and self-aware systems, and how the limitations of event logging motivate our work. Relevant concepts from the areas of provenance, runtime models, model versioning, and model storage technologies are also discussed.

2.1 Autonomous and self-aware systems

Kephart and Chess presented their vision for *autonomic computing* back in 2001 [29], emphasising how software systems would become so large and complex that architects would not be able to fully anticipate all the interactions in advance. As such, many interactions and related concerns would need to be dealt with during execution. They presented a system architecture known as the MAPE-K loop (Monitor, Analyze, Plan, Execute that works over a Knowledge base) [3], where the system ran on top of models of its environment, its decisions, and their consequences, built over a feedback loop.

A system's ability to make decisions on it's own exacerbates the need for trust. This was already identified in [46], where the authors argue that the system must garner the confidence of its users and developers by explaining why the system acted in a particular way over time. Otherwise, the systems may not be adopted by users and general public [1].

Self-awareness implies the capture of the runtime system state in an explicit way, to therefore underpin decision-making for self-adaptation, and other self* properties. One way to represent this current state and underpin self-awareness it to use runtime models, which are presented in 2.4.

2.2 Event logging

A way to support explanations is to record or log the context of the time when a decision was made, i.e. record what the system was seeing, doing, and "thinking" each time it made a decision. Traditional logging frameworks (e.g. Log4j for Java [2]) can be used to produce a log of various events in the system. Log data may be used to identify system states or sequence of steps in a process for analysis after an event. This process has its own difficulties, as Yuan et al. [54] identified in a study of several high-profile open-source programs. Developers typically do not get their log messages on the first try: many have to be modified as afterthoughts, being changed in 18% of all revisions. 26% of those are related to the verbosity level, 27% are related to logging new variables, and 45% are about modifying the static text. With better tools, this time could have been saved and reinvested into the system itself.

Structured logging tries to produce better logs by making them easier to parse with other tools, and by providing more guidance on how to design the logged information. Legeza, Golubtsov, and Beyer briefly mention the use of JSON/XML for logs, and focus more on the guidance about their content [33]. They consider that a log message is divided into metadata (when, where, and its severity), and content (what happened, why, what's next, and additional details). Legeza et al. say that when/where can be automated, the severity needs to be manually picked, and the content itself must be manually crafted in an iterative process.

Legeza et al. also mention the difficulties in correlating logs from different microservices in the same system, suggesting the addition of unique request IDs which are passed along all data paths for this purpose. In general, relating events (e.g. the start and end of an activity) can be difficult, especially in a highly concurrent system.

2.3 Provenance

When tracking the reasons that motivated an autonomous system to make a decision, a principled approach should be followed as opposed to inserting log statements at the own discretion of the developer. We claim that the field of

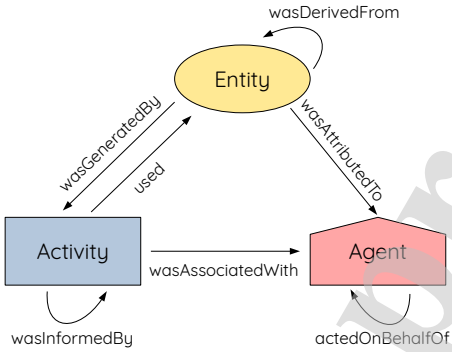


Fig. 1. W3C PROV provenance graph structure

data provenance can provide such principles. Provenance can be defined as "all the information and relationships that contributed to the existence of a piece of data" [41]. The Open Provenance Model (OPM) provides a standard reusable ontology to record provenance information [35]. OPM was the foundation for the W3C PROV provenance ontology, which is available in various notations: one of those is the PROV Data Model (PROV-DM) [23], which is used in our proposal (Section 3.2).

As shown in Figure 1, W3C PROV provenance graphs are formed by Agent nodes, Activity nodes, and Entity nodes. These nodes are connected with various kinds of edges that establish causal relationships, such as *"used"*, *"wasGeneratedBy"*, or *"wasDerivedFrom"*.

- **Agents** identify who is performing an activity (*"wasAssociatedWith"*): the exact level of granularity depends on the system. In a multi-threaded system, for example, each thread could be an agent. Delegation can be described as a relationship of trust between two agents (*"actedOnBehalfOf"*), which may occur as the result of a user interaction that results in a scheduled task. Agents generate entities through their activities (*"wasAttributedTo"*).
- **Activities** identify high-level tasks performed by the system, and can be nested at multiple levels to represent subtask relationships. Activities are related to entities through their inputs (*"used"*) and outputs (*"wasGeneratedBy"*). An activity using an entity generated by another activity *"wasInformedBy"* that activity. An activity takes place during a certain time interval.
- **Entities** represent system model features at the attribute and value level. When the value for an attribute changes, a new entity is created and related to its previous version (*"wasDerivedFrom"*). These entities could be grouped or categorised like sub-assemblies, which enable higher level of abstraction through a reduction in details.

Provenance can be automatically generated by a system at runtime [27]. We compare our approach with other automated provenance collection methods in Section 7.

2.4 Runtime models

Initially, models were used to mainly support the documentation, development, and deployment of systems. More recently, models have been used during the execution by the system itself, other systems or even humans [7]. Self-aware

systems need to have a way to reflect upon their own behaviour or goals and manipulate them to adapt as needed to meet their goal [12, 22]. To do this, they maintain a *runtime model* [10]: "a causally connected self-representation of the associated system that emphasises the structure, behaviour, or goals of the system from a problem space perspective". "Causally connected" means that a change in the runtime model will impact the system, just like a change in the system will be observable through the runtime model.

A recent survey from Bencomo, Götz and Song [7] identified and classified 275 papers on runtime models. Many of these papers (123) used runtime models to build self-adaptive systems. 41 papers used runtime models to assure certain non-functional properties in a system, and 32 used runtime models for self-optimisation and self-organisation. The survey reports that most runtime models operate at high levels of abstraction (specifically, 131 at the architecture level, and 32 at the goal level). However, there are still some runtime models at the process (12), context (20), and/or code (16) levels.

2.5 Versioned model storage with model repositories and graph databases

Model-driven software development (MDSD) [49] of a complex system involves working in a team, frequently over a large model. Such approaches to model development require people to work concurrently on model parts which can be versioned and merged into a single model.

Persisting models to flat text files is one way that a mature file-based Version Control System (VCS), such as Git [48], could be leveraged for tracking versions of a model to enable collaborative working. However, these tools compute changes on a per-line basis, which does not translate well to common file-based model interchange formats such as XMI [37]. Therefore, purpose-specific model repositories have been developed that persist large models to storage, with model versioning capabilities and collaborative working, such as the Eclipse Connected Data Objects project (CDO) [15].

Graph databases use nodes and edges to model data, unlike a relational database that uses tables; they can support a high number of concurrent users accessing a large volume of data. Daniel et al. [14] developed NeoEMF, a multi-database model persistence framework which included support for the Neo4j graph database management system (DBMS). Barmpis et al. [4] showed that large collections of file-based model fragments could be indexed into a single graph database for fast querying with their Hawk system.

More recently, the representation of the history of a model in a graph database has received attention. Haeusler et al. [25] observed a gap in the capabilities of graph databases, which were lacking versioning features, and presented ChronoSphere: a graph-based model repository with support for version control. Garcia-Dominguez et al. [18] showed an extension of the Hawk system that indexed evolving collections of file-based model fragments into temporal graphs, and provided a query language with the ability to navigate the history of the model.

3 *CRONISTA*: ARCHITECTURE AND DESIGN

The previous section discussed the challenges of traditional logging approaches when used for autonomous and self-aware systems. More principled approaches to collecting runtime data using provenance ontologies were considered, including the use of runtime models as a high-level abstract view of system state that could be used as a subject for logging. Combining provenance and the high-level abstractions of a runtime model can therefore, produce a more appropriate approach to logging for the case of autonomous and self-aware systems.

In this section, *Cronista*[1], an automated provenance-collector system for runtime models is presented. Its infrastructure is intended to be reusable across systems, and to be integrated with existing modelling frameworks to reduce implementation costs. This section first outlines the general architecture of *Cronista*, with its high-level components and their interactions. Next, the data representations produced and its features for storage management are discussed. Finally, the details of the operation of these components are provided.

3.1 Software architecture

Cronista is designed to observe systems that explicitly maintain abstractions of their internal states at runtime to underpin self-awareness [6, 31]. This internal representation of state is considered as a runtime model, which will be referred to as the *system model* [11]. Tracking changes to a runtime model has been studied in the literature, using approaches such as model versioning [24] or filmstrip models [28]. However, these approaches only capture the changes, and do not associate them with the cause or reason (i.e. provenance) of the change. Therefore, an explicit explanation for a change is not immediately available.

From an architectural point of view, the automated collection of provenance can be considered to be orthogonal to the functionality that the system needs to provide to its end users. Separating the provenance collection as much as possible from the core system functionality would make it easier to reuse efforts on provenance collection between systems. We propose to follow the component-based architecture shown in Figure 2, with a system whose runtime model is being monitored by *observers* installed in each of its concurrently running agents: their model access is being intercepted and notified to the *curator* through a message queue. The curator processes messages from the queue in order of arrival, using one of the available *history model stores* to record the information in a specific storage technology: this may be either a model repository, or a graph database. This separation across clear interfaces is designed to allow observers, curators, and history model stores to be deployed across separate machines, and to allow for alternative implementations (as in the case of the history model stores).

The observer components are at the edge of the architecture. They are responsible for the integration into the specific technology/language used and observe what the system is doing, sending messages about system activities and model access in the format expected by the curator to a queue. These components can be kept lightweight in memory, as they do not need to store significant amounts of history, e.g., the observer may be part of a low-powered Internet of Things (IoT) device with limited resources, or it may be in a server in a large data centre.

Curators take the messages sent by the observers, and interact with the history model storage API to maintain *Cronista*'s own models about the history of the system: these *history models* will be described in Section 3.2. Unlike the observers, curators do not need to tightly integrate with the system's underlying technology: they are only interested in taking the messages being sent into the queue and processing. The specific way to connect the curator and the observers would depend on the system being observed: if the observers and curator can live in the same machine (e.g. if they are running in a powerful machine at a data center), that connection could be a share memory queue to maximize throughput. If this is not possible (e.g. the observers are deployed on remote resource-constrained hardware), the observers could send messages over the network to a curator living in the data center. The basic functionality for the curator remains unchanged across technologies: messages are processed into provenance graph nodes that are stored into a history model.

[1] Cronista is Chronicler in Spanish. The term chronicler alludes to the writer who compiles and writes historical or current events, in the literary genre that receives the name of chronicle. In some cases, he held an official position whose role it was to perform such functions.

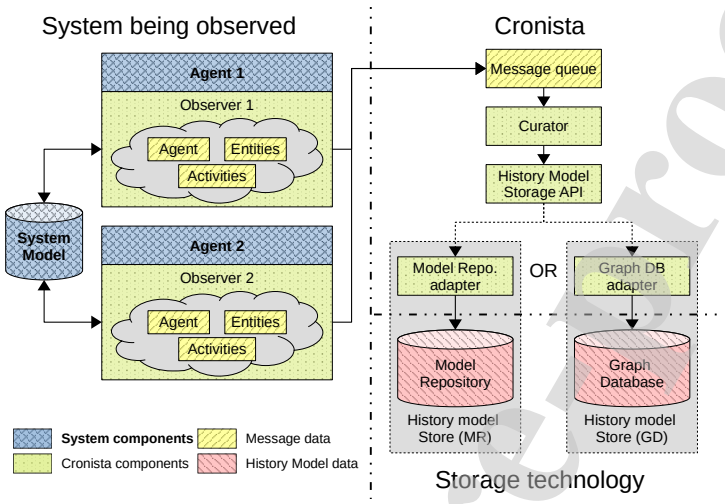


Fig. 2. Architecture of *Cronista*: top-level components and interconnections.

History model stores are the third and final type of component in the architecture of *Cronista*. They receive requests of the curator through a *History model storage API*, and map them to a specific storage technology (Figure 2). A history model store needs to handle the details about managing long histories of potentially very large models, while keeping disk usage and processing times under control. Multiple implementations of the history model store may be needed, based on varying needs across systems (e.g. differing throughput, model sizes, and retention periods). In addition, each specific storage technology will come with their own ecosystem of tools, such as query languages, data visualisations, or data protection.

3.2 Data representation

Figure 3 shows an outline of the history models that records the provenance data collected from a system at runtime. This model is created alongside a running system in a way that it is independent of the system and its resources. Creating a distinct separation between a system and history is the first step in managing the resource requirements for a history model, which may affect the monitored system.

The history model creates a series of time windows in chronological order to represent the passage of time. A time window can be seen as representing a block of time, which could range from seconds to days depending on the application. Each time window refers to a *base version* of the system model at the point in time the window started. The changes to the system model from this point in time are represented in an accompanying provenance graph. Therefore, a sequence of changes to a model can be recalled using provenance to recreate a system model state, for any point in time covered by a time window. Capturing a base model version breaks the dependency between time windows, as earlier time windows are not required to establish the state of a system model. Independent time windows make it

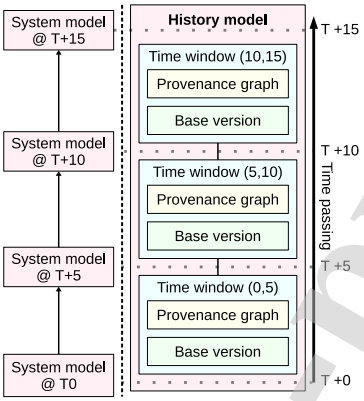


Fig. 3. History model

possible to simply "forget" time windows that are no longer relevant, enabling only a limited amount of history, e.g. a day or week, to be stored.

In each time window, the provenance graph represents more than the changes to the base model version taken at the start of the time window. The provenance graph also tracks the processes and responsibilities associated with the concrete change, with the aim to enable explanations. The main questions that users may ask and the relevant information to capture for each case are listed below:

- **Who made the change?** The system (an executing thread) or a human interaction could be responsible, therefore a graph should track the various "agents" participating in a system.
- **What was happening when the change was introduced?** A high-level overview of the processes in a system is needed that can associate the activities that cause a change. Describing a process at a high-level may be more useful than referencing a line of code that may not be available, or overly technical for the intended consumer of an explanation. Furthermore, there may be concurrent activities taking place that may have complex interactions.
- **Which parts of the system model informed the change?** Activities or process in a system make up a decision process which is guided by information in a system's model. Therefore, being able to recall the informing parts of a model will help in analysis and explanation of a decision.

A formal structure is needed for creating a provenance graph that integrates this information; a provenance ontology can be used. W3C PROV-DM (discussed in Section 2.3) is one such provenance ontology which has been established and validated as a standard. The provenance graphs in *Cronista* conform to a metamodel that is based on W3C PROV-DM.

3.3 Data collection

Based on the general architecture and the intended representation of the collected historical data, this section provides a more detailed description of how the observer and curator components operate in the current implementation of the architecture in Figure 2. It must be noted that the current implementation for the observer components assumes that the system to be observed is running in a single machine: its multiple agents are concurrent execution threads, which

Listing 1. Java try-with-resources for activity scopes

```
try (var aScope = new ActivityScope("ActivityName")) {
    // ... model reads and writes ...
}
```

operate on a system model shared across all threads. The technology hosting the shared model is expected to isolate concurrent model changes between the system threads, for example CDO, which supports ACID transactions. Likewise, the current version of the curator lives in the same system process as the observer: the curator receives the messages from the observer through a shared memory queue to avoid network overheads.

3.3.1 Observer. Observers intercept the atomic events such as the defined start/end of an activity and read/write operations on the system's runtime model during an activity. The observer produces messages that detail the agent, activity, and entities involved. All messages are inserted into a single thread-safe bounded blocking queue that is read by the curator. The rest of this section will outline how agents, activities, and entities are identified and described in the messages.

Agents are represented in the messages with a unique ID, each system thread is treated as an agent, and is tracked by a separate observer. The agent ID is based on its unique thread name. When a thread completes its execution, the observer is stopped, and no more messages are sent about that agent.

Activities are performed by an agent, have an identifiable name, and run from a start time until an end time. If an activity takes place repeatedly, each occurrence has its own unique ID. An agent is only capable of performing one activity at a time, but activities can be nested inside each other. A stack structure can be used to track the activities started by an agent, with the activity on top of the stack being the current focus of the agent, which is removed when the activity ends. The observer is responsible for sending "activity started" and "activity ended" messages to the curator when these events take place: the messages identify the activity that is taking place and the agent that is performing it.

A system developer has several options for denoting activities within a system. Annotation of activities can be an automatic or manual process, with the trade-offs discussed in Section 2.3. An activity describes the execution of one or more lines of code that perform actions via the system model. One approach might be to structure the code so the activity is represented by specially designated functions, using the stack trace to track nested activities. However, this approach may only be suitable for new systems being developed from scratch with provenance in mind: older systems may require expert analysis to understand the stack traces produced. A second approach is to use some form of code annotation, creating *activity scopes* or blocks within the code that enclose the work done by an activity. The specific syntax to define these scopes or blocks of code would depend on the language being used, and would require a developer to insert the start and end points for each activity into the code.

Listing 1 shows one way to implement these activity scopes for programs written in the Java language, taking advantage of its try-with-resource blocks which automatically allocate and free a resource when entering and leaving the block. In this case, the resource is an entry in the activity stack: the entry is added upon entering the block (activity started), and removed when leaving it (activity ended). This block-based approach supports the idea of nesting activities as well, through the nesting of try-with-resource blocks. The amount of code that a system developer wraps with a block determines the granularity of a defined activity. A broad activity description would cover many lines of code, providing a more abstract summary of many low-level operations. On the other hand, using several smaller blocks that cover a few lines would provide a more detailed description of what the system is doing.

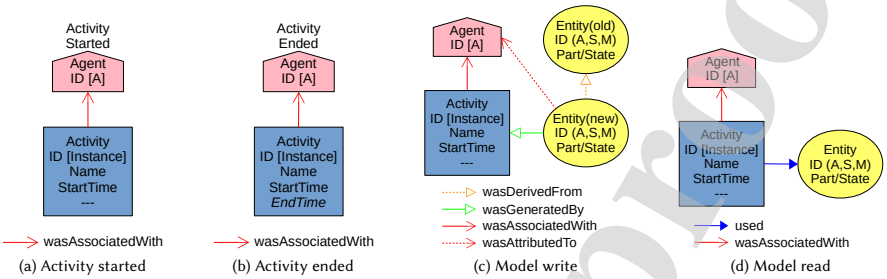


Fig. 4. Subgraphs produced by each type of message

Entities refer to the parts of the system runtime model being accessed (e.g. the value of a specific attribute of a specific model element). When such an entity is read, a "model read" message is sent: this message identifies the agent and the activity, and includes an *entity* description mentioning the part of the model that was accessed and the value that was read. Conversely, when a certain part of the model is modified, the observer sends a "model write" message that includes the agent and activity descriptions, and two entity descriptions: one before the change, and one after the change.

Providing consistent entity descriptions to the curator becomes complicated when concurrent activities are committing changes to a model repository. It is necessary to explicitly distinguish entities that are reflected in the persistent storage of the model repository (a *storage entity*), from entities that are only in the memory of a specific agent and have not been committed yet into the model repository (a *memory entity*). For that reason, the entities in the "model read" and "model write" messages are qualified by a triple (a, s, m): a identifies the agent whose memory space is being used, s is the storage revision of the model repository the entity belongs to, and m is number of times the entity has been changed in the agent's memory space since it was last committed to the model repository (0 for a storage entity).

The concrete details for how to instrument a model to produce atomic events for model accesses depend on the technology used to implement the system model. Ensuring manually that all model accesses report to the observer would be error-prone and time-consuming: instead, some form of code generation would be ideal. Code generation is already common in popular technologies such as the Eclipse Modelling Framework (EMF), where Java implementations of modelling concepts are generated from a high-level description of the model structure (its *metamodel*): this is the approach followed in the current implementation of *Cronista*.

3.3.2 Curator. The curator can process the messages from the observer in a stateless manner, as they contain all the information needed to create graph representations. As such the messages from an observer contain duplicate data, which provides some mitigation for potentially lost messages that would cause details in the graph to be lost. The duplicated data is removed by the curation process, which reuses existing graph nodes whenever possible in order to link together the results of processing each of the incoming messages. The messages include unique IDs that are used to find and create the nodes as needed. Figure 4 shows the resulting subgraphs from processing each of the four message types:

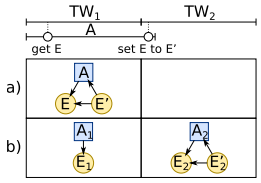


Fig. 5. Options for representing an activity A spanning time windows TW_1 and TW_2: a) keeping activities within the time window they started in, b) letting messages populate different time windows.

- For an `activity started` message, the curator will ensure the ACTIVITY node exists: this will be a new node as each occurrence of an activity has a unique ID. If this the first activity started by an agent, then an AGENT node will also be created. The AGENT and ACTIVITY nodes are related using *"wasAssociatedWith"*. The resulting subgraph is shown in Figure 4a.
- For `activity ended` messages, the curator will reuse the existing ACTIVITY node from the previous `activity started` message, and update the node to reflect its ending time. This would result in the subgraph in Figure 4b. Interestingly, even if the `activity started` message was somehow missed (e.g. due to a network failure), this would still result in the same subgraph being produced due to the stateless approach being followed.
- While processing `model write` messages, the curator will ensure the activity and agent are reflected on the graph, and then create ENTITY nodes to reflect the old and new entities. It is possible that the old ENTITY node may already exist, if it was accessed before: this will simply result in that node being reused. In such a case, this message would only require creating the ENTITY node for the new entity and linking it with the others. As shown in Figure 4c, the new entity will be considered to have been generated by the current activity, it will be attributed to the current agent, and it will be derived from the old entity.
- Processing a `model read` message results in the subgraph in Figure 4d, where the read entity is said to have been used by the relevant activity of the agent. In the most extreme case, where an entity was read by the same activity that created it (e.g. an activity setting a field and later retrieving it), all nodes would be reused and only a *"used"* relation may need to be added. In addition, it may be possible to infer a *"wasInformedBy"* relation between two activities if one activity used the entity that the other activity generated.

The reuse of existing nodes allows for capturing the information in a series of messages in a minimal number of nodes and relationships. However, this approach requires a considerable amount of searching for existing nodes with a matching ID. To keep message throughput high, a provenance graph could be built in memory, then committed to storage and removed from memory when a new time window is created. Alternatively, search indexes for nodes and IDs could be held in memory during a time window to reduce storage I/O for searches.

A provenance graph will become exceptionally large over prolonged periods of time, or when a busy system is rapidly creating messages. Therefore, the ability to dynamically unload and load history model data is important. The use of memory to construct a provenance graph is limited by the amount of system memory available. The time windows in the history models overcome this problem by dividing a run time into blocks of time. However, an activity that spans multiple time windows now presents a problem with how to distribute the provenance nodes between them.

Figure 5 presents two options for how provenance nodes could be arranged on multiple provenance graphs. In the example, an activity A spans both time windows TW_1 and TW_2. The value of the entity E is read in TW_1, and it is written

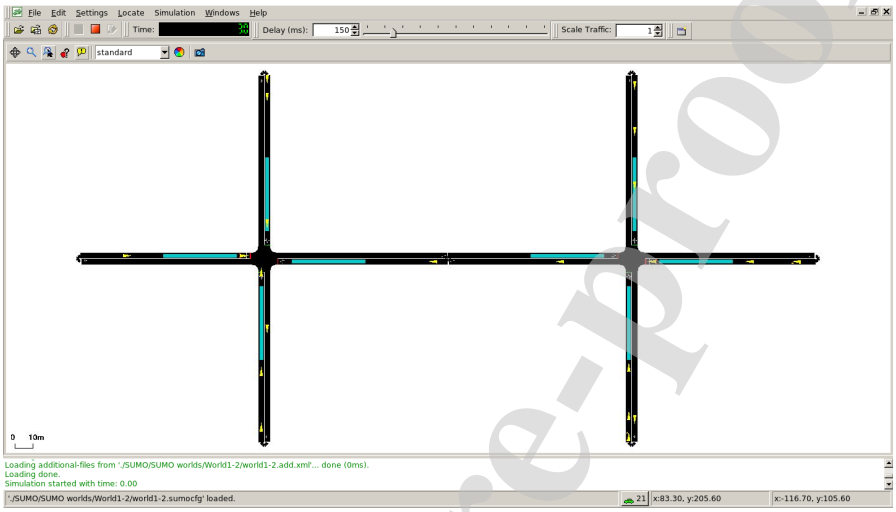


Fig. 6. Screenshot of the SUMO-based traffic simulation

as E' in TW_2. Option a) would have the curator process all the nodes into the first-time windows where an activity occurred. If an activity never finished (such as a continuous background task), it would not be possible to unload the time window, further complicating the curation process. Option b) creates node representations in the time window in which they occurred. The read operation in TW_1 is shown as A_1 and E_1. When processing the write operation shown in TW_2, a search of existing nodes for A and E fails, resulting in the creation of A_2, E_2, E'_2. There is an increase in storage costs for this approach, but it does remove complications for long running activities and permits unloading of passed time windows. The independence between time windows in this approach simplifies "forgetting" past history as each time window represents information for events in a given period.

4 CASE STUDY

The previous section described *Cronista*, our system for automated provenance collection for runtime models. This section will apply *Cronista* to an existing traffic simulation, which is controlled by several agents that coordinate through a shared system model. The rest of the section introduces the case study, sets out the research questions, explains the experimental process, and justifies our choices of technologies and their selected configurations.

4.1 Description of the case study

The system to be extended is a traffic simulation running on the open-source SUMO engine [21]. Figure 6 shows a partial view of the simulation at hand, which consists of two 4-way junctions; each managed by its own *junction controllers*. The controllers run concurrently, each using its own thread, and they share a connection to SUMO. The traffic lights in each intersection follow a cycle of *phases*: when a phase ends, the next one starts. The controllers can intervene to end phases earlier than the regular schedule. Each controller runs its own MAPE feedback loop:

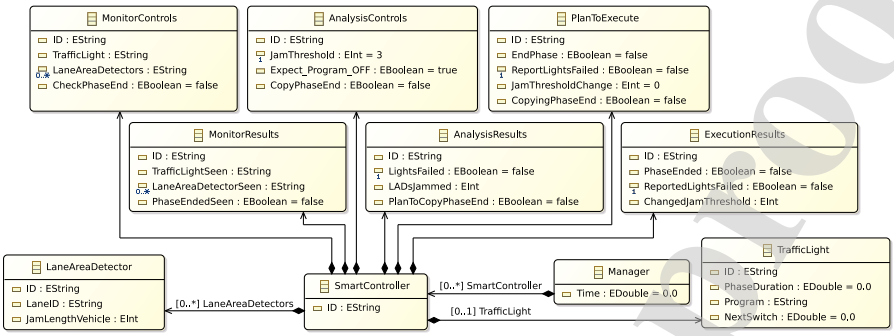


Fig. 7. Class diagram for the metamodel of the system model

- **Monitor**: reads the number of cars (yellow triangles) stopped at each lane area detector or LAD (blue rectangles). Reads the current state of the traffic lights, and checks the last plan that the other junction controller used (it may want to copy it). Records both pieces of information in the system model.
- **Analyze**: checks if traffic is "jammed" at one of the LADs, by comparing the number of cars against a threshold J (initially set to 3). Checks if the other controller ended a phase for its traffic lights in its last MAPE loop iteration. Records both pieces of information in the system model.
- **Plan**: based on the number of jammed LADs, it creates a plan for incrementing J by one (if more than 2 are jammed), or decrementing J by one (if less than 2 are jammed). If more than 2 LADs are jammed, or if the other junction controller ended a traffic light phase in its last iteration, then it creates a plan to end the current traffic light phase. Records current plans in the system model.
- **Execute**: conducts valid plans set out in Plan by communicating to SUMO and updating the system model. To protect against thrashing, Execute will block a phase change plan if the phase has been running for less than half of its duration.

A class diagram for the metamodel that the system model conforms to is shown in Figure 7. In general, a `Manager` program runs several concurrent `SmartControllers`. Each of the MAPE phases is represented by a type that collects its inputs (e.g. `MonitorControls`), and a type that collects its outputs (e.g. `MonitorResults`). There are also entities which represent the elements managed by each controller: the `TrafficLights` and the `LaneAreaDetectors`.

This simulation is a proof-of-concept of a simple traffic management scenario, yet it captures the basic elements of a more realistic simulation of city traffic. For the purposes of the case study, it shows a system which monitors the environment, analyses the situation, sets out plans to adjust itself, and attempts to execute those plans by interacting with the other participants. This creates information flows within the system that users and developers will want to follow through a provenance graph (e.g. to answer questions such as "why did the traffic lights end their phase at this point?"). It also has multiple concurrent agents interacting with the system model.

4.2 Research questions

A goal of this case study is to answer the following research questions about the proposed reusable provenance layer:

(RQ1) What are the costs involved in the use of the provenance layer, depending on the chosen storage implementation? This includes developer effort, additional processing time, and the use of system memory and disk space.
(RQ2) What advantages can be gained from the collected provenance information, depending on the chosen storage implementation? The chosen storage approach will provide its own querying and reporting tools, which may follow an imperative style mandating specific traversals, or a declarative style that only specifies the pattern of interest and leaves the traversal strategy up to the tool.

RQ1 will be evaluated by running the simulation without the reusable provenance layer, with the provenance layer using a model repository to store the provenance graph, and with the provenance layer using a graph database to store the provenance graph: space/time usage and the number of lines of code involved will be measured and compared. RQ2 will be evaluated by developing sample queries illustrating common use cases across both storage layers.

4.3 Storage technology selection

A model repository and graph database were selected from the available open-source projects. Open-source projects make their source code available, which is beneficial for transparency and evaluation purposes of research. They can also benefit from large communities both using and contributing to a project.

Model Repository, CDO [15] is a mature model repository, hosted by the Eclipse Foundation, who host open-source MDE projects. The development of CDO is active, with no feature restrictions, it also has native support for EMF, making it a familiar technology for the modelling community, which reduces the learning curve for *Cronista*.

Graph Database, As discussed in 2.5, there are many different graph databases with distinct characteristics, having the flexibility to change between them is desirable. TinkerPop is a vendor-agnostic API that enables supporting databases to be interchanged. JanusGraph is a TinkerPop-enabled graph database, that is mature, open-source and supports horizontal scaling with a variety of database backends; for these reasons JanusGraph was selected for the initial implementation.

An in-memory database configuration was used for JanusGraph, so that the curator could keep pace with the messages received from the observer. A series of short experiments, where 10 batches of 1000 vertices were created in the graph database in quick succession, confirmed different database configurations effect responsiveness to requests as seen in the literature [32]. However, the curator with a default CDO (disk-based) configuration had no problem in keeping pace with the messages receive rate.

5 RESULTS

This section presents the results of the case study for each of the research questions in Section 4.2. The costs involved are presented first, showing the metrics collected during the development of the system and its execution. This is followed by an account of the process to create provenance queries that were used to identify a fault in the traffic control system.

5.1 RQ1: costs involved

Developer costs. The integration of *Cronista* into the Java-based traffic controller (the *instrumented controller* or IC from now on) required manually adding 16 activity scopes in two levels. First, an activity scope encloses each of the MAPE phases. Next, each MAPE phase contained a further 3 nested activity scopes (e.g. Monitor had “monitor traffic light”, “monitor LAD”, and “monitor phase end from the other controller”). This required wrapping the relevant parts of

		200 ticks		1000 ticks		5000 ticks	
Metric	History model store	Mean	SD	Mean	SD	Mean	SD
IC time	No provenance	32.67s	0.22s	152.90s	0.24s	753.72s	0.16s
	CDO	33.31s	0.32s	153.83s	0.34s	765.53s	0.93s
	JanusGraph	40.87s	0.63s	155.89s	0.22s	758.98s	0.64s
IC memory	None	10.94MiB	0.03MiB	11.51MiB	0.08MiB	14.74MiB	0.35MiB
	CDO	18.30MiB	0.82MiB	55.20MiB	0.71MiB	227.65MiB	3.74MiB
	JanusGraph	17.85MiB	0.04MiB	36.33MiB	0.48MiB	127.44MiB	3.24MiB
HMS memory	CDO	232.63MiB	15.720MiB	380.32MiB	10.85MiB	559.85MiB	17.24MiB
	JanusGraph	870.28MiB	6.47MiB	900.64MiB	6.70MiB	986.99MiB	13.59MiB
HMS net I/O	CDO	1.85MB	0.38MB	10.69MB	0.29MB	53.58MB	0.82MB
	JanusGraph	10.12MB	0.10MB	53.20MB	0.96MB	262.40MB	6.52MB

Table 1. RQ1: means and standard deviations of execution times and maximum memory usage for the instrumented controller (IC) and maximum memory usage and network I/O for each history model store (HMS), over 10 simulations across 200/1000/5000 ticks.

the junction controller code in 16 try-with-resources blocks, as in Listing 1. Commits to CDO were modified manually so they would go through the observer for the current agent: this is needed to manage the distinction between memory and storage versions. The base class for the generated classes implementing the system metamodel was changed from the default `CDOObjectImpl` to our `LoggingCDOObjectImpl`. Finally, the simulation had to notify the curator about its completion, so it would safely shut down the history model store (HMS).

The simulation grew after these changes from 823 lines of Java code to 846, as measured by `sloccount` 2.26 [53] while ignoring generated code. These 23 lines of manually written code represent less than a 3% increase in code.

Runtime costs. During each experiment, four processes were running on the same machine. The IC and the SUMO 1.4.0 traffic simulation ran as standard applications. The system runtime model was managed by a CDO 4.12.0 Docker container[2]. The HMS was managed by a second Docker container, using the same CDO Docker image or the JanusGraph 0.5.3 Docker image[3].

The simulation was run 10 times over 200/1000/5000 ticks, using three different configurations: without *Cronista*, with *Cronista* using the CDO HMS, and with *Cronista* using the JanusGraph (TinkerPop) HMS. These experiments were done on an Ubuntu 20.04 system with a Linux 5.4.0 kernel, using Oracle JDK 11 for the IC. The system ran on a Thinkpad X1 Carbon laptop with an i7-6600U CPU (dual-core with hyperthreading), with 16GB of RAM and an SSD. The Java-based IC was run with an initial heap size of 256MiB, and a maximum heap size of 512MiB. Execution times and memory usage of the IC were measured from a background thread in the traffic controller application, up to the point where the simulation had completed, all messages from the observer had been processed by the curator, and the HMS had been shut down. Memory usage and network I/O of the HMSs were collected through the `docker stats` command while the experiment was running, to cover all processes running in each Docker container.

Table RQ1 provides statistics on the execution times for the IC, the memory usage for the IC and HMSs, and the network I/O of the HMSs. The mean IC time for JanusGraph on 200 ticks is nearly 10 seconds longer than the time without *Cronista*. This is mostly due to the fact that JanusGraph takes longer to process the first burst of messages from the observer: while the curator can provide enough throughput to keep up with the rest of the messages, processing this initial backlog requires spending additional time after the simulation has completed. On the other hand, the CDO

[2]https://gitlab.com/sea-aston/cdo-docker
[3]https://github.com/JanusGraph/janusgraph-docker.git

HMS is fast enough to finish processing all messages less than a second after the 200 ticks of the simulation ended. Regardless, this is not an issue for longer simulations JanusGraph speeds up after the first few thousand vertices have been added, and becomes comparable to CDO in insertion speed.

On the other hand, the memory usage of the IC does grow over time with the current implementation. We anticipate that this increase over time is due to implementation details of the curator component, which suggests that there is room for further optimisation. This also shows that if the IC needs to run in a memory-constrained environment, it may be wise to deploy the curator on a different machine and have the observer and curator communicate over a network connection. Further experimentation will be required on this aspect, but in any case it is clear that the JanusGraph HMS required less memory on the IC.

There are noticeable differences between the CDO and JanusGraph (TinkerPop) HMSs in terms of resource usage. It must be noted that the Docker-based measurements cover every layer of both options, and JanusGraph has a very different architecture to CDO. CDO is persisting the graph to disk, which allows it to unload unused parts of it, while JanusGraph in its in-memory configuration is always keeping the entire graph in system memory.

The disk space used by CDO increased in a linear manner, and disk usage figures stabilised across runs for longer histories. On average, each tick took about 35.88KB of disk space in the HMS for the longest runs of 5000 ticks. While this shows that a modern hard disk could potentially record the simulation for a long time, there would still be a need to prune old unwanted history at some point in order to limit disk usage.

5.2 RQ2: leveraging provenance

Provenance graphs can become very large and complex: a combination of several approaches is needed to extract information from them. In our previous work [43], graph visualisations were used to explore the information, but these quickly grew too large and complex to be of practical use. For that reason, this work focuses on query-based approaches to extract information. For example, the survey by Herschel [27] mentions searching by item/time/type of element tracked, navigation (e.g. by following relationships or changing granularity levels), and structured query languages.

The provenance graphs collected in the history model contain knowledge of how the system ran, and why it reached its current state. This information could be used by the system itself (enabling a degree of history-awareness about its operation, as mentioned in Section 2.4), or by the developers and users of the system. In this paper we will focus our attention on the use of the provenance graph by system administrators and developers to analyse the behaviour of the system. This analysis may be done while the system is running, or after the system has failed or has been stopped after an event of interest.

As a diagnostic aid, the snapshots and provenance graphs in the history model can be used to reconstruct the state of a system. The reconstructed system could be animated using the entity state changes and activities, thus enabling the moment of failure to be observed. Beyond the simple playback provided by a step-by-step recording of the execution of the system, a provenance graph can also show the information that a particular activity used to introduce a certain change in the system, or how a particular part of information was changed over time by the various processes in the system. In short, the history model allows developers and users to know *why something changed*, rather than just seeing a sequence of snapshots of its history.

As a concrete example, in the rest of this section we will show how to investigate the root cause of a defect that we have deliberately introduced in the traffic controllers of Section 4, and to check that it has been fixed. We will perform this task from the point of view of a new developer in *Smart Traffic Inc.*: on our first day, we are told to investigate an issue with a recent update to a smart junction controller. The report says that the traffic lights are changing too often.

Listing 2. Excerpt of query Q1: find information used when a phase was ended

```
// 1. When was PhaseEnded set to true?
var ePhaseEnded = Entity.all.select(ed |
  ed.attr() == 'PhaseEnded' and ed.bool() and ed.isWrite());
for (entJT in ePhaseEnded) {
  entJT.printLayer(0);
  // 2. Where did the EndPhase this activity used come from?
  for (entUsed1 in entJT.wasGeneratedBy()?.Used
      ?.selectOne(e | e.attr() == 'EndPhase')) {
    entUsed1.printLayer(1);
    // 3. What information was used when EndPhase was set?
    for (entUsed2 in entUsed1.wasGeneratedBy()?.Used) {
      entUsed2?.printLayer(2);
    }
    '\n=end='.println();
}}
```

Listing 3. Example of output of Q1 with faulty system

```
L0: PhaseEnded true > GenBy > executeEndPhase
  L1: EndPhase true > GenBy > planEndPhase
    L2: AnalysisResults AnalysisResults@OID268
    L2: PlanToExecute PlanToExecute@OID267
    L2: LADsJammed 5 > GenBy > analysisLADJammed
=end=
```

All the developer knows is that the junctions follow a feedback loop, and that they have a provenance graph. We will see how this investigation can be done through the capabilities of the CDO and TinkerPop HMSs.

5.2.1 Forensic analysis with CDO. Assuming the system had been configured to use the CDO HMS, a CDO model repository would be storing the provenance graph that we need to query: the next step is to run a few queries on it to get the information we need, and for that a query language is needed. In the present case study, CDO was being used with its default H2 backend (a relational database), so SQL could be used: unfortunately, this would not directly translate to other backends. Instead, we can write the query with any of the existing model query languages, such as OCL (supported out of the box by CDO), or the Epsilon Object Language (EOL) [30]. For the case study at hand, we chose to work with EOL, as it has recently gained support for transparent parallel execution [34]. Further, EOL can access models stored in CDO through the use of an Epsilon extension developed by one of the authors of this paper [19].

The developer would query the provenance graph to find the activity where the phase changed (`PhaseEnded` in `ExecutionResults` was set to true). Then, the developer would ask for the information used to make such a change, at several levels or *layers*. The developer would run the EOL query in Listing 2: the `printLayer` EOL context operation prints the name and value of the entity, and the name of the activity that generated it. The query would produce outputs such as those in Listing 3. The outputs show that `PhaseEnded` was set to true in the `executeEndPhase` activity, which was informed by `EndPhase`, which was set to true in `planEndPhase` after checking `LADsJammed`, which was set to 5 in `analysisLADJammed`. This is odd: `LADsJammed` should never be larger than 4, the number of LADs at each junction.

Listing 4. Excerpts of seeded fault located by Q1

```
// activity scope
try (var s = new ActivityScope("analysisLADJammed")) {
  analysisLADs(sc.getLaneAreaDetectors(),
  sc.getMonitorResults(), sc.getAnalysisControls(),
  sc.getAnalysisResults());
}
// method being called
public void analysisLADs(...) {
  //sysAR.setLADsJammed(0); // ← FAULT
  for (LaneAreaDetector sysLAD : sysLADs) {
    if (sysMR.getLADSeen().contains(sysLAD.getSumoID())) {
      if (sysLAD.getJamLength() > sysAC.getJamThreshold()) {
        sysAR.setLADsJammed(sysAR.getLADsJammed() + 1);
}}}}
```

Listing 5. Examples of output of Q1 with fixed system

```
L0: PhaseEnded true > GenBy > executeEndPhase
  L1: EndPhase true > GenBy > planEndPhase
    ...
    L2: LADsJammed 3 > GenBy > analysisLADJammed
=end=
L0: PhaseEnded true > GenBy > executeEndPhase
  L1: EndPhase true > GenBy > planEndPhase
    ...
    L2: LADsJammed 2 > GenBy > analysisLADJammed
    L2: PlanToCopyPhaseEnd true > GenBy > analysisPhaseEndEvent
=end=
```

The developer then knows that the problem is in the `analysisLADJammed` activity. At this point, the developer can look at the code (see Listing 4), to find out that someone inadvertently commented out an important line which resets the `LADsJammed` counter before recalculation. After fixing the query, the developer can then let the system run further, to then re-run the query and check that the phases are ending for the correct reasons, producing an output such as in Listing 5. The output shows valid cases when the phase should end, i.e. when `LADsJammed` is above the threshold (set at 2 by default), as in line 4, or when copying the behaviour of the other controller as in line 10.

To save space, in this example we had the developer write the query in Listing 2 all at once. However, in practice the developer would most likely follow a number of steps to iteratively build the query: i) look for the cases when `PhaseEnded` was set, ii) look for what informed those cases, and finally iii) focus on `EndPhase` at layer 1 and then look for what informed its value at the time. From Herschel's point of view, there are elements of search, step-wise navigation, and structured querying in the example. As such, we believe EOL is expressive enough to cover most scenarios, and the ability to extend types with context operations (such as `printLater` or `attr`, which are not part of the history metamodel) would make it feasible to create a reusable library of functions to cover various scenarios. This library would also abstract away some of the complexities of the provenance graph, e.g. the use of discrete time windows. It may be possible to package these queries themselves into a UI for domain experts that covers the most common cases. Still, EOL may not be the most concise language for certain queries: for instance, a user may just want

Listing 6. Example Gremlin query and output for counting how often a traffic light phase ended, as seen from the Gremlin console.

```
gremlin> g.V().
  has('Entity','AttributeName','PhaseEnded').
  has('AttributeValue','true').count()
==>234
```

Listing 7. Example Gremlin query showing the properties of the entities used by an activity that caused a traffic light phase to end.

```
g.V().
  has('Entity','AttributeName','PhaseEnded').has('AttributeValue','true').
  limit(1).
  out('WasGeneratedBy').out('Used').valueMap()
```

to see if there is a connection from a particular entity to a particular activity. For that case, the path expressions in graph query languages may be better suited.

5.2.2 Forensic analysis with Tinkerpop. If a TinkerPop HMS is used, then several options are available for querying the provenance graph. For this case study, the Gremlin language included in TinkerPop was used to run queries from the Gremlin console, which is based on the Groovy scripting language. Gremlin is available for other programming languages, such as Java, Scala, Python, or Rust, but the basic primitives are the same.

The general steps to investigate the issue in the TinkerPop HMS would be similar to those in the CDO HMS. First, the developer would write a query to check how many times the phase ended and see to what degree there were anomalies with their frequency. The query in Listing 6 searches the graph for a vertex of type `Entity`, whose property `AttributeName` has the value `PhaseEnded`, and whose property `AttributeValue` has the value `True`. Repeating this query while looking for `AttributeValue = False` and comparing the results shows a potential problem with the phase being ended more frequently than expected.

When exploring the graph in Gremlin, having convenient functions such as `valueMap()` that can display the properties on a vertex are helpful. Accessing the next layer of nodes in the provenance graph through a relationship can be done with `out()`. In combination, these can be used to explore the graph in a stepwise fashion, starting from a specific point of interest. Listing 7 shows an example of a query to explore the edges step by step. While exploring provenance information, accessing the entities used by an activity and then stepping into their related activities will likely be a common operation in any investigation. The simple and repeatable pattern of combining this two operations is friendly to new developers, and it is closely related to the navigation approaches mentioned by Herschel [27].

As the developer navigates through the provenance graph, the path expands from the effect and approaches the cause. The repetition of a few simple query patterns can be built up to reach the source of a fault, without expert knowledge of the query language. The query in Listing 8 was built up this way, tracing back the phase endings to readings of `LADsJammed` above 4, which should be impossible as there are only 4 lane detectors. This is the same problem that was highlighted by Listing 2, but with a much simpler query created step by step. Knowing that the issue is with the LADsJammed counter not being reset (as in Listing 4), the problem can be fixed.

Comparing the EOL query in Listing 2 and the Gremlin query in Listing 8, it is interesting to note that while they achieve the same goal of finding the erroneous data that caused the unwanted traffic light changes, they follow very different styles. Where the imperative nature of EOL required loops and conditional checks, the declarative and pattern-oriented nature of Gremlin simply required adding more steps to the traversal of the graph. While the author of

Listing 8. Example Gremlin query tracing the excessive count of traffic light phase endings back to its cause.

```
g.V().
  has('Entity','AttributeName','PhaseEnded').
  has('AttributeValue','true').
  out('WasGeneratedBy').out('Used').
  has('AttributeName','EndPhase').
  out('WasGeneratedBy').out('Used').
  has('AttributeName','LADsJammed').
  valueMap()
```

these queries was new to both EOL and Gremlin, writing the EOL queries required the assistance of an experienced user of EOL, whereas the Gremlin query did not require more than some cursory exploration of the Gremlin primitives. These impressions would need to be confirmed empirically by a broader study with external participants, which we consider to be a valuable line of future work.

As a last note, it is important to remark that with its broader acceptance by industrial practitioners, the graph database community has a wider arrange of options for visualisation. Within the TinkerPop ecosystem, one such graphical visualisation tool is Graphexp [44]. Using Graphexp to follow the relationships between highly connected vertices caused severe slowdown and clutter, requiring frequent restarts making it impractical for a full investigation. However, it can help when writing queries, by enabling a quick visualisation of vertices of interest and their neighbours.

6 THREATS TO VALIDITY

The threats to validity are considered against the classification provided by Feldt et al. [16].Internal validity focuses on how sure we can be that the treatment actually caused the outcome. External validity is concerned with whether we can generalise the results outside the scope of our study.

Internal. The simulations used to create the history model data in the experiments were run for up to 5000 ticks to obtain averages. Therefore, the metrics collected may not be representative of a simulation that ran for a longer period, which could expose unseen resource leaks or stresses that may cause a system to fail.

For RQ1, the direct comparison of resources required by JanusGraph and CDO is exacerbated by JanusGraph, as it needs an in-memory configuration to keep pace. In future work, alternative disk-based configurations will be tested to provide a more direct comparison of performance. However, the difference between disk and memory storage, did not significantly affect the findings in RQ2 which explored the collected provenance data.

External. CDO and JanusGraph were chosen due to their prominent status in the model repository and graph database communities, and their state-of-the-art selection of features and reported performance in the literature. Regardless, it is still true that only one model repository and one graph database have been tested. Other model repositories and graph databases could be explored, as well as alternative configurations and optimisations.

Cronista has only been evaluated with one application, using one specific type of runtime model in one domain. However, it has been designed to be reusable, with explicit separations between the system, the curator, the observer, and the HMS components, and with a metamodel of the history model that is independent of the metamodel of the system runtime models. Regardless, this reusability has not been tested across multiple applications, system runtime models and/or domains yet. Additional case studies would be needed. Further, these case studies would also help clarify

the throughput levels that need to be supported in various domains. In this case study, the curator kept pace with the observer, but a system with much higher event throughput could pose challenges.

The current version of *Cronista* is dependent on EMF and its `EObjects` for instrumentation, which limits Cronista to model-based systems that use EMF. Using a more generalised approach to instrumentation, like aspect-oriented programming, would overcome the limitations by encapsulating the instrumentation. However, relaxing the modelling tool requirement introduces the potential for systems with partial or no runtime models to be instrumented. *Cronista* is dependent on the type and quality of abstractions within a system. For instance, the provenance graph of a system that does not keep track of its own structure and/or goals in its runtime model will be less insightful, as it will not be possible to know why those have changed. In other cases, the runtime model may only be understandable to a system developer. In these cases a mapping to a more understandable form for other audiences would be required. Furthermore, if the system has no runtime model (i.e., it only keeps track of its internal state at an implementation level), such a runtime model would need to be introduced to have the system work with Cronista. A system must also be causally connected with its runtime model, such that changes in the runtime model like self-representations, cause changes to a system's state and vice-versa; to enable an accurate reflection of the system's evolution in the history model.

7 RELATED WORK

There are other approaches aimed at automating provenance collection. The survey by Herschel et al. on provenance [27] states that there are three groups of solutions for automated collection of provenance in general programming languages. The first group requires explicit annotations to be added to the code. The second group collects provenance without requiring changes (e.g. via static analysis). A third group combines both approaches, with the more abstract and easier to understand information from manual annotations acting as a summary and the transparently collected information as a detailed record that can be explored. Ideally, a provenance collection approach should use this third approach as we have done in this work.

The provenance collected by *Cronista* is application-level provenance similar to SPADE [20]. SPADE is an open-source provenance middleware with more flexible capabilities: it also runs as a system service in the background, capturing operating system-level provenance (e.g. open files and connections), or application-level provenance. For application-level provenance, SPADE allows developers to manually introduce provenance information in a dedicated DSL, or to use compiler instrumentation to automatically track the provenance of certain function calls (e.g. via LLVM compiler options). *Cronista*, is different as provenance of a runtime models is collected automatically; an approach we believe is unique.

Zhao et al. integrated SPADE with the distributed file system FusionFS to track changes in a distributed file system used by a high-performance computing (HPC) system [55]. They reported negligible overheads for coarse-grained provenance, when using a 32-node cluster filesystem. However, fine-grain provenance had a much higher overhead, on their single-node experiments, but did better on the 32-node configuration.

Pinheiro et al. used a graph database to store the provenance of data in bioinformatics experiments [40], opting for Neo4J and PROV-DM. Their choice of a graph database was based on the availability of existing query languages, visualisation tools and flexible data schema. The overhead of collecting provenance for the genomics workflows, added less than 1.5% of the total disk usage. *Cronista* 's provenance collection is more fine-grained, thus produces more data.

Beyond automated provenance collection, there are other approaches that capture historical information about the evolution of a system. Parra et al. [38] demonstrated an approach that creates a temporal graph from a system's structured logs, which a user can query. They also present a 4-level roadmap for introducing time-awareness for

self-adaptive systems. On their roadmap *Cronista* achieves level-1 (forensic explanations), with the potential for level-2 (live history-aware explanation). Level-2 would require the ability to perform rapid queries: the InTempo querying scheme by Sakizloglou et al. [45] could be one way to enable rapid history recalls from the *Cronista* HMSs.

8 CONCLUSIONS AND FUTURE WORK

This paper has presented *Cronista*, an approach for tracking the behaviour of self-adaptive and autonomous software systems that runs using runtime models, by automatically collecting the provenance of changes to those runtime models. Specifically, rather than recording everything that happens in the system, the approach abstracts away from details and records only the access to the higher-level system runtime model.

Cronista follows a modular architecture, with an explicit separation between its key components: a set of observers, a curator, and a history model store. The structure of the history model is independent of the structure of the system runtime model, allowing *Cronista* to be reusable across multiple systems without requiring changes to its code, as long as they are based on the same modelling technology.

Cronista supports multi-threaded systems through the use of multiple concurrent observers. The curator processes accesses in a stateless way while splitting them over time windows. The chronologically-ordered time windows represent a history model based on the system's execution, and can be deleted by users if not all the past history is desired: this is currently left to users as a manual step however, we plan to introduce automated history pruning (which would save resources by forgetting time windows) in future versions of *Cronista*.

In addition to providing updated and expanded descriptions of the observer and curator components, this paper has expanded the evaluation by comparing the trade-offs involved in storing the provenance graph in a model repository (CDO) and graph database (JanusGraph over the vendor-agnostic TinkerPop API). This comparison has been enabled by the improved architecture since our prior work [42]. The JanusGraph-based history model store introduced a more user-friendly query language with Gremlin. The latter and made it possible to leverage the graph analysis and visualisation tools in the TinkerPop community. On the other hand, it used considerably more resources than CDO and required more fine-tuning before it showed the throughput needed to keep up with the model access in our case study.

There are several areas for further investigation. In its current version, the queries require knowing how to write queries, and the answers can be technical in nature. Given that *Cronista* now supports the popular TinkerPop API, it may be possible to leverage results from the graph visualisation community on how to allow less technical users to investigate the proveanance graph. We also foresee that our approach can support autonomy of the system, as the running system can use its own provenance graph to perform self-diagnosis when its system model becomes invalid. For instance, a watchdog agent in the case study could have detected the invalid `LADJammed` value, and then it would have identified the responsible activity through a query, disabling it temporarily before warning a system operator.

The case study in this paper uses an architectural system model designed around the MAPE-K feedback loop, which is a popular type of runtime model, as shown in the survey by Bencomo et al [7]. Future case studies will explore the application of *Cronista* against larger systems, against systems with different runtime models such as goal-level or process-level models, and against systems with partial or no runtime models. To enable these case studies, additional system instrumentation techniques may be required such as aspect-oriented programming. These case studies would provide additional insights into how a system's internal abstractions can be leveraged, to gain insight into the systems such as how they evolve their goals and processes; as well as exposing more dependencies between *Cronista* and a system's internal abstractions.

On the infrastructure side, a future line of work is to revisit the need for discrete time windows. The use of disjoint provenance graphs for each time window can complicate the writing of queries: some of the context operations in Listing 2 required additional code to handle discontinuities between time windows. We plan to compare the use of *"continuesIn"* links between activities and entities that span windows with the use of a single *rolling provenance graph* that is automatically pruned by a background process. Either way, users may wish to protect certain activities or entities from pruning, or specify different retention periods. As an alternative approach to solving these complexities for users, a high-level provenance language could be created that abstracts away the time windows: this could avoid the challenges that pruning would introduce.

Finally, this work has considered only a system running on a single machine: for systems with higher numbers of event throughputs, or systems where the observer is running in a resource-constrained machine (e.g., an IoT device), it could be necessary to run the curator and observer in different machines. The curator and observer could communicate over a network, using a reliable messaging protocol such as MQTT [36] to handle message delivery complexities (e.g., redelivery). In a distributed system, different machines may have their own clocks: this lack of a global time may complicate the order of the incoming messages. To solve this, we will investigate approaches for event ordering in distributed systems such as the work on XVector by Beschansnickh et al. [8], which creates vector-timestamped logs for concurrent and distributed systems: specifically, XVector uses local vector-clocks on each node to establish the partial causal ordering of events. Having a single central curator could present a bottleneck for larger systems: in this case, distributing the work across multiple curators would be needed. One source of inspiration is the work on runtime megamodels by Vogel and Giese [51], where multiple local runtime models are integrated into a global runtime megamodel. Cronista would have one curator (and local provenance graph) per local runtime model, and a curator (and provenance graph) for the runtime megamodel. This would split the work across multiple levels of runtime models.

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] R. Andrews, J. Diederich, and A. Tickle. 1995. Survey and critique of techniques for extracting rules from trained artificial neural networks. *Knowledge-Based Systems* (1995). https://doi.org/10.1016/0950-7051(96)81920-4

[2] Apache Foundation. 2020. Log4J homepage. https://logging.apache.org/log4j/2.x/ Date last checked: February 14th, 2021.

[3] P. Arcaini, E. Riccobene, and P. Scandurra. 2015. Modeling and Analyzing MAPE-K Feedback Loops for Self-Adaptation. In *Proc. of SEAMS 2015*. 13–23. https://doi.org/10.1109/SEAMS.2015.10

[4] Konstantinos Barmpis, Antonio García-Domínguez, Alessandra Bagnato, and Antonin Abherve. 2020. Monitoring model analytics over large repositories with Hawk and MEASURE. In *Model Management and Analytics for Large Scale Systems*, Bedir Tekinerdogan, Önder Babur, Loek Cleophas, Mark van den Brand, and Mehmet Aksit (Eds.). Academic Press, 87–123. https://doi.org/10.1016/B978-0-12-816649-9.00014-4

[5] Victoria Bellotti and W. Keith Edwards. 2001. Intelligibility and Accountability: Human Considerations in Context-Aware Systems. *Human–Computer Interaction* 16 (2001), 193–212. https://doi.org/10.1207/S15327051HCI16234_05

[6] N. Bencomo and L. H. Garcia Paucar. 2019. RaM: Causally-Connected and Requirements-Aware Runtime Models using Bayesian Learning. In *Proc. of MODELS 2019*. ACM, 216–226. https://doi.org/10.1109/MODELS.2019.00005

[7] N. Bencomo, S. Götz, and H. Song. 2019. Models@run.time: a Guided Tour of the State-of-the-Art and Research Challenges. *Software and Systems Modeling* 18, 5 (2019), 3049–3082. https://doi.org/10.1007/s10270-018-00712-x Springer-Verlag.

[8] Ivan Beschastnikh, Perry Liu, Albert Xing, Patty Wang, Yuriy Brun, and Michael D. Ernst. 2020. Visualizing Distributed System Executions. *ACM Transactions on Software Engineering and Methodology* 29, 2 (Mar 2020), 1–38. https://doi.org/10/ggnfj8

[9] F. A. Bianchi, A. Margara, and M. Pezzè. 2018. A Survey of Recent Trends in Testing Concurrent Software Systems. *IEEE Transactions on Software Engineering* 44, 8 (2018), 747–783. https://doi.org/10.1109/TSE.2017.2707089

[10] G. Blair, N. Bencomo, and R. B. France. 2009. Models@ run.time. *Computer* 42, 10 (2009), 22–27. https://doi.org/10.1109/MC.2009.326

[11] G. S. Blair, N. Bencomo, and N. B. France. 2009. Models@run.time. *IEEE Computer* 42, 10 (2009), 22–27. https://doi.org/10.1109/MC.2009.326

[12] Javier Cámara, Kirstie L. Bellman, Jeffrey O. Kephart, Marco Autili, Nelly Bencomo, Ada Diaconescu, Holger Giese, Sebastian Götz, Paola Inverardi, Samuel Kounev, and Massimo Tivoli. 2017. *Self-aware Computing Systems: Related Concepts and Research Areas.* Springer International Publishing,

Cham, 17–49. https://doi.org/10.1007/978-3-319-47474-8_2

[13] National Research Council. 2014. *Complex Operational Decision Making in Networked Systems of Humans and Machines: A Multidisciplinary Approach.* National Academies Press, Washington DC. https://doi.org/10.17226/18844

[14] Gwendal Daniel, Gerson Sunyé, Amine Benelallam, Massimo Tisi, Yoann Vernageau, Abel Gómez, and Jordi Cabot. 2017. NeoEMF: A multi-database model persistence framework for very large models. *Science of Computer Programming* 149 (Dec. 2017), 9–14. https://doi.org/10.1016/j.scico.2017.08.002

[15] Eclipse Foundation. 2019. CDO Model Repository. https://www.eclipse.org/cdo/ Date last checked: February 14th, 2021.

[16] Robert Feldt and Ana Magazinius. 2010. Validity Threats in Empirical Software Engineering Research - An Initial Survey.. In *Proceedings of SEKE 2010* (San Francisco Bay, CA, USA).

[17] Q. Fu, J. Zhu, W. Hu, et al. 2014. Where do developers log? an empirical study on logging practices in industry. In *Companion Proceedings of ICSE 2014*. ACM, 24–33. https://doi.org/10.1145/2591062.2591175

[18] Antonio Garcia-Dominguez, Nelly Bencomo, Juan Marcelo Parra-Ullauri, and Luis Garcia-Paucar. 2019. Querying and annotating model histories with time-aware patterns. In *Proceedings of the ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems*. ACM, Munich, Germany, 194–204. https://doi.org/10.1109/MODELS.2019.000-2

[19] A. García-Domínguez and S. Madani. 2020. emc-cdo GitHub project. https://github.com/epsilonlabs/emc-cdo Date last checked: February 14th, 2021.

[20] A. Gehani and D. Tariq. 2012. SPADE: Support for Provenance Auditing in Distributed Environments. In *Middleware 2012 (LNCS)*. Springer, Berlin, Heidelberg, 101–120. https://doi.org/10.1007/978-3-642-35170-9_6

[21] German Aerospace Center. 2019. SUMO homepage. http://sumo.sourceforge.net/ Date last checked: February 14th, 2021.

[22] Holger Giese, Thomas Vogel, Ada Diaconescu, Sebastian Götz, Nelly Bencomo, Kurt Geihs, Samuel Kounev, and Kirstie L. Bellman. 2017. *State of the Art in Architectures for Self-aware Computing Systems.* Springer International Publishing, Cham, 237–275. https://doi.org/10.1007/978-3-319-47474-8_8

[23] P. Groth and L. Moreau. 2013. *PROV-Overview.* Working Group Note. W3C. https://www.w3.org/TR/prov-overview/ Date last checked: February 14th, 2021.

[24] M. Haeusler, T. Trojer, J. Kessler, et al. 2018. Combining Versioning and Metamodel Evolution in the ChronoSphere Model Repository. In *Proc. of SOFSEM 2018*. Edizioni della Normale, 153–167. https://doi.org/10.1007/978-3-319-73117-9_11

[25] Martin Haeusler, Thomas Trojer, Johannes Kessler, Matthias Farwick, Emmanuel Nowakowski, and Ruth Breu. 2019. ChronoSphere: a graph-based EMF model repository for IT landscape models. *Software and Systems Modeling* 18, 6 (Dec 2019), 3487–3526. https://doi.org/10.1007/s10270-019-00725-0

[26] S. He, J. Zhu, P. He, and M. R. Lyu. 2016. Experience Report: System Log Analysis for Anomaly Detection. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. 207–218. https://doi.org/10.1109/ISSRE.2016.21

[27] M. Herschel, R. Diestelkämper, and H. Ben Lahmar. 2017. A survey on provenance: What for? What form? What from? *The VLDB Journal* 26, 6 (2017), 881–906. https://doi.org/10.1007/s00778-017-0486-1

[28] 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

[29] J. O. Kephart and D. M. Chess. 2003. The vision of autonomic computing. *Computer* 36, 1 (Jan. 2003), 41–50. https://doi.org/10.1109/MC.2003.1160055

[30] D. S. Kolovos, R. F. Paige, and F. Polack. 2006. The Epsilon Object Language (EOL). In *Proceedings of ECMDA-FA 2006*. Bilbao, Spain. https://doi.org/10.1007/11787044_11

[31] S. Kounev, P. Lewis, K. Bellman, N. Bencomo, et al. 2017. *The Notion of Self-aware Computing.* Springer International Publishing, Cham, 3–16. https://doi.org/10.1007/978-3-319-47474-8_1

[32] Tibor Kovács, Gábor Simon, and Gergely Mezei. 2019. Benchmarking Graph Database Backends—What Works Well with Wikidata? *Acta Cybernetica* 24, 1 (May 2019), 43–60. https://doi.org/10.14232/actacyb.24.1.2019.5

[33] V. Legeza, A. Golubtsov, and B. Beyer. 2019. Structured Logging: Crafting Useful Message Content. *:login;* Summer 2019, Vol. 44, No. 2 (2019). https://www.usenix.org/publications/login/summer2019/legeza

[34] S. Madani, D. S. Kolovos, and R. F. Paige. 2019. Towards optimisation of model queries : A parallel execution approach. *Journal of Object Technology* (2019). https://doi.org/10.5381/JOT.2019.18.2.A3

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

[36] MQTT.org. 2021. MQTT homepage. https://mqtt.org/ Date last checked: June 18th, 2021. Archived at https://archive.is/PqhCX.

[37] Object Management Group (OMG). 2015. *XML Metadata Interchange (XMI) Specification.* Technical Report. https://www.omg.org/spec/XMI Date last checked: February 7th, 2021.

[38] J Parra-Ullauri, A. García-Domínguez, L. García-Paucar, and N. Bencomo. 2020. Temporal Models for History-Aware Explainability. In *12th System Analysis and Modelling Conference (SAM '20), October 19–20, 2020, Virtual Event, Canada.* https://doi.org/10.1145/3419804.3420276

[39] Anastasiia Pika, Moe T. Wynn, Colin J. Fidge, et al. 2014. An Extensible Framework for Analysing Resource Behaviour Using Event Logs. In *Advanced Information Systems Engineering*. Springer International Publishing, Cham, 564–579.

[40] R. Pinheiro, B. Aires, A. F. Araujo, M. Holanda, M. E. Walter, and S. Lifschitz. 2014. Storing provenance data of genome project workflows using graph database. In *2014 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. 16–22. https://doi.org/10.1109/BIBM.2014.6999292

[41] B. Pérez, J. Rubio, and C. Sáenz-Adán. 2018. A systematic review of provenance systems. *Knowledge and Information Systems* 57, 3 (Dec. 2018), 495–543. https://doi.org/10.1007/s10115-018-1164-3

[42] Owen Reynolds, Antonio García-Domínguez, and Nelly Bencomo. 2020. Towards automated provenance collection for runtime models to record system history. In *SAM '20: 12th System Analysis and Modelling Conference, Virtual Event, Canada, October 19-20, 2020*, Abdelouahed Gherbi, Wahab Hamou-Lhadj, and Ahmed Bali (Eds.). ACM, 12–21. https://doi.org/10.1145/3419804.3420262

[43] O. Reynolds, A. García-Domínguez, and N. Bencomo. 2020. Automated Provenance Graphs for models@run.time. In *ACM/IEEE MODELS 2020 Companion Proceedings*. https://doi.org/10.1145/3417990.3419503

[44] Benjamin Ricaud. 2021. Graphexp GitHub project. https://github.com/bricaud/graphexp Date last checked: February 7th, 2021.

[45] Lucas Sakizloglou, Sona Ghahremani, Matthias Barkowsky, and Holger Giese. 2020. A scalable querying scheme for memory-efficient runtime models with history. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS '20)*. Association for Computing Machinery, New York, NY, USA, 175–186. https://doi.org/10.1145/3365438.3410961

[46] P. Sawyer, N. Bencomo, J. Whittle, E. Letier, and A. Finkelstein. 2010. Requirements-Aware Systems: A Research Agenda for RE for Self-adaptive Systems. In *Proceedings of RE'10*. https://doi.org/10.1109/RE.2010.21

[47] Andrew D Selbst and Julia Powles. 2017. Meaningful information and the right to explanation. *International Data Privacy Law* 7, 4 (12 2017), 233–242. https://doi.org/10.1093/idpl/ipx022

[48] Software Freedom Conservancy. 2020. Git project homepage. https://git-scm.com/ Date last checked: February 14th, 2021.

[49] Thomas Stahl, Markus Voelter, and Krzysztof Czarnecki. 2006. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, Inc., Hoboken, NJ, USA.

[50] M. Szvetits and Uwe Zdun. 2013. Enhancing root cause analysis with runtime models and interactive visualizations. In *Proceedings of the 8th Workshop on Models@Run.time co-located with MODELS 2013*, Vol. 1079. CEUR-WS, 39–51.

[51] Thomas Vogel and Holger Giese. 2012. A language for feedback loops in self-adaptive systems: Executable runtime megamodels. *2012 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)* (Jun 2012), 129–138. https://doi.org/10.1109/SEAMS.2012.6224399 arXiv: 1805.08678.

[52] K. Welsh, N. Bencomo, P. Sawyer, and J. Whittle. 2014. *Self-Explanation in Adaptive Systems Based on Runtime Goal-Based Models*. 122–145. https://doi.org/10.1007/978-3-662-44871-7_5

[53] D. A. Wheeler. 2013. sloccount homepage. https://sourceforge.net/projects/sloccount/ Date last checked: February 14th, 2021.

[54] D. Yuan, S. Park, and Y. Zhou. 2012. Characterizing logging practices in open-source software. In *Proceedings of ICSE 2012*. IEEE Press, Zurich, Switzerland, 102–112. https://doi.org/10.1109/ICSE.2012.6227202

[55] D. Zhao, C. Shou, T. Maliky, and I. Raicu. 2013. Distributed data provenance for large-scale data-intensive computing. In *2013 IEEE International Conference on Cluster Computing (CLUSTER)*. 1–8. https://doi.org/10.1109/CLUSTER.2013.6702685

Credit Author Statement

Owen Reynolds: Conceptualization, Software, Investigation, Formal analysis, Writing - Original Draft, Visualization

Antonio García-Domínguez: Supervision, Conceptualization, Software, Investigation, Writing - Review & Editing

Nelly Bencomo: Supervision, Conceptualization, Writing - Review & Editing

Declaration of Interest Statement

Declaration of interests

☑ The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

☐The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: