Proceedings Paper:
Reynolds, Owen James, Garcia-Dominguez, Antonio orcid.org/0000-0002-4744-9150 and Bencomo, Nelly (2023) Automated Provenance Collection at Runtime as a Cross-Cutting Concern. In: Proceedings - 2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS-C 2023. 2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS-C 2023, 01-06 Oct 2023 Proceedings - 2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS-C 2023. Institute of Electrical and Electronics Engineers Inc., SWE, pp. 276-285.

https://doi.org/10.1109/MODELS-C59198.2023.00057

Automated provenance collection at runtime as a cross-cutting concern

Owen James Reynolds
SEA research group,
Aston University
Birmingham, UK
0000-0002-5639-0533

Antonio García-Domínguez
ASE research group,
University of York
York, UK
0000-0002-4744-9150

Nelly Bencomo
AIHS research group,
Durham University
Durham, UK
0000-0001-6895-1636

***Abstract*—Autonomous decision-making is increasingly applied to handle highly dynamic, uncertain environments: as incorrect decisions can cause serious harm to individuals or society, there is a need for accountability. For systems that use runtime models to represent their observations and decisions, one possible solution to understand their behaviour is to study the provenance of the changes to those runtime models. In this paper, we investigate how to use Aspect-Oriented Programming (AOP) to solve the cross-cutting concern of automated provenance collection through aspect-oriented programming, as a generalisation of previous work which could only operate on models created with the Eclipse Modeling Framework. We present a variety of patterns to identify the elements of the runtime model that are of interest for automated provenance collection, as well as the additional supporting infrastructure needed to cover the gaps left by the lack of a dedicated modelling framework. Two case studies have been conducted. The first study replicates past results with an EMF-based system, using AOP instead of customising the code generation, and compares runtime overheads and required developer effort. The other case study investigates the use of Cronista with AOP on a system using plain Java classes for its runtime model. The results show that the new AOP-based approach for automated provenance collection can effectively replace the old generator-based approach, while being applicable to a broader range of systems, at a cost of a small increase in runtime memory usage for EMF-based runtime models.**

***Index Terms*—Provenance, runtime models, aspect-oriented programming.**

I. Introduction

The right for explanation and algorithmic accountability for autonomous decisions are growing concerns for society, with the European Union General Data Protection Regulation (GDPR) being one example of a legal requirement for explainability [1]. However, the increasing uncertainty and complexity of software-based systems makes it more difficult to identify the causes for runtime behaviour or decisions [2]. Different kinds of uncertainty can be introduced, e.g. concurrent processes [3], or processing of partial information [4].

In our prior work, Cronista was developed to help explain the behaviour of systems that have runtime models [5] representing their observations and decisions, which are built with the Eclipse Modelling Framework (EMF) [6]. Cronista creates a history model that contains a graph describing the changes to the runtime model, following a provenance ontology. Previous work [7] identified several external threats to validity: Cronista had only been evaluated with one system, partly due to its dependency on the EMF code generator, and further case studies were needed with different applications and runtime models.

This paper revisits this design decision, comparing two approaches to deliver automated provenance collection across a system as a cross-cutting concern: the original approach using EMF's code generator, and a new approach using aspect-oriented programming (AOP) [8]. AOP has the advantage of being agnostic to the specific modeling framework being used, being also applicable to systems that have not used any modeling frameworks and only use plain objects for their runtime models.

The paper is presented as follows. Background concepts are provided in Section II. Section III discusses how to implement the Cronista observer components via AOP. In Section IV, the new AOP-based approach is evaluated across two case studies, with results presented in Section V. A discussion of the overall findings, threats to validity and related works is in Section VI. Finally, conclusions and lines of future work in Section VII.

II. Background

A. Provenance for Software Systems

Provenance is all the information that describes how something came to be. Provenance has proved to be helpful across a wide array of applications [9], with examples such as LogProv [10] applying provenance to big data to evaluate trustworthiness of data and processing pipelines, or the SPADE [11] open-source provenance collection middleware solution that collects application or operating system level provenance.

The survey by Herschel et al. [12] identified three general approaches for provenance capture: manual program annotation/instrumentation, automated static analysis, and a hybrid mix of the two. Our Cronista system, first presented in our previous work [7], is a hybrid: using *activity scopes* as coarse-grained provenance of activities, complemented by its automated collection of runtime model accesses/changes from the system. These runtime models contain information about the system's environment and state, used to enable self-adaptation [13]. Cronista tracks the provenance of changes to a runtime model which serves as a high-level abstraction of the system.

The work was partially funded by the EPSRC Research Project Twenty20Insight (Grant EP/T017627/1).

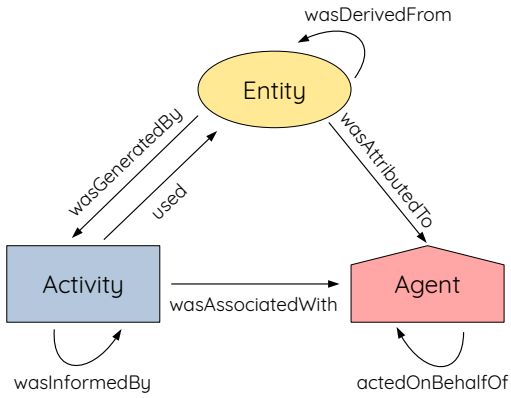

Figure 1: W3C PROV-DM

Working at a higher level of abstraction allows for tackling complexity while using provenance. The collected provenance allows for exploring causal connections and discovering how a system's state came to be. Cronista uses W3C PROV [14] to structure its provenance graph, using the graph nodes and edges shown in Figure 1:

- **Activities** represent a piece of work done by the system that is relevant to the problem domain, with a name that is meaningful to a domain expert. In Cronista, these are executions of a certain code region (i.e. a Java try-with-resources block). Activities can be nested within each other: Cronista keeps automatically track of this information through a stack. Entries are pushed when the activity starts, and popped when it ends (whether successfully or not). The top of the stack always represents the current activity.
- **Agents** are the actors responsible for executing an activity. In Cronista, each Java thread is an agent of its own.
- **Entities** are the objects from the system's runtime model that are being accessed and modified by the agents within the activities. In Cronista, these can be EMF objects, or plain Java objects (depending on the implementation chosen by the developer).

Cronista aims to reduce the cost of maintaining provenance graphs to provide explanatory capabilities for new or existing systems. Cronista follows a modular architecture to provide a reusable set of components that can be extended or adapted to various domains or systems (See Figure 2). The current prototype of Cronista is available under the Eclipse Public License from its GitLab repository[1]. This section presents the high-level architecture of Cronista. Other sections shown later discuss how the interaction with runtime models in Cronista was redesigned to broaden its support to a wider range of systems, by eliminating the dependency on EMF.

As shown in Figure 2, the Observer of concurrent *Agent*s responds to events detected by instrumenting the system, detecting access and modification of its *Entities* within its *Activities*. The Observer sends messages about the events to

[1]https://gitlab.com/sea-aston/cronista

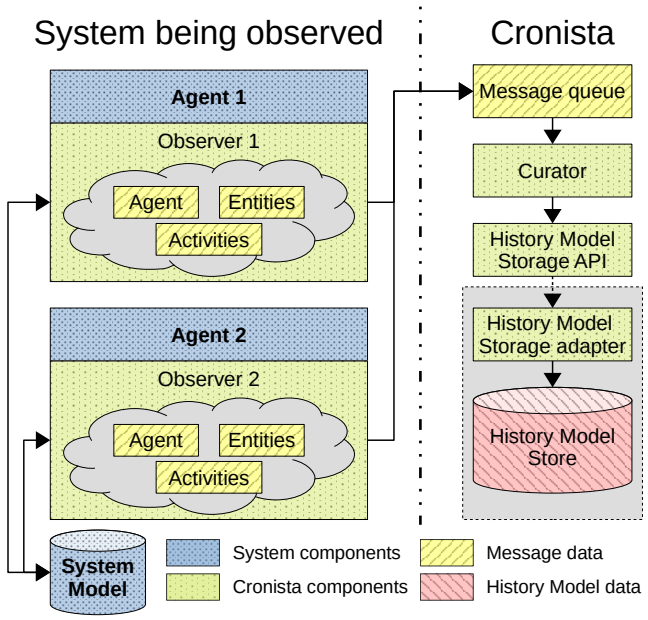

Figure 2: Cronista's System Architecture

a *message queue*: each message is a complete description of which agent accessed which model element during a particular activity. These messages are queued and processed in order of arrival, by a *Curator* that creates a provenance graph representing the information in the messages.

To support long-running systems the Curator maintains the collected data using a *history model*. This model is composed of time windows, containing a snapshot of the system model and a provenance graph of changes to the model. Time windows have no data dependencies between them, which enables them to be deleted (i.e. forgotten) for storage management. Cronista can use various storage technologies for the *history model store* (HMS) through adapter modules: currently supporting Eclipse CDO model repositories [15] and Apache TinkerPop-compatible graph databases [16]. These technologies have existing query languages such as Epsilon Object Language [17] and Gremlin [18], which enable exploration of the provenance graphs in the HMS.

B. Addressing Cross-Cutting Concerns via Aspects

Cronista's Observer components require adding code in many places (e.g. across a system's runtime model and its manipulation), which is a clear case of a *cross-cutting concern*, as defined by [19]: *a feature or (non-)functional requirement whose implementation is scattered over more than one module in a system; which may result in the tangling of code*. Kiczales et al. [8] identify cross-cutting concerns as *aspects* which are difficult to cleanly capture in code, and propose Aspect-Oriented Programming (AOP) techniques to compose and isolate reusable aspect code. Filman and Friedman [20] consider that a good AOP system minimises the degree to which programmers change their behaviours to benefit from AOP.

Mcheick et al. [21] applied AOP to a Chess game without modifying the original code. In this study, they attempt to implement logging, add new movements for pieces, and add a GUI. Their logging is simpler than Cronista's provenance graphs, as they insert console logging statements and trace all method calls from all classes. Our approach to collecting provenance would require more generalised AOP instructions to reflect the runtime model abstractions with a provenance ontology, instead of handcrafted log descriptions. Schaler et al. [22] also consider provenance collection to be a cross-cutting concern, and equally solve it using AOP techniques, but for a different kind of system: database systems.

III. Observation via Aspect-Orientation

This section will discuss implementing Cronista's Observation components using an AOP approach, implemented in Java using the mature AspectJ framework. The exact pointcuts may differ between systems, but general object-oriented design practice suggests common reusable patterns.

To explore the challenges of removing the modeling framework, three successive approaches will be explored, each relying less and less on EMF and the structure of the code that it generates. The first approach uses an aspect for EMF-based models, which replicates the approach in Section II-A by injecting code using AspectJ. Then, a second implementation relies on accessor and mutator methods, common in object-oriented programming and generated by EMF. The third implementation targets object fields without relying on EMF nor on getter/setter methods. However, not using a modeling framework introduces challenges, due to a lack of facilities.

The approaches require annotating the model classes and adding an `instrument` package to the system. This `instrument` package includes the `Inspector` class, and a collection of AspectJ *aspects* made up of *pointcuts* and *advice*. Pointcuts identify specific system events (e.g. method calls, known as *join points* in AspectJ) where certain code (the advice) should run: the Observer instrumentation.

A. AOP for Provenance Collection over EMF Models

The first approach replaces the code generator for inserting the instrumentation code for provenance collection into an EMF model, producing the same provenance as the approach in Section II-A. This approach uses the conventions of the code generated by EMF reducing the need for manual annotation.

Defining the pointcuts seems simple, as the EMF code generator uses a consistent naming style for the accessor methods (`get*()` and `is*()`) and the mutator methods (`set*()`). Unfortunately, this approach has a potential problem, as other methods in the model classes (some of which may have been manually written) may not call these operations and instead directly manipulate the underlying storage of the class. Instead, it is best to use the EMF *feature delegation* capabilities to consistently capture EMF model feature accesses: this is a common approach used in alternative EMF storage layers (e.g. CDO or NeoEMF). When the EMF generator model is told to use *dynamic* feature delegation, it produces model accessor (`eDynamicGet`) and mutator (`eDynamicSet`) methods for the model features that are clearly identifiable for pointcuts.

In large systems, there may be more than one EMF model, and we only want to instrument the runtime model itself. Cronista provides the `ObserveModelEMFeDynamic` annotation to identify which classes are to be instrumented. Developers need to add this annotation to the classes generated by EMF, either manually[2] or with EMF dynamic templates.

```
pointcut pcGet() : call(* eDynamicGet(..))                          1
  && @within(ObserveModelEMFeDynamic);                              2
pointcut pcSet() : call(* eDynamicSet(..))                          3
  && @within(ObserveModelEMFeDynamic);                              4
```

Listing 1: EMF dynamic feature delegation pointcuts

Based on these findings, it is possible to create a set of robust pointcuts as shown in Listing 1. These pointcuts target the internal `eDynamicSet` and `eDynamicGet` method calls (using `call`) that are done from within classes having the above `ObserveModelEMFeDynamic` annotation (using `@within`). This is similar to the prior EMF code generator-based approach in Cronista, where the customized `EObject` root class overrode those same `eDynamicGet` and `eDynamicSet` methods. This similarity means that the new approach works at the same abstraction layer, and that the advice should run at the same times and produce the same model provenance data.

```
after() returning (Object r): pcGet() {                             1
  if (observe) {                                                    2
    observe = false;                                                3
    try { inspector.inspectEMFeDynamic(/*thisModelAccess*/); }      4
    catch (Exception e) { e.printStackTrace(); }                    5
    finally { observe = true; } } }                                 6
before() : pcSet() { if(observe) { /* ... logic as above ... */ } } 7
after() : pcSet() { if(observe) { /* ... logic as above ... */ } }  8
```

Listing 2: Excerpt of AspectJ advice for provenance collection from a system model created using EMF

The pointcuts need advice code that will run after a model read (to intercept the returned value), and before/after a model write (to intercept the old/new value). Listing 2 shows the advice that will be weaved at those points in the system. In addition to the invocation of the `Inspector` method with the appropriate arguments, the advice uses an internal `observe` flag to prevent recursive inspection calls, which can occur as a result of the inspection process.

Finally, an `Inspector` class provides methods (reusable across runtime models using the same persistence framework, e.g. CDO) to collect all necessary provenance information for model accesses. Some details are easily determined based on when and where advice is applied, such as model access type and time. However, information such as the model part name, type and value will require reflection, which can be challenging with complex classes like `CDOObjectImpl`. In particular, the `Inspector` is responsible for generating an identifier for the part of the model that was accessed (e.g. using CDO IDs,

[2]EMF code generator preserves the Annotations between code generations.

```
1 pointcut getBean(Object t) : target (t)
2   && execution(* get*()) && @within(ObserveModel);
3 pointcut setBean(Object t, Object a) : target(t) && args (a)
4   && execution (* set*(..)) && @within(ObserveModel);
```

Listing 3: JavaBeans getter/setter pointcuts

```
1 pointcut getAttr(Object r) : get(* *)
2   && (@within(ObserveMethod) || @withincode(ObserveMethod))
3   && (@annotation(ObserveModel) || @target(ObserveModel))
4   && !@annotation(HideMe);
5 pointcut setAttr(Object a) : set(* *)
6   && (@within(ObserveMethod) || @withincode(ObserveMethod))
7   && (@annotation(ObserveModel) || @target(ObserveModel))
8   && !@annotation(HideMe);
```

Listing 4: Java object field read/write pointcuts

ID model features, or Java object IDs as a last resort), and maintain version numbers for the various features in a model element. These complexities are discussed in Section III-D.

B. AOP for Provenance Collection over JavaBeans

The previous section shows that a consistent structure is essential for automating provenance collection with AOP. Modelling tools such as EMF provide code generators capable of producing model implementations with a consistent structure, which are also clearly separated from the logic manipulating them. However, there are other approaches to creating structured code: for example, a well-organised software development team may follow certain coding guidelines or integrate various frameworks: the JavaBeans API is one example.

The JavaBeans API specification [23] seeks to create reusable software components by prescribing a common approach to structure objects. As such, the specification requires the use of accessor (*getter*) and mutator (*setter*) methods for JavaBeans properties. This use of get/set methods for a property is comparable to that seen in EMF model code for model features: its consistent method naming approach (e.g. `getMyModelFeatureName()`) can be used to extract the name of the feature, and AspectJ can retrieve the returned value (for a getter) or the new value to be set (for a setter). Other implementation details for the inspector would rely on the object following the JavaBeans convention correctly, such that the Observer receives accurate data for model changes. Listing 3 shows example AspectJ pointcuts for intercepting the execution of the get/set methods of the JavaBeans classes annotated with *@ObserveModel*.

C. AOP for Provenance Collection over Object Fields

In some Java programs, there is no consistent separation between the runtime model and the processes manipulating this runtime model, and some of the information is not made visible through a consistent interface (i.e. via getters/setters named according to some convention). As an alternative, AspectJ can be instructed to intercept reads and writes on object fields, using pointcuts such as those in Listing 4.

The relevant classes or fields implementing the runtime model are annotated with *@ObserveModel*, and the methods whose interactions with the runtime model need to be captured in the provenance graph are annotated with *@ObserveMethod*. Not all model accesses require provenance collection: for example, a method that saves the runtime model to a file may not require provenance collection.

If the class is annotated with *@ObserveModel*, all fields will be considered relevant for provenance: individual fields can be excluded by annotating them with *@HideMe* (in a "blacklisting" approach, where everything is included by default). As an example, a field in a class may be used for an expensive computation whose provenance is not required, but all other fields are. If only a few fields are desired, users can annotate only those fields with *@ObserveModel* instead of annotating the whole class (in a "whitelisting" approach, where nothing is included by default).

D. Object-Entity Correlation

In an observed system, multiple objects in memory may relate to the same model element, e.g. if different agents manipulate different copies of the model. In order to create accurate provenance links, Cronista must be able to correlate the access of a certain property/method in an object (e.g. "the name field of the Person object at memory position M was read") to the identity of a specific entity in the provenance graph (e.g. "the name of Person #1 was read"). This "object-entity correlation" from the (object, feature) pair intercepted by the observer to the (model element, feature) pair that constitutes the entity can take various forms, if using a modeling framework (or not), and how the modeling framework is configured.

Modeling frameworks such as EMF provide several ways of identifying model elements, e.g. through a domain-independent identifier set at the moment of creation (*xmi:id* or CDO object identifiers), through the location of the model element in the containment tree (e.g. "second child of the root object"), or through a domain-specific identifier feature declared in the metamodel (an "EID attribute" in EMF). These approaches have the advantage of being persistent between system executions. In addition, *xmi:id*, CDO object identifiers, and EID attributes remain the same even if the object is moved across the containment tree of the model.

Plain Java objects do not have persistent unique IDs such as those in CDO objects. As an approximation, Cronista uses the concatenation of the Java object's hashcode and the name of the field to produce an *EntityID*. Within the lifetime of a Java object, the *hashCode()* of an object will produce a consistent identifier which in most practical situations will be different for different objects (though it is not guaranteed to be unique). Serializing this object will not always preserve the *hashCode()* upon deserialisation, however: different executions of the system will produce disjoint provenance graphs if relying on *hashCode()*, which may or may not be acceptable to the user. The reason for the graphs for different executions being disjoint would be that the same conceptual model representations would not have the same identifier across those executions. For example,

Object A on the first execution has *ID-123*, and would have *ID-234* on the second execution: thus, they would be considered to be different objects. Solving the issue essentially requires some form of persistent identifier such as those provided by EMF/CDO.

E. Entity Versioning

The various activities in the system may read and write to different versions of the model: it is important to clearly identify which version was accessed. This is another aspect where the underlying infrastructure is important: some model repositories, such as CDO, provide model versioning. However, CDO only provides model element-level versioning, and the version number is only incremented upon a commit: between commits, the provenance entities of that element (its features) could be updated multiple times. Therefore, additional field level versioning is required. For CDO-based models, Cronista maintains a (storage version, in-memory version) pair for each provenance entity: the storage version is controlled by CDO, and the in-memory version is set to 0 upon a CDO commit, and is incremented upon modification.

In-memory versioning requires each thread to take its own copy of the model from storage. Cronista tracks changes for each thread's in-memory version, and the system is responsible for resolving conflicts when the in-memory models are committed back to storage. System models stored in CDO would require the second thread (agent) to merge or abort a conflicting commit, to keep the CDO model consistent: the Cronista provenance graph would show the agents, activities and entities involved in that conflict.

Since plain Java objects lack versioning, Cronista provides a fallback implementation based on their *EntityID* and the Observer instrumentation that detects updates. Cronista keeps a list of *EntityIDs* for plain Java objects, with their storage versions kept at 0 and their in-memory versions being incremented on every update. This simple approximation is sufficient to track the provenance of each execution of the single-thread plain Java system used in the second case study in Section IV-A: for more complex and long-running systems, adopting a versioned model store with thread-safety features and persistent object IDs would be recommended.

IV. Evaluation

This section describes the experimental approach used to evaluate the effectiveness of integrating Cronista with an AOP-based observer as described in Section III.

A. Systems under study

The first system, *TrafficControl* (TC) featured in previous Cronista case studies [7]. This system can autonomously adapt its runtime behaviour, using an EMF-based runtime model to control the traffic lights on a junction in a simulation. The original version used the approach in Section II-A to produce explanations based on its provenance graph. This system has been selected to show that the new AOP-based approach in Section III-A collects the same provenance information.

The second system, *AI-Checkers* (AIC) is a Java-based AI (a classic alpha-beta tree search) to play checkers [24]. AIC was not built with a modelling tool: plain Java objects are used to model the game state (e.g. game board). These objects can be represented using an ENTITY. Similarly, ACTIVITIES could be used to represent the processes an AI player performs in a turn, such as move planning.

B. Research questions

Our goal is to enable Cronista to be used with systems which have not been built with EMF, with minimal use of resources and maintaining the quality of the provenance graph. The more general approach of AOP should have comparable memory and time costs, and produce provenance graphs that match those which would be produced by the EMF method, and not impose undue demands on the developer. The evaluation of the new method seeks to answer the following questions:

- ***RQ1:*** *When an explicit EMF-based runtime model is used by the system, can the aspect-based observer effectively replace the old generator-based approach?*
- ***RQ2:*** *When the system does not use a modeling framework for its runtime model, how much developer effort is required to apply Cronista, and what are the runtime overheads imposed by Cronista?*

RQ1: A direct answer to RQ1 can be provided by developing new versions of TC that keeps the original activity scopes and replaces the generator-based observer with the new AOP-based observer. Given the potentially high cost of graph isomorphisms for even moderately sized graphs, it was decided to use a scenario-based equivalence test: the same fault as in [7] was introduced, and the same queries on the provenance graph were conducted to find the root cause. Two configurations of the AOP-based observer were used: one which collected the same information as the original EMF-based observer, and one which only collected a partial provenance graph with only the information needed for that specific query. Partial provenance collection from the runtime model offers a potential reduction in runtime overheads, in exchange for a loss of provenance information. Finally, an unmodified version of TC was run to set a baseline. These versions were labelled as follows: **NoProv** (code and model unchanged, the resource baseline), **CodeGen** (code and full model changed using code-generator), **AspFull** (code and full model changed using AOP), and **AspPart** (code and part of the model changed using AOP).

The provenance graphs produced by each TC version were verified using a Gremlin graph query. This graph query checks for a known graph pattern (shown in Figure 3) which can identify the injected fault in the system. Thus, each modified TC system was run with and without the injected fault, to confirm provenance collection is working.

The query used to verify the provenance graphs traces the cause of the system ending the current traffic light phase (Figure 3). Such a query can be used to diagnose frequent phase ending behaviours. The query starts from an entity representing an execution result of a `PhaseEnded` being true; then traversing edges and nodes back to a counter for the

number of Lane Area Detectors (LADs) reporting a *jammed* condition. A *jammed LAD* condition is counted when a queue of cars in a LAD at a junction exceeds the perceived acceptable number of waiting cars queuing. When the number of LADs reporting *jammed* exceeds 2, the system ends the phase in an attempt to clear the *jammed LADs*.

In all cases, TC was run while measuring system and query execution times, memory consumption, and network input/output (as measured through a JanusGraph Docker container). Experiments were run under Debian 10, Linux kernel 4.19.0-17-amd64, OpenJDK 11.0.12 (default Java settings), using a computer with an AMD Phenom II X4 970 CPU (3.5Ghz), 16GB RAM, and SSD storage. A background thread measured execution times and memory usage until the simulation completed, the curator processed all messages from the observer, and HMS shut down.

```
1  private static Game game;
2  private static Move move;
3  public static void main(String[] args) { ...
4      if (player1iscomputer) {
5          try (var scope1 = new ActivityScope("Player 1 plans"))
6          { move = cpu1.alphaBeta(game); }
7          try (var scope1 = new ActivityScope("Player 1 moves"))
8          { game.applyMove(move, game.board); }
9          move.printMove(); } ... }
```

Listing 5: AIC: Activity Scopes for Player 1's turn

RQ2: This research question is evaluated by applying Cronista to a system whose runtime models are implemented as plain Java objects: AI-Checkers [24] (AIC). AIC has a simple design and a runtime model close to the problem domain, based on a representation of the game board, and the creator of AIC did not have provenance collection in mind when designing the system. Without consulting the original developer, integrating Cronista required understanding the intended actions of the system code and finding the right level of granularity for the activities. The codebase required some refactoring to expose information to the AOP-based observer due to limitations in what AspectJ can intercept.

The general application of Cronista remains similar to that for RQ1: i) integrate the observer into the system, ii) delimit the high-level activities through activity scopes, iii) delimit model parts for tracking, and iv) use the provenance graph to answer queries to feed explanations. Steps ii), iii) and iv) are iterated to incrementally increase the amount of provenance collected. After each iteration step iv) is performed to check the quality of the provenance graph and to look for missing nodes and relationships. GraphExp [25] can be used to visually inspect the provenance graph for missing relationships. However, the Gremlin console [16] will be needed to access graphs which become too complex for visualisation.

Our prior experience of developing Cronista guides the approach of applying provenance collection: starting with a coarse-grained provenance collection, and then progressing towards a more fine-grained approach. For example, relationships between ENTITIES and ACTIVITIES can be checked manually by a developer, to ensure the information being passed between ACTIVITIES is being correctly represented. As the complexity/size of the system's code grows, so does the provenance graph representing the execution. Therefore, like writing code, an incremental process is needed to manage complexity. The steps used to create a provenance-enabled version of AIC are presented below.

a) First iteration: Captured the game as a sequence of high-level activities for each player. Method calls for the move planning and movement were located in the system's main method and annotated with activity scopes, covering both AI players (Listing 5). When defining the activity scopes, there was a need to turn local variables such as `move` into object fields, so they could be intercepted by AspectJ. In addition, variables passing information between two activity scopes need to be exposed as features of a model element to create ENTITIES connecting ACTIVITIES. Furthermore, a variable `game` containing a representation of the game board also had to be turned into a field.

b) Second iteration: The next iteration sought to expose some information from the planning process. However, as mentioned previously, the heuristic search process could result in excessive amounts of provenance data. Therefore, an activity BESTMOVESELECTION was marked to extract the high-level concepts as entities, such as BESTMOVE and BESTMOVEVALUE. The BESTMOVESELECTION activity occurs for each depth of possible moves searched in the limited time available.

c) Third iteration: The system model is missing concepts that describe the *number of equally best scoring moves* found and the *chosen move*. The existing code collected the equally best scoring moves in an array; then, based on the size of the array, generated a random number to choose a move from the array. Thus, the number of moves was hidden from Cronista as a property of the array, as was the randomly chosen index. Creating fields for both enables Cronista to observe the values and generate entity representations on the provenance graph.

d) Changes for reproducibility: AIC was modified in several ways to obtain reproducible performance results during the experiments. A parameter was added to control the random seed, to ensure the same best move would be selected during the multiple executions needed to control for system variability. The time limits for searching the best moves were also removed, leaving only the depth limit: this prevented differences due to minor timing changes produced by I/O and system processes. These changes ensure that replaying a game with the same seed will produce a similar provenance graph with consistent resource usage, to reduce noise in the resource usage measurements.

e) Query invocation: Once these modifications had been done, the provenance graphs of AIC were used to answer provenance related queries. Figure 4 shows one such query that can be used to trace the played moves back to the decision process that selected it from the results of a search that found a number of equally good moves. Additionally, metrics were collected at runtime to evaluate the overheads caused by Cronista when compared to the original versions of AIC. The same approach as in RQ1 was used to take the memory, time,

Metric	Version	Mean	SD
TC time	NoProv	754.93s	0.25s
	CodeGen	757.59s	1.59s
	AspFull	757.20s	0.46s
	AspPart	757.00s	0.42s
TC memory	NoProv	14.98MiB	0.18MiB
	CodeGen	37.44MiB	0.89MiB
	AspFull	48.97MiB	2.75MiB
	AspPart	47.55MiB	2.67MiB
HMS memory	NoProv	756.29MiB	6.73MiB
	CodeGen	928.44MiB	11.78MiB
	AspFull	941.96MiB	10.56MiB
	AspPart	926.11MiB	12.65MiB
HMS net IO	NoProv	0.01MB	0.00MB
	CodeGen	101.40MB	12.62MB
	AspFull	150.78MB	13.23MB
	AspPart	110.06MB	6.21MB

Table I: Means and standard deviations of execution times, maximum memory usages, and network I/O for TC, over 10 simulations across 5000 ticks.

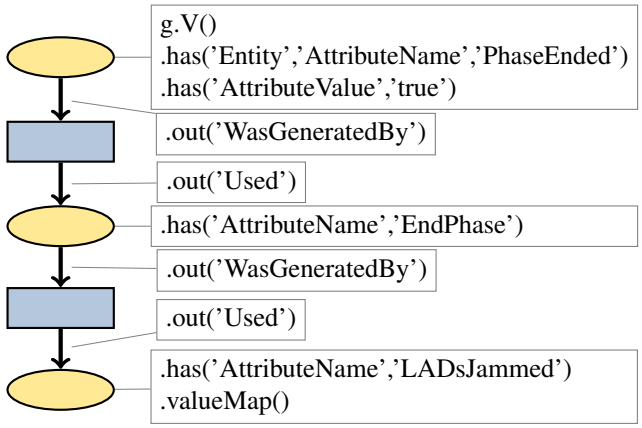

Figure 3: Gremlin query, find the cause of the PhaseEnded

and disk usage measurements. Five different seeds were used, and each seed was run 10 times. The experiments used the same system as in Section IV-B.

V. Results

Results of the experiments outlined in Section IV-B.

The first system, for space reasons Table I shows the metrics for the longest experiment run for TC, which was run for progressively longer simulations (ticks). With the provenance collection enabled, the simulation takes longer to run (from 754.93s to 757.59s on average, using the code generation approach). Similarly, the amount of maximum memory consumed also increases when using provenance collection (from an average of 14.98MiB on NoProv to 37.44MiB on CodeGen). However, CodeGen, AspFull and AspPart consume slightly different amounts of memory (Table I). CodeGen uses less

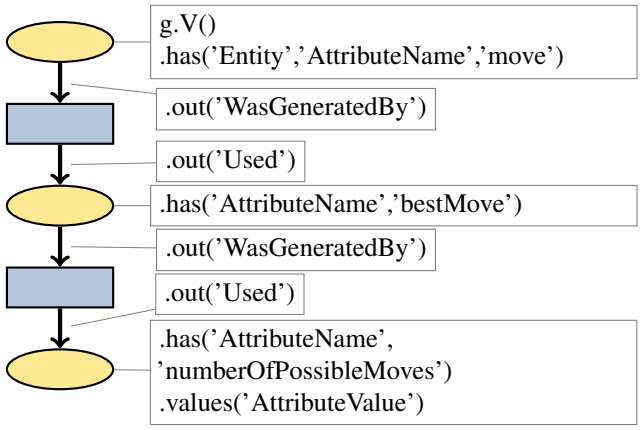

Figure 4: Gremlin query, find number of best possible moves

than both AspPart and AspFull, and AspPart is showing a slight reduction over AspFull. Finally, the execution times in Table I show that all the provenance collection approaches take slightly longer than NoProv (NoProv averaged 754.93s TC time, CodeGen averaged the longest TC time 757.59s): part of this comes from the processing and shutdown of the curation process, which lags behind the traffic control processes slightly.

The Gremlin query used to verify the provenance collected from CodeGen, AspFull and AspPart all returned the expected subgraph (Figure 3), confirming the provenance graphs were valid and contained the required data. Analysis of the query results for each TC with/without the fault correctly returned the `LADsJammed` count for each `PhaseEnded` occurrence. For example, queries on a faulty TC presented `LADsJammed` counts above 4, which exceeds the number of LADs connected to the system. Similarly, a working system did not have counts above 4, and the system also did not end phases as often.

The Second system, Table II shows the metrics collected from the AIC system, which played 5 seeded games 10 times each. The AIC times with no provenance collection (NoProv) show that the seeded games are ending at the same time with very little deviation (between 0.03s and 0.27s). Therefore, the changes to the system to have the AI players make deterministic decisions are working as intended. The time differences seen between NoProv and the aspect-based provenance collection (AspFull) is the added overhead of Cronista's Observer components. Table II shows a small increase in the total execution time of approximately 3.5s on average has occurred; this is similar to the results for TC.

The metrics in Table II under NoProv conditions show that memory usage did not significantly change with the game duration for different game seeds. AspFull shows Cronista's Observer components increased the system memory usage. This increase depended on the game duration: the shortest game (Seed1) used the least memory. With and without provenance, the deviation in memory usage of AIC was negligible.

Table II also contains the metrics from the history model store. The NoProv instance shows the resource usage of an unused HMS, setting a baseline. For AspFull, memory and

		Seed1		Seed2		Seed3		Seed4		Seed5	
Metric	Version	Mean	SD	Mean	SD	Mean	SD	Mean	SD	Mean	SD
AIC time (s)	NoProv	103.20	0.01	149.52	0.02	382.80	0.43	421.83s	0.02	230.61	0.01
	AspFull	106.91	0.06	152.94	0.04	386.75	0.27	425.29	0.10	234.02	0.03
AIC memory (MiB)	NoProv	1.54	0.00	1.55	0.00	1.55	0.00	1.55	0.00	1.55	0.00
	AspFull	11.90	0.00	12.31	0.00	15.18	0.00	15.11	0.00	13.12	0.00
HMS memory (MiB)	NoProv	752.98	15.32	753.77	12.00	747.79	15.00	751.57	13.41	745.45	13.41
	AspFull	768.54	4.71	806.77	24.74	886.22	7.40	885.64	6.01	876.78	6.01
HMS net IO (MB)	NoProv	0.01	0.00	0.01	0.00	0.01	0.00	0.01	0.00	0.01	0.00
	AspFull	4.61	0.00	6.53	0.00	17.58	0.00	17.41	0.05	9.58	0.00

Table II: Means and standard deviations of metric per Table1 for AIC, over 5 Game seeds each run 10 times. As AIC was not EMF-based, there is no code generation-based version to compare against.

network I/O increase as expected in longer running games. Some spread can be seen in memory usage (with standard deviations of 24.74MiB for an 806.77MiB graph), however, the baseline NoProv shows up to 15.32MiB of SD can occur even with an unused graph. Network I/O shows little to no change when replaying the same seed, which implies the Observer is reporting similar data for each replay.

It is possible to use AspectJ with annotations to collect provenance data from a system that uses POJO. Figure 4 shows a Gremlin query which accesses the provenance graph produced from AIC to retrieve ENTITIES and ACTIVITIES that connect a `Move` back to the `NumberOfPossibleMoves` it was selected from. Thus the number of alternative moves the AI Player found can be causally connected to the move played. As a developer, this query tells us how many times the decision process ended with a random selection from a collection of equally scoring best moves. As such we might use the information to further refine the decision making process, by extending the search time or adding tie breaking strategies.

VI. DISCUSSION

A. Overall findings

Deployment of Cronista's Observer components can be automated using AOP and annotations to enable provenance collection of changes to a system's runtime model. A system's runtime model does not need to be an EMF model; AOP allows for using plain Java objects as the runtime model. However, Java objects lack many features provided by a modelling framework, such as persistent object identities. Cronista provides an initial set of tools to overcome these challenges, and a more complex system may implement custom solutions for these problems.

Enabling Cronista with AOP opens the potential for provenance collection from systems that use other Java-based modeling frameworks. Furthermore, these frameworks might include solutions for some missing Java object features that make them harder to track. The framework could be told to automatically generate Cronista annotations for specific elements, which could then be targeted via pointcuts.

The history model that Cronista creates reflects the changes made to a system's runtime model and, as such, depends on the quality of the system's model. Therefore, modelling tools that encourage creating descriptive and human-understandable models are more likely to produce a history model than can explain a certain behaviour. For example, in the AIC case study, the number of moves found was not explicitly defined in the runtime model and had to be added. On the other hand, TC had a model explicitly designed around the feedback loop and business domain, with the intent to produce provenance information. Future work could investigate and quantify how modelling best practices impact the effort needed to add explanatory capabilities to a system, and the quality of the resulting provenance graphs.

Cronista imposes some time and memory overheads onto the observed system, as all logging frameworks do. Regardless, the observed overheads have been manageable, adding a few (~3.5s) seconds to total runtime as seen in Table I and Table II, most likely caused by Cronista's history model store starting up. AOP adds a baseline memory overhead which is a slight increase over the CodeGen approach (~10MiB as seen on Table I); against a small application like AIC this can seem disproportionate (Table II). However, while the memory usage for data collection does not rise significantly over time, ENTITY tracking does consume some memory to reduce history model lookups at runtime. In Table I, **AspPart** only offered a minor saving of resources in this instance compared to **AspFull**: this approach may need to be re-evaluated within larger systems. Further optimisations could further reduce these, such as removing the start-up time by hosting the history model store on a separate machine.

In terms of developer effort, AOP allows developers to use Cronista by only adding a few annotations to their models. The same aspects can be used across models of the same modelling technology (e.g. CDO models). The aspects can be packaged in a library to facilitate reuse, achieving some implementation efficiencies across multiple systems.

B. Threats to validity

This section will discuss the threats to the validity of the results of this paper, by using the classification by Feldt et al. [26]. Specifically, it will discuss internal validity (whether

the treatment caused the outcome), and external validity (how well the results could be generalised).

1) Internal validity: Integrating provenance collection into the AIC case-study was dependant on correct interpretation of the codebase and the developers' original intentions. As such, the quality and detail of the provenance could be improved with a deeper knowledge of the system. The provenance graph query was chosen based on what the authors of the paper would consider to be useful, rather than having been chosen by the AIC developer.

The graph database used was memory-based to maximise throughput in Cronista with minimal tuning. However, a larger deployment would likely need a disk-backed graph database and performance tuning to hold the massive graphs that would be produced by large and busy systems. As such, a limitation with the studies is that our simulations run for limited periods of time (just over 12 minutes for TC): there is future work in evaluating the metrics for larger systems, with higher throughputs and longer executions.

2) External validity: The case studies have not involved the potential users of the collected provenance. Such a case study would be better performed when an end user-friendly application has been created to visualise and explore the provenance. While our current tools to access provenance might inspire a provenance exploration tool, in its current form the technical skills (i.e. writing a Gremlin query) are likely too demanding for typical non-technical end users.

Cronista has been tested against two different systems, where different design approaches have been taken. These systems were developed by different people, so we have some confidence Cronista can extract meaningful provenance from systems. However, further evaluation with more systems would be desirable. Cronista is sensitive to the design style of the system, and the coding conventions that have been used. Less structured systems may require refactoring to make their internal models understandable.

C. Related work

The survey by Herschel et al. [12] identified three general approaches for provenance capture: manual program annotation/instrumentation, automated static analysis, and a hybrid mix of the two. Cronista is a hybrid: activity scopes provide a coarse-grained provenance of activities, complemented by its automated collection of model accesses from the system.

SPADE [11] is an open-source provenance collection middleware solution, which can collect application or operating system level provenance. While SPADE targets function calls within an application/operating system, Cronista specifically targets the runtime model of a system.

Schaler et al. [22] also consider provenance collection to be a cross-cutting concern, and equally solve it using AOP techniques, but for a different kind of system: database systems. In their work they do not consider models or runtime models as the subject for provenance collection, instead requiring the user to define the *points of interest* for which provenance is to be collected.

VII. Conclusions and future work

This paper demonstrated using aspect-oriented programming (AOP) to automate the deployment of Cronista's instrumentation without using the code generator of a modelling framework. This allowed Cronista to work with systems using plain Java objects, and potentially with other Java-based modelling frameworks. The results in Section V showed that an AOP approach can collect the same provenance information as a code generator-based approach. Section III discussed the parallels between the original EMF instrumentation, the new AOP-based instrumentation for EMF models, and the AOP-based instrumentation for plain Java objects. Given the positive results, it can be concluded that the collection of the provenance of the changes to a runtime model can be packaged as a reusable cross-cutting concern.

Modelling frameworks such as EMF provide solutions for most provenance collection challenges that appear when using models built with plain Java objects, which lack the persistent object IDs and versioning required to track their changes over time. However, modelling frameworks can also lack the versioning of model features at sufficient granularity (i.e. fields are not versioned). These gaps can be either covered by the basic implementations provided by Cronista, or by integrating an existing solution (e.g. CDO provides object-level IDs and versioning). Additionally, a system designed with a modelling mindset and tools is likely to create more meaningful system model abstractions and thus more meaningful provenance. Therefore, modelling tools and techniques seem to improve a system's history model: they provide robust model implementations and encourage descriptive system model designs to help create more explainable history models.

Adding a logging solution to a system will likely increase the runtime resources and impose additional overheads. However, the results indicate that Cronista has a manageable overhead for the systems under study. In addition, Cronista did not significantly affect the target systems' performance or functionality in both case studies. Given the modular architecture of Cronista, future work could seek to separate the most resource-intensive parts of Cronista (curation and history model storage) to a different machine from the monitored system.

Regarding the developer effort required to integrate Cronista, the proposed approach has been designed as a collection of components that can be reused across systems by being packaged as Java libraries: these include the activity scopes, observer, curator, and history model store. A system developer can reuse the aspects in the AOP-based instrumentation (also available from its GitLab public repository) by applying the Cronista annotations on the target system's runtime model classes. Validating its reusability in further experiments with other developers and systems is also future work.

Finally, the presented paper focuses on collecting provenance information, and querying it with a developer-oriented tool (Gremlin queries). Non-technical system users would need a more approachable user experience, that allows for exploration and avoids information overload. In the future, we plan to

investigate the design of end user-facing interfaces for exploring Cronista history models, e.g. as a timeline of activities or evolving states, with explorable casual relationships.

References

[1] A. D. Selbst and J. Powles, "Meaningful information and the right to explanation," *International Data Privacy Law*, vol. 7, no. 4, p. 233–242, Nov 2017.

[2] N. R. Council, *Complex Operational Decision Making in Networked Systems of Humans and Machines: A Multidisciplinary Approach*. Washington DC: National Academies Press, 2014.

[3] F. A. Bianchi, A. Margara, and M. Pezzè, "A survey of recent trends in testing concurrent software systems," *IEEE Tr. on Soft. Eng.*, vol. 44, no. 8, p. 747–783, 2018.

[4] V. Bellotti and W. K. Edwards, "Intelligibility and accountability: Human considerations in context-aware systems," *Human–Computer Interaction*, vol. 16, pp. 193–212, 2001.

[5] G. S. Blair, N. Bencomo, and N. B. France, "Models@run.time," *IEEE Computer*, vol. 42, no. 10, pp. 22–27, 2009. [Online]. Available: https://doi.org/10.1109/MC.2009.326

[6] Eclipse Foundation, "Eclipse EMF," Oct. 2021, date last checked: October 5th, 2021. [Online]. Available: https://projects.eclipse.org/projects/modeling.emf.emf

[7] O. Reynolds, A. García-Domínguez, and N. Bencomo, "Cronista: A multi-database automated provenance collection system for runtime-models," *Inf. and Soft. Tech.*, vol. 141, Jan 2022.

[8] G. Kiczales, J. Lamping, A. Mendhekar *et al.*, "Aspect-Oriented Programming," in *Proceedings of ECOOP'97*, ser. LNCS. Jyväskylä, Finland: Springer-Verlag, Jun. 1997, vol. 1241, pp. 220–242.

[9] B. Pérez, J. Rubio, and C. Sáenz-Adán, "A systematic review of provenance systems," *Knowledge and Information Systems*, vol. 57, no. 3, p. 495–543, Dec 2018.

[10] R. Wang, D. Sun, G. Li, M. Atif, and S. Nepal, "Logprov: Logging events as provenance of big data analytics pipelines with trustworthiness," in *Proceedings of BigData'16*, Dec 2016, p. 1402–1411.

[11] A. Gehani and D. Tariq, "SPADE: Support for Provenance Auditing in Distributed Environments," in *Middleware 2012*, ser. LNCS. Berlin, Heidelberg: Springer, 2012, pp. 101–120.

[12] M. Herschel, R. Diestelkämper, and H. Ben Lahmar, "A survey on provenance: What for? What form? What from?" *The VLDB Journal*, vol. 26, no. 6, pp. 881–906, 2017.

[13] N. Bencomo, S. Götz, and H. Song, "Models@run.time: a guided tour of the state of the art and research challenges," *Software & Systems Modeling*, 2019.

[14] P. Groth and L. Moreau, "PROV-Overview," W3C, Tech. Rep., 2013, date last checked: February 14th, 2021. [Online]. Available: https://www.w3.org/TR/prov-overview/

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

[16] Apache Foundation, "Apache TinkerPop homepage," Oct. 2021, date last checked: October 5th, 2021. [Online]. Available: https://tinkerpop.apache.org/

[17] D. S. Kolovos, R. F. Paige, and F. Polack, "The Epsilon Object Language (EOL)," in *Proceedings of ECMDA-FA 2006*, Bilbao, Spain, 2006.

[18] M. A. Rodriguez, "The gremlin graph traversal machine and language (invited talk)," in *Proceedings of DBPL'15*, ser. DBPL 2015. New York, NY, USA: ACM, 2015, p. 1–10.

[19] A. Kaur, Arvinder, and K. Johari, "Identification of crosscutting concerns: A survey," *International Journal of Engineering Science and Technology*, vol. 1, Dec 2009.

[20] R. E. Filman and D. P. Friedman, "Aspect-oriented programming is quantification and obliviousness," in *Workshop on Advanced Separation of Concerns (OOPSLA 2000)*, 2000. [Online]. Available: https://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.136.6632

[21] H. Mcheick and S. Godmaire, "Designing and implementing different use cases of aspect-oriented programming with AspectJ for developing mobile applications," in *Proceedings of the 7th International Conference on Software Engineering and New Technologies*, ser. ICSENT 2018. Association for Computing Machinery, Dec 2018, p. 1–8.

[22] M. Schäler, S. Schulze, and G. Saake, "Toward provenance capturing as cross-cutting concern," in *Proceedings of the 4th USENIX conference on Theory and Practice of Provenance*, ser. TaPP'12. USENIX Association, Jun 2012, p. 15.

[23] Sun Microsystems, "JavaBeans," 08 1997, version 1.01-A. [Online]. Available: https://download.oracle.com/otndocs/jcp/7224-javabeans-1.01-fr-spec-oth-JSpec/

[24] D. Brody, "ai-checkers," Jan 2021. [Online]. Available: https://github.com/dbrody112/ai-checkers

[25] B. Ricaud, "Graphexp github project," 2021, date last checked: February 7th, 2021. [Online]. Available: https://github.com/bricaud/graphexp

[26] R. Feldt and A. Magazinius, "Validity threats in empirical software engineering research - an initial survey," in *Proceedings of SEKE 2010*, Jan 2010.