AUTOMATED PROVENANCE COLLECTION OF RUNTIME MODEL EVOLUTION TO ENABLE EXPLANATION

Owen James Reynolds

Doctor of Philosophy

ASTON UNIVERSITY
September 2024

Abstract

Aston University

Automated provenance collection of runtime model evolution to enable explanation

Owen James Reynolds
Doctor of Philosophy
September, 2024

Context: New techniques exist to build large complex systems that perform autonomous decision-making. These systems may present emergent behaviours at runtime that were unforeseeable at design time, which may need to be understood. Data collected at runtime can be used to understand the system's behaviour. However, 'collecting runtime data' usually leaves developers deciding what data to log and how.

Objective: To develop a systematic approach to collecting runtime data. Furthermore, the approach should utilise automated techniques to minimise design-time costs while providing a consistent, reliable and robust implementation. Finally, the approach should maintain the runtime performance of a system.

Method: Develop, implement and evaluate an approach to collecting runtime data based on Model-Driven Engineering practices combined with provenance-based techniques. A series of case studies evaluate an implementation using two different target systems. The collected runtime data is analysed to verify that it contains indicators of observable system behaviours or can 'explain' the causes of a system fault.

Results: The experiments show that the proposed approach to collecting runtime data requires some extra effort. However, the system's rate of execution can be considered minimally changed.

Conclusions: Systematically collecting runtime data from a system to describe the changes and causes of changes to its runtime model provides insights into a system's behaviour. The system's design-time costs are managed via reusable and automated coding practices. Similarly, runtime costs are mitigated by adjusting the level of abstraction at which data is collected. Furthermore, data is stored using a representation that permits irrelevant data to be deleted.

Keywords: Runtime models, Model-Driven Engineering, automated provenance collection, self-explanation, self-adaptive-systems

Acknowledgements

In keeping with thesis traditions, I would like to acknowledge the support of the people in my life during my PhD studies.

To my wife, Melanie Reynolds, you are the best person in my life, and I thank you for the opportunity you found me with this PhD. I have met some wonderful people and had a great experience despite all the uncertainty and pressure!

To my boys Jack and James, you should always remember to aim for the moon, you will probably need to learn how to design and build a rocket to get there, but it will be worth the hard work, in the end, I promise.

To my supervisor Antonio García-Domínguez, your talents for teaching, communicating and technical expertise are exceptional, and they are only rivalled by your honesty, caring and compassion for your students. These are gifts I hope you never lose.

To my senior supervisor Nelly Bencomo, thank you for reminding me to look up at the stars and to think about the bigger picture. And I will try to remember your good advice about asking if I don't ask for something, I will never get something.

To my support tutor Chantal Karatas, you diagnosed my dyslexia, which had gone unnoticed all my life. You supported me as I started to recognise its effects on my life. I will often need to work harder than others, and it will often go unnoticed or unrewarded. I thank you for all your support.

To my last-minute supervisor Lucy Bastin, thank you for joining my supervision team in the last few months/year of my PhD. Your genuine interest and excitement in my work kept me going when things started to seem futile and pointless.

My fellow researcher students Thomas Carr, Renato Barros, Daniel Rodriguez Criado, Huma Samin, Luis Garcia Paucar, Juan Marcelo Parra, Sanobar Dar and many others. We may not have had a conventional time as PhD students, but we all seem to be getting through it. May you all have bright and successful lives with whatever you choose to do.

List of Publications

Towards Model-Driven Self-Explanation for Autonomous Decision-Making Systems
Date: September 2019
DOI: 10.1109/MODELS-C.2019.00095
Conference: 2019 ACM/IEEE 22nd International Conference on
Model Driven Engineering Languages and Systems Companion (MODELS-C)

Automated provenance graphs for models@run.time
Date: October 2020
DOI: 10.1145/3417990.3419503
Conference: MODELS '20: ACM/IEEE 23rd International Conference on
Model Driven Engineering Languages and Systems

Towards automated provenance collection for runtime models to record system history
Date: October 2020
DOI: 10.1145/3419804.3420262
Conference: SAM '20: 12th System Analysis and Modelling Conference

Cronista: A multi-database automated provenance collection system for runtime-models
Date: August 2021
DOI: 10.1016/j.infsof.2021.106694
Journal: Information and Software Technology 141(8):106694

Automated Provenance Collection at Runtime as a Cross-Cutting Concern
Date: August 2023
DOI: 10.1109/MODELS-C59198.2023.00057
Conference: SAM '23: 15th System Analysis and Modelling Conference

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Research subject presented in this thesis

The research presented in this thesis started around the beginning of 2019. A renewed interest in explaining computer systems had begun in Computer Science; this interest was ignited by the negative media attention on the perceived and actual problems caused by 'algorithms' or 'computer-based' decision-making (automated decision-making) in society. Crawford et al. [24] published a report on several social impacts these automated decision-making systems had on society.

An increase in the problems caused by automated decision-making was likely due to software systems becoming increasingly pervasive in society, which could be linked to advances in creating autonomous decision-making systems. These systems could be built using Artificial Intelligence (AI) and Machine Learning (ML) techniques. However, these systems are becoming unavoidably complex; they can perform increasingly more complex tasks independently (of human operators), in environments that are not fully understood. All these complexities mean that predicting a system's runtime behaviour at design-time is very difficult, and despite best efforts, these systems could still surprise their users and developers.

It is not just AI or ML techniques that can make a system's runtime behaviours difficult to predict or know. Concurrent systems are intrinsically non-deterministic: their runtime processes can be for complex interweaving sets of processes, which makes them difficult to test [11]. Furthermore, complexity can also come from the body of information a system uses when making decisions: knowing what information affected a decision outcome can be difficult [7, 98]. An example of a complex system that makes decisions is what is known as a self-adaptive autonomous system

(SAS). These systems adapt their behaviour to an environment under uncertainty, which can result in emergent behaviours that have unintended consequences [112]. Such a system may achieve the intended goals or objectives, but the process of 'how' the system does this could cause the user to have some reservations about its ability: e.g. visibly erratic behaviour, or a counterintuitive solution to a problem.

Crawford et al.'s [24] report demonstrates that society is responding to the broader usage of automated decision-making systems, raising questions about accountability and impact assessments, and calling for legislation. As a result, algorithmic accountability and the right to an explanation have become a vigorous discussion about what sort of explanation people are entitled to [101, 60]; General Data Protection Regulation (GDPR) [41] is an example of a legal requirement for a system's decision to be explainable. However, GDPR does provide a guide or practical advice on implementing a system that can explain its decisions that is compliant with the regulations [60].

Broadly, the research project seeks to enable systems with a form of self-explanations through automated techniques: automating the application of self-explanation functionality to a system, and automating the creation of explanations about a system's execution. These are 'explanations' in an abstract sense, the conveying of 'what happened' and 'why' during the system's execution; this approach does not produce a system capable of "question-and-answer" conversational explanations. However, it can provide data or enable a system to know about its execution activities: this could be used to enable a system with a 'conversational explanation' feature in the future.

The motivations and specific goals of the research project are discussed in Chapter 3. Next in this chapter is the research methodology, contributions, and thesis structure.

1.2 Research methodology

There is a range of research methodologies that can be used for Computer Science [57]. Traditionally, research methodologies are divided into two groups: quantitative, and qualitative methods [45]. Quantitative methods focus on collecting and analysing data statistically, whereas qualitative methods study data subjectively. Seaman [99] discusses the merits of using a mixture of both quantitative and qualitative methods for investigating Software Engineering problems. Certain problems lend themselves well to quantitative study: for example, comparing performance metrics like CPU, memory, and disk space of software components. However, qualitative methods can provide richer data

beyond statistics: these can be used to illuminate the statistical results of quantitative studies further. The example Seaman provides is that a quantitative study might statistically show a new software engineering technique is better than another, but a qualitative study (follow-up interviews with developers) might find the reasons for the improvement.

Owen describes design research in his paper "Design Research: building the knowledge base" [91], arguing design as a discipline is not science nor is it art, yet some knowledge is especially suited to be built using the study and practice of design. In this paper, the process of knowledge building or accumulation is abstracted as the product of action, where something is done and the result is judged in a cycle: knowledge is used, its use is evaluated, and the results of the evaluation produce further knowledge. This cycle of knowledge-building can occur in a realm of theory or a realm of practice. The realm of theory is analytical and based on findings or discoveries, whereas practice is a synthetic process of invention or making.

Based on Owen's description [91], we can identify design as an appropriate approach to build new knowledge in Computer Science. A software engineer building software can be viewed as an act of design science, because it involves inventing or making a software artefact; it is a synthetic process in the practical realm of knowledge creation, and the (software) artefacts can be evaluated to provide new knowledge.

Owen (1998), "Design Research: Building the Knowledge Base"

"In the acts of both doing and judging, questions are asked, answers obtained and decisions made. How these are formed is the key to using knowledge successfully to build new knowledge. Questions, answers and decisions differ fundamentally in nature from discipline to discipline. They are framed from the value systems embedded in the disciplines."

The design science literature further extends the ideas that Owen presented as design research. This literature includes Wieringa's Design Science Methodology [114], which describes designing and investigating artefacts for information system and software engineering research. The design and study of an artefact is performed in the context of an identified problem. An artefact is designed to treat (or improve) a problem in a context. The artefact's design is judged by evaluating its interactions with the problem. From the evaluation, new knowledge is gained, which informs the next design cycle

(if needed) and further improves the artefact's treatment of a problem.

Wieringa [114] provides a classification of empirical knowledge questions for evaluating a research artefact. The first classification is the knowledge question goal, which provides either a description or an explanation: *descriptive questions* ask for what happened with no explanation ('what', 'when', 'where' questions), and *explanatory questions* ask 'why' something happened. A second classification is open and closed knowledge questions: *open questions* contain no specification of the possible answer, and *closed questions* contain a hypothesis for the potential answer. This classification provides a total of 4 empirical knowledge questions. Wieringa [114] provide some example questions:

1. **Descriptive open question**: What is the execution time in this context?
2. **Explanatory open questions**: What input causes the dip in the graph of recall against database size? Is there a mechanism in the algorithm that is responsible for this?
3. **Descriptive closed question**: Is the execution time of one iteration less than 7.7ms?[1]
4. **Explanatory closed questions**: Why is the execution time of the method in these test data more than 7.7ms? Is this loop responsible for the high execution time?

There is another classification of questions which Wieringa [114] says is specific to design science research: empirical knowledge questions can be classified according to the subject matter (i.e. what the question is about). Because design science creates an artefact in a context, questions about the artefact itself can be asked. Wieringa provides examples of 4 categories of questions that can be asked in most design science research:

- **Effect**: what effects are produced when an artefact interacts with a context?
- **Trade-off**: what difference between effects occurs with different artefacts in the same context?
- **Sensitivity**: what is the difference between the effects of the same artefacts in different contexts?
- **Requirement satisfaction**: to what extent does the effect satisfy a requirement?

Honang Thuan et al. [56] tell us that the importance of clear research questions is well documented in the literature. However, they also report a lack of literature for guiding researchers on how these

[1] These are also called positive hypotheses, and are confirmed/falsified by empirical observations.

questions should be constructed. To remedy this problem, they analyse the design science literature to understand how researchers construct and formulate their questions. The analysis identifies several forms of research questions and ways to construct them, including a typology of common questions:

- **Way of knowing ("What [] is available?")** Questions of the existing knowledge, and what new knowledge research will produce.
- **Way of framing ("which [] define?")** These can be inner-world questions about the internal structure of an artefact, or outer-world questions regarding the artefact's usefulness or ability to meet a requirement.
- **Way of designing ("How can we []?")** These are the most common questions asked about how an artefact can be realised. A further sub-division of these questions into concerns is presented:
    1. **Conceptualizing** an artifact: "how can we represent?" or "how can we process?"
    2. **Operationaling** an artifact: "how can we implement?" or "how can we use?"
    3. **Evaluating** an artifact: "how can we evaluate?"

The publications from Wieringa [114] and Honang Thuan et al. [56] address the nature of questions in Computer Science, as Owen's [91] quote identified them to be different for each discipline. This leaves only the question of how to perform the actions of design science research. A design science methodology for information systems research is presented by Peffers et al. [92]: this publication describes the activities of each step in the design science cycles. Wieringa [114] described a broadly similar methodology. Peffers et al. demonstrated their 6-step methodology by applying it to existing works, follwing these steps:

1. **Problem identification and motivation**: setting out the problems that an artefact is being designed to treat, and a justification of the value or reason for creating the artefact.
2. **Objective of the solutions**: inferred from step 1, this step defines to what extent is an artefact expected to treat/improve a problem. This may include the use of existing knowledge to identify the scope of what is feasible to achieve with a new artefact design.

3. **Design and development**: it can be subdivided into the discrete activities required to produce an artefact. Design science can produce different types of design objects as artefacts: constructs, models, methods or implementations. Within these design objects, a research contribution is embedded in their design. Furthermore, this step may include determining the desired functionality and architecture of an artefact, instead of producing it. Existing knowledge or theory may be used to transition from the objectives step to the design and development step of creating a new artefact.

4. **Demonstration**: the developed artefact can be exhibited through experimentation, simulation, case study, proof, or similar activity.

5. **Evaluation**: using observations and measurements, an artefact is judged on its ability to treat a problem or meet a requirement. An evaluation may include empirical evidence or reasoned proof that the objectives in step 2 have been achieved: at this stage, a researcher may iterate back to step 3, improving the artefact further instead of ending the process and reporting any shortcomings.

6. **Communication**: the above steps have produced some knowledge which a researcher should publish and disseminate[2]. In the communication of the knowledge created through design science research, it is important to include descriptions of:

    - The problem and its importance.
    - The artefact produced.
    - Identify the utility and novelty of the artefact.
    - Rigour applied to the design.
    - The artefact's effectiveness or relevance for other researchers or stakeholders.

Peffers et al. explain how researchers would enter and move through the various steps in the design cycle. The process steps are nominally in sequential order, but a researcher is not expected to always start from step 1. In reality, a researcher could initiate their research from any step. Depending on how a design science project is initiated, a researcher may begin from a different step. 4 examples of design science initiation points are given:

[2]Wieringa et al. consider the communication step as part of research management in their methodology [114].

- **Problem-centred** projects follow a nominal sequence from step 1: a researcher identifies a problem, which may have been identified in other research projects through observation or reported as future work.
- **Objective-centred** projects are triggered by an external need for an artefact: for example, an industry or research project need. These start at step 2 with an objective.
- **Design and development-centred** projects investigate the use of an existing artefact which may not have been formally proposed as a solution to a problem in another domain. These projects are initiated from step 3.
- **Client/context-initiated solution** projects begin at step 4: a researcher seeks to retroactively apply design science rigour to an artefact which can be observed to solve a problem.

This thesis is based on Peffers et al.'s descriptions of design science because they provided a more extensive demonstration and description of their methodology. The mapping of the thesis structure to Peffers et al.'s methodology is covered in the next section.

1.3 Thesis structure

The thesis presents a design science research project that explores the creation of an approach to developing a computer system with a form of self-explanation, using automation. Following this introduction, Chapter 2 provides an overview of the literature on several topics related to the research presented in this thesis.

The chapters covering the research project activities are structured using the steps from the design science methodology by Peffers et al. [92]. Thus, Chapter 3 covers the initiation of the research project, with sections for each step: motivation for the project (Section 3.1), problems that the project could address (Section 3.2) and objectives (Section 3.3) of the project including the research question to be answered, and the scope of the project.

The design and development activities for the overarching research project are presented across three chapters. Each chapter presents the design science projects that evolve part of the research project's main artefact: from a conceptual design to an implementation, which is evaluated through experimentation and then further improved. Chapter 4 is divided into Sections 4.1 and 4.2, each

divided structured recursively according to the same structure as the overarching research project. Each section produces a conceptual design as an artefact to treat the problem outlined in Chapter 3.

The conceptual designs produced in Chapter 4 motivate the design science cycles in Chapter 5, which create an initial implementation (Section 5.1) of the conceptual design. The initial implementation is used in an experiment (Section 5.2) to evaluate the approach. Results of the experiment then initiate a cycle of refinement in Section 5.3.

The final design and development chapter 6 takes the artefact developed in Chapter 4 and tests its generalisability by changing the context under which it is used to treat the problem. Section 6.1 presents a project which extends the initial implementation for use in contexts beyond the intended scope defined in the objectives of Chapter 4. The chapter ends with Section 6.2, which applies a design science cycle to perform the experiment that evaluates the sensitivities.

Chapter 7 concludes the overarching design science research project by providing answers to the research questions in Section 7.1. There is a summary of future works in Section 7.2.

1.4 Contributions

As mentioned in the previous section detailing the structure of the thesis, a design science methodology has been used to perform the research presented. As such, the contributions of this thesis are not from the realms of theory: knowledge has been created in the realm of practice. The problems involved with enabling systems to explain themselves have been explored through designing and developing artefacts to treat the problem: this involved applying existing knowledge in a novel way. Solutions to the identified problem have been created, demonstrated, and communicated as prescribed by the methodology described in Section 1.2.

The work presented has created the following contributions:

1. The findings of this research project have been published in several papers listed in the front matter. (Page 4)

2. A data model based on a provenance ontology that represents the changes to a system's internal high-level abstractions as a *history model*, which creates a series of time windows for the system's runtime that enables *'forgetting'* of past records for storage management. (Chapter 4)

3. Cronista: a prototype implementation that demonstrates the conceptual modular architecture proposed for collecting, processing, and storing runtime data. The produced software artefacts are publicly available on GitLab [94], including the code for Cronista, the Traffic Manager and AI-Checkers case studies, and related experiments. (Chapters 5, 6)

4. Empirical data (processor, memory and storage resources) from several case studies (experiments) performed across different application domains (traffic control, AI for games). (Demonstration sections in Chapters 5, 6)

5. Evaluation of the utility for the data collected from the system executions in the case studies, using existing off-the-shelf technologies. (Demonstration sections in Chapters 5, 6)

Chapter 2

Background

Background reading is provided here to support the reading of the idea presented in this thesis. The contributed ideas build upon several existing concepts from the literature on modelling and provenance.

2.1 Models and Modelling

In this background section, models and the related modelling concepts are explained. An understanding of modelling will help in understanding the later sections of this thesis. The section presents the concept of models, how models can be used for software engineering, the use of models in software at runtime and the use of model versioning to track changes.

2.1.1 Models

Defining what a model is can be difficult, and people's interpretations of what a model is can be very different. However, there are some accepted descriptions in the literature, which will be useful to help understand later sections. Rothenberg [97] writes about AI, simulation and modeling, in which models are described as "a cost-effective use of something in place of something else". Rothenberg refers to the practical use of a model, where it can be cheaper, safer or simpler to use a model in place of a real-world counterpart or for some purpose. The model itself is an abstract representation of reality: this representation doesn't need to reflect all aspects of reality, only those required for the intended purpose.

Seidewitz [100] discusses what models mean in software development, and provides descriptions

that relate to Rothenberg's but are addressed in the context of systems, which may help relate models to the work in this thesis. Seidewitz states "A model is a set of statements about some system under study." The statements are an expression of the system under study (SUS), which are either true or false. Where Rothenberg uses the term "reality", Seidewitz uses "SUS", but in both cases, a model is considered a representation of something else. Kühne [69] adds to this description, "a copy is not a model"; the argument is that exact copies do not offer a reduction in cost like a model. Furthermore, copies do not suffer the inaccuracies of a model, which occur in the abstraction processes. Kühne identifies the need for copies, but they should be called *degenerate models* and not models, which are a reduced cost, abstract representation of something else.

Models and modeling can be used for different purposes, such as to describe an existing system or prescribe a new system [13]. Seidewitz [100] gives the following examples in terms of software development using the Unified Modeling Language (UML) [86]. A model could be created to show the components of a system and their relationships; a different model would be needed to describe the processes of the system. Conversely, UML could also be used to create a model as a specification (prescription) for a system's parts and their relationships. Again, different models would be needed to prescribe the processes that manipulate the parts of the system. All these examples use a model for a specific purpose and the model should only be used for that purpose.

Rothenberg [97] cautions against using models incorrectly or inappropriately. Failing to understand the intended purpose, limitations or interpretation of a model can cause serious harm: Rothenberg cites medieval blood-letting as an inappropriate use of a model. Patients were killed due to the physicians' assumed model of the human body, which presumed blood-letting to be beneficial for the body. To conclude, understanding a model is more than correctly interpreting what the model represents but also knowing the intended purpose and limitations of the model itself.

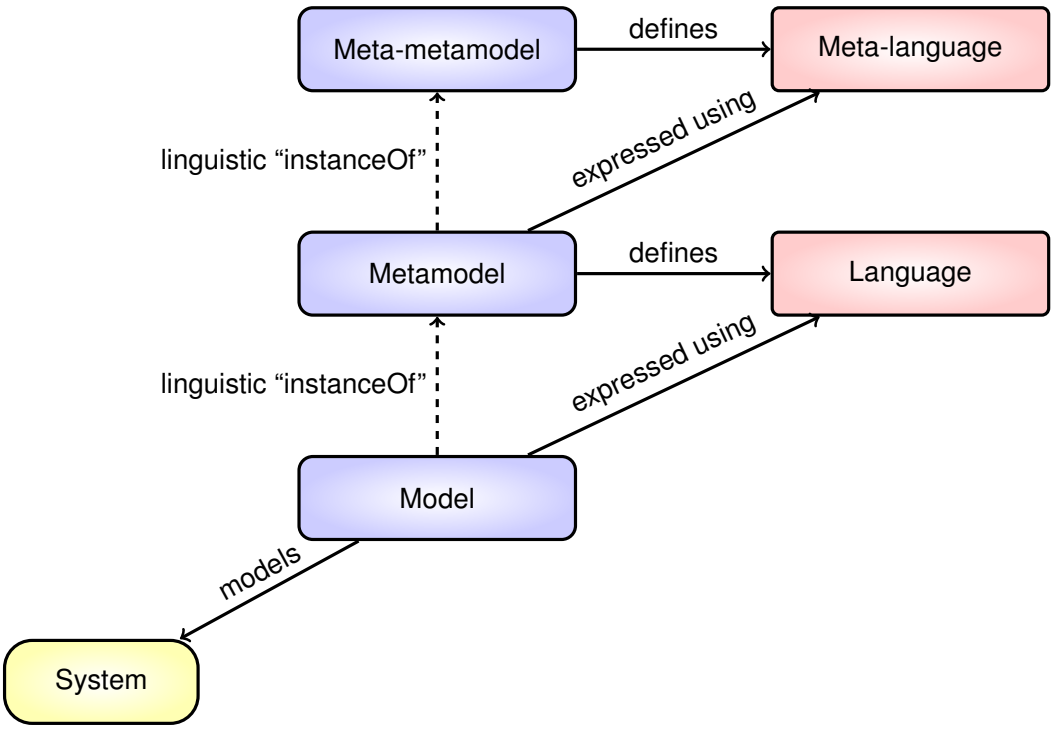

Figure 2.1: Kühne’s language definition stack[69]

Summary: Models

1. Models are a cost-effective replacement for reality (things or processes)
2. Models describe or are referent of something else
3. Models are abstract representations, not copies
4. Models have an intended purpose and their use for other purposes can be inappropriate

2.1.2 Model-driven engineering

Rodrigues da Silva [96] describes the use of models in MDE as being more than just documentation artefacts: the models are central artefacts in the software engineering process. In MDE, a “system” is a generic concept for a software artefact, which could be an application, platform or other software component [96]. Therefore, a model is a system in itself, and it can be defined by another model: this second model would be a metamodel. The prefix “meta” denotes that a modeling process has been

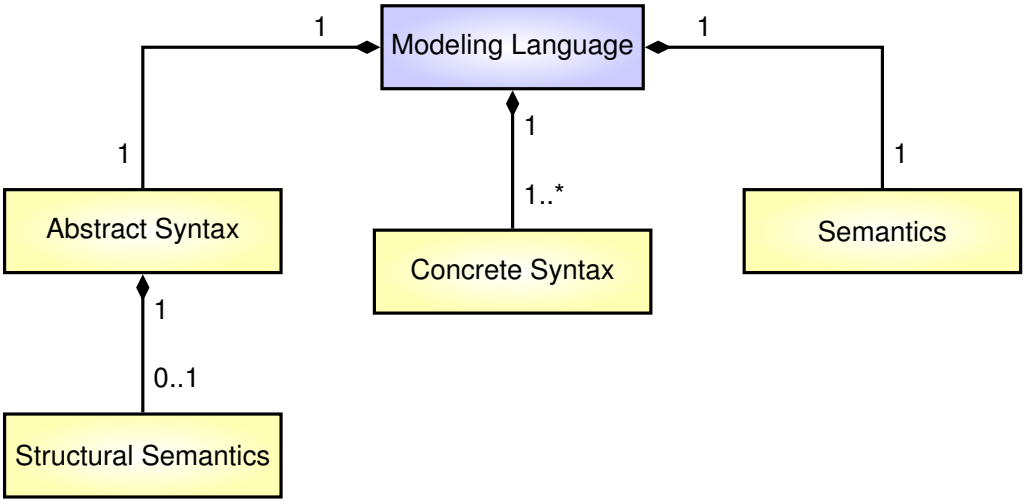

Figure 2.2: Rodrigues da Silva's Modeling Language definition

applied twice [69], producing a model of a model. However, Kühne [69] recognises some confusion arises with the terms "model" and "metamodel", which are frequently used in MDE discussions. A "model" can be called a "metamodel" under some circumstances; incorrectly using these terms can lead to communication problems and confusion. Kühne provides a language definition stack (Figure 2.1) that can help explain the relationships between a system, model and metamodel through the languages they define or are expressed in.

Looking at Figure 2.1, a *system* is modeled with one or more *model(s)*, which are expressed using a *language*: more specifically, a modeling language. The concepts of the modeling language are defined by a *metamodel*. Thus, the *model* is a linguistic instance of the *metamodel*. This relationship pattern is repeated again to the next layer up: the *metamodel* is expressed using a *meta-language*, whose concepts are specified in a *meta-metamodel*. Typically, the *meta-metamodel* is defined in terms of itself. Therefore, no additional layers are needed.

Figure 2.2 shows a model for modeling languages, which are composed of *abstract syntax*, *concrete syntax* and *semantics* [96]. The *abstract syntax* captures the concepts, abstractions, and relationships to be expressed: these are typically discovered from the domain or the domain knowledge of experts; the abstract syntax is the metamodel. Furthermore, *structural semantics* may be needed to support an *abstract syntax*, to apply additional constraints or rules which cannot be achieved with the *abstract syntax* alone. A *concrete syntax* defines the notation for the modeling language, this is, how a user would use the language to express a model; notations can be graphical or text-based. Furthermore, a modeling language may have more than one *concrete notation* to

enable different ways to interact with the model. Finally, *semantics* complete the modeling language: these can be executable semantics that define how a computer executes a model; or non-executable semantics that may define other rules or constraints that make a valid model. Therefore, a modeling language is defined by an abstract syntax expressed with a notation defined by its concrete syntax, and valid models can be interpreted according to the semantic rules.

The models in MDE need to be defined in a consistent way to enable model *transformations* [96]. There are two types of model transformations: *model-to-model* and *model-to-text*. Model-to-model transformations generate models from other models. Model-to-text transformations are used to generate textual documents or files from the model, which can be source code, configuration files, structured data such as XML or any other type of text file. Thus, a final software product may contain a mix of generated and non-generated software artefacts, where some components have been created from models using transformations and others have been manually created by a developer.

Summary: Model Driven Engineering

1. A model is a system and can be modeled with a metamodel.
2. A metamodel defines the concepts of the modelling language used to express models.
3. Meta-metamodel is usually the last layer in the language stack.
4. A model-to-model transformation produces 1 or more target models from 1 or more input models.
5. A model-to-text transformation produces text-file outputs for 1 or model inputs.

2.1.3 Model-driven-engineering technologies

Rodrigues da Silva [96] tells us that MDE is a paradigm and has no “one” concrete tool; MDE does not define or mandate the use of a specific model or modeling language. However, there are concrete tools that provide support for some common modeling tasks that might be performed when using an MDE approach to creating a system. For example:

- **Model editing**: some form of user interface (graphical or text-based) for a user to create or edit a model using a concrete syntax.
- **Model validation**: checking a model conforms to the semantics of its modeling language.
- **Model transformation**: from a set of source models to a set of target models.

Therefore, MDE technologies are tools that are designed to aid a user/developer working with models. These tools can be reused across multiple modeling languages that share a common technology, thus, used across many different models and systems built using models. They are not necessarily bespoke tools for creating a specific system for a specific domain.

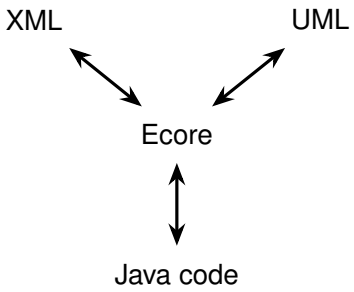

Figure 2.3: EMF's Ecore model representation of a model, which can be transformed into UML/XML/JAVA representations of the same model [104]

The Eclipse Modeling Framework [32] (EMF) is an open-source modeling framework that provides a meta-language (meta-metamodel) implementation (Ecore) with support for features to edit, validate and generate Java code implementations of a model [96]. Furthermore, EMF is the core of the larger Eclipse Modeling Project [33], which promotes model-based development technologies as a unified set of frameworks, tools and standard implementations. As such, there are a number of other frameworks/tools that have been created on top of EMF (Papyrus, Epsilon, XText), which broaden the scope of available technologies supporting EMF models. These tools save system developers time by removing the need to implement their own modeling tools for tasks like editing, validating and transforming.

An Ecore model can be created directly with an Ecore editor or generated from UML, XML, or Java code (Figure 2.3) [104]. It is also possible to generate UML, XML or Java code for a model defined using Ecore. Furthermore, the Ecore model can also be used as the metamodel of a modeling language based on its concepts.

EMF provides a model code generator; however, it also provides more than just the stateful

representations of the model as Java classes. A runtime framework for the model is included, which provides several features for using the model's generated Java classes in a system. For example, notifications for changes to model parts according to the Observer design pattern. These save a developer time as they do not need to implement and test their own routines for them.

Another feature provided by EMF's runtime framework is object persistence. This feature enables the model objects in memory to be persisted to disk, serialising the objects and their identities to a file. XMI is the chosen native file format for serialising models in EMF.

In much larger software projects, teams of developers typically work collaboratively on software artefacts. As such, these teams would maintain code or software assets in a version control system (VCS) like Git [103]. It is possible to store XMI representations of an EMF model in a Git repository as they are structured text files (XMI is an XML-based format). However, Git performs comparisons and merges at the line level, and is not model-aware. More suitable model-aware VCS solutions have been developed, like EMFStore [31] and Connected Data Objects (CDO) [28], which have native support for comparison and merging. CDO uses a database to store a model; its versioning is based on snapshots and branching (similar to Git) to track concurrent changes to a model. Branches permit developers to work concurrently on independent copies of a model; the developers later merge the branches containing the changed copies of the model.

When working with a large model or collaboratively modeling in a team, manually checking the changes made by a teammate can be a challenge. In the larger Eclipse project, there are tools that can help overcome these challenges, and some of them can be used directly with an Ecore model stored in CDO. For example, when merging two model branches in CDO, it is necessary to identify conflicting changes to the same model part; EMF Compare [29] provides tools for comparing and merging models. A more general need to find certain elements of a large model can be tackled using model query languages; for EMF models, some examples include the Object Constraint Language (OCL) [34] or the Epsilon Object Language (EOL) [67].

2.1.4 Runtime models

So far in this section, models have only been considered as design-time artefacts. Raising the level of abstraction is a feature of designing a system using models, which can make it easier to understand a complex system. Runtime models are models that are used during the execution of a system, to

raise the level of abstraction within a system's execution [12]. Furthermore, runtime models also allow a system to be self-aware with a model of their concerns, such as architectural aspects [40, 83] or requirements [9].

Runtime models can be used to build different types of systems, by modeling different concerns at different layers of the system. A recent survey from Bencomo, Götz and Song [10] 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.

A common example of the use of runtime models is self-adaptive systems or self-aware systems (SAS), as they use an architectural model representing themselves to self-reflect on their own state [10]. 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 [44, 18]. To do this, they maintain a runtime model [12]: "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. Furthermore, the system's manipulation of its runtime model to adapt itself to achieve its intended goal could be observed as changes to the model occur.

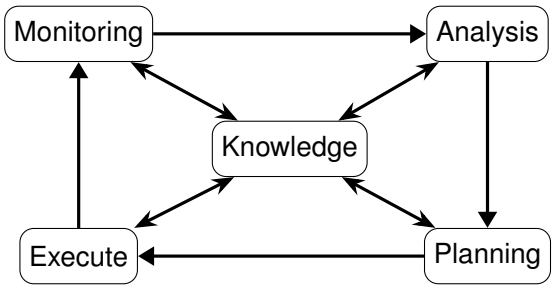

Figure 2.4: MAPE-K

Many self-aware systems operate as feedback loops, and MAPE-K is a common architecture for those [5, 43, 8]. The MAPE-K loop is divided into four phases: Monitoring, Analysis, Planning, and Execution, operating on top of shared Knowledge, which can be a runtime model. Each element is defined as follows:

- **Monitoring**: collecting data from the system and environment (via sensors).
- **Analysis**: evaluation of the data collected to assess the need for adaption.
- **Planning**: produce a workflow to adapt the system to meet a goal.
- **Execute**: enacts the workflow determined by the plan, which could change system parameters or drive an actuator to interact with the environment.
- **Knowledge**: or runtime model, that represents the system, environment, goals and other representations of importance to the system.

Runtime models can be implemented using EMF and CDO as Seybold et al. have demonstrated [102].

2.2 Aspect-Oriented Programming (Cross-cutting Concerns)

MDE is one way that code can be created through an automated process, but it is not the only approach. Aspect-oriented programming (AOP) is another technique for automating the production of code: this section will provide some background information on AOP.

A *cross-cutting concern* is defined by Kaur and Johari [65] as *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. [66] 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 [39] define AOP as *the desire to make programming statements of the form "In programs P, whenever condition C arises, perform action A".* AOP approaches vary in the programs they are applicable to, how the conditions are defined, and how the actions are woven into the existing code. Filman and Friedman [39] also consider that a good AOP system minimises the degree to which programmers change their behaviours to benefit from AOP.

Mcheick et al. [75] applied AOP to a Chess game without modifying the original code. In this study, they add logging, implement new movements for pieces, and attach a GUI. Their approach to logging is simple, as they insert console logging statements and trace all method calls from all

classes. However, their work demonstrates the use of AOP to access and extract (log) states from within a system, while also avoiding the tedious repetition of writing code for logging.

The Eclipse Foundation host the AspectJ project [30], which is a seamless aspect-oriented extension to Java. AspectJ is one example of a modern developer tool that enables developers to use AOP to cleanly manage crosscutting concerns as modules. The AspectJ project page suggests that AOP could be used for error checking and handling, synchronisation, context-sensitive behaviour, performance optimisations, monitoring and logging, debugging support, and multi-object protocols.

2.3 Provenance

This section will provide background on the general concept of provenance. Computer scientists have investigated the use of provenance concepts for tracking data and processes in computer systems.

2.3.1 Provenance general concept

Provenance is all the information that describes “how something came to be”, which includes the comprising elements, relationships and processes that produced them [19]. Pérez et al. consider provenance to be synonymous with lineage, which refers to the sources or origins of something [93]. They draw the comparison of provenance and lineage from the origins of the word provenance, which was used mainly when discussing of the history of artwork; the value of artwork can in part be determined by its history of ownership (provenance).

Collecting the provenance of something being created, exchanged or modified can be useful for several reasons. Herschel et al. [54] provides ‘a survey on provenance: What for? What form? What from?’. The survey broadly explains the applications/reasons for collecting provenances as:

- **Quality**: supply chain provenance can be used to assess the quality and the origins of a component or a product. For example, the provenance of food goods in a supply chain could tell us the farm and factory processes that produced it.
- **Replication,** scientific experiment provenance enables the reproduction of experiments or can make the result of difficult-to-repeat experiments more accessible as details of the process(es) are collected in addition to the results of the experiment.

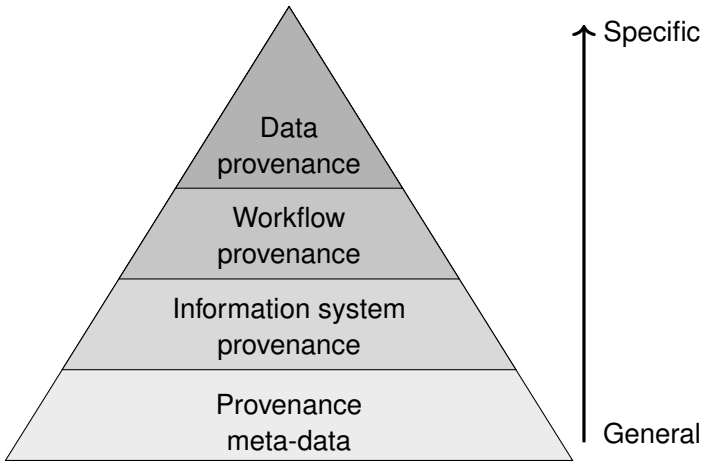

Figure 2.5: Provenance type hierarchy according to Herschel et al. [54]

- **Understanding,** complex data processing provenance can assist in understanding the processes that produced something. This could be for debugging purposes or refinement of a data analysis process.

In a prior paper, Herschel et al. [53] established a hierarchy of the types of provenance that can be collected (Figure 2.5). Provenance types are sorted in the hierarchy by domain scope, with more specialised or focused types of provenance at the top and generalised provenance at the bottom.

- **Provenance meta-data**, the most generalised form of provenance that can be used to describe any type of data and process. There can be an overlap with other forms of data collection, such as data collected to describe data (meta-data). For example, the resolution of a digital image is a form of meta-data that is required to process the image; however, the resolution would also be important information collected in the provenance describing the capture/post-processing of an image.

- **Information system provenance**, is the next layer up, which now confines the provenance to complying to a given standard representation. Constraining the provenance to a standard representation enables it to be interoperable between systems at the expense of some freedom to represent anything. An example of such a standard is W3C PROV, discussed in Section 2.3.4.

- **Workflow provenance**, further constrains information provenance to represent workflows as a specialisation, which might use custom representations. However, a standardised representation of provenance is often used to benefit from the interoperability aspects. Furthermore,

workflow provenance distinguishes between *retrorespective*, *prospective provenance* and level of *granularity* (explained later in this section).

- **Data provenance**, provides the highest level of detail by tracking the individual data items and processes that manipulate them. This approach typically can only be applied to structured data models and declarative query languages with clearly defined semantics for individual operations. Thus, also requires the highest instrumentation level to collect all the provenance data.

Finally, there are some terms that Herschel et al. [54] use to describe properties of provenance that should be explained in relation to workflow provenance: *retrospective* or *prospective* and the *granularity* of its information. In the context of workflow provenance, retrospective and prospective provenance describes when provenance is collected concerning the execution of the workflow.

- **Retrospective** provenance describes a workflow that has been executed.
- **Prospective** provenance predicts the outcome of a workflow before execution.

The term *granularity* is used to describe the level of detail or amount of summarising in the collected provenance information. A spectrum of coarse-grained to fine-grained exists. The appropriate level of granularity that provenance should be collected depends on the context and the provenance reader's knowledge of the workflow's internal processes.

- **Coarse-grained** provenance is suitable when the details of a process in a workflow are not needed: the details of the process are summarised.
- **Fine-grained** provenance exposes the inner workings of the processes within the workflow, providing as much information as possible.

The granularity of the collected provenance depends on the instrumentation level used to collect the provenance. As such, data provenance tends to result in fine-grained provenance as the level of instrumentation tracks changes to data items at a low level. Conversely, provenance metadata can be very coarse as the details are omitted or summarised. For example, the details of the process to render a digital image on a screen might be summarised to simply 'display image'.

In this section, provenance has been described based on the findings of Herschel et al. [54] in their paper titled "A survey on provenance: What for? What form? What from?". In the next section,

the “What from?” will be examined in the context of computer science as provenance is being used in the context of computing.

Summary: Provenance

1. Information that describes ‘how something came to be’
2. Provenance can enable or provide:
    - Assurances for the *quality* of a product or workflow.
    - Enable the *replication* of a process or workflow.
    - Assist with *understanding* a process or workflow.
3. Provenance can be:
    - General and *Coarse-grained*.
    - Specific and *Fine-grained*.
4. Provenance descriptions can be:
    - *Retrospective*, describing workflows that have happened.
    - *Prospective*, describes (potential) workflow outcomes.

2.3.2 Workflow provenance

Herschel et al. [54] provide examples of the applications for workflow provenance across domains, which include experimental science, business, data analytics and general programming. General programming is a broad domain, however, Herschel et al. highlight the defining workflow characteristics for this type of provenance collection:

- **Data dependencies**, changes to data in an application, for example, variables.
- **Control dependencies**, a program’s control loops and conditions.

Provenance collection from (general) programming languages is an active area of research. There are some implementations, which give some examples of the form that provenance collection from programming languages can take.

StarFlow [2] and noWorkflow [85] are two approaches for collecting provenance from Python script-based workflows. The workflows they are applied to are used for experiments and data analysis, thus overlapping with some of the other application domains mentioned above. Both collect retrospective fine-grained provenance from a workflow, however, the approaches taken to collecting provenance from Python scripts are very different. StarFlow [2] provides a script-centric Data Analysis Environment for programmers who are not software engineers. Thus, a programmer creates an abstract workflow using StarFlow's simple data model for workflow steps [2]. Therefore, StarFlow can only collect provenance for workflows created within its provided environment.

A less constrained approach to workflow provenance is provided by noWorkflow [85], which can be applied to an unmodified Python script. To enable provenance collection noWorkflow creates provenance definitions for the user-defined methods and variables, which are extracted from Python's abstract syntax tree. In addition, noWorkflow captures both deployment and execution provenance. The deployment provenance collects information about the environment the workflow is being executed in. Execution provenance is collected using Python's profiling API whereby it registers itself as a listener. As such, this approach is transparent and does not require users to instrument/change the workflow code or use a special environment like StarFlow.

Provenance collection from programming languages is not restricted to scripts. SPADE [42] is an example of provenance collection for compiled languages (specifically C/C++). SPADE started as a form of Operating System (OS) level provenance, which collected the provenance of applications interacting with OS resources. However, further development of the approach enabled application-level provenance, which is a finer-grained provenance that captures the internal states/processes of an application. SPADE captures application provenance data with instrumentation, which is added to an application at compile time; the instruments collect and emit data to an external SPADE process that collects and stores the data.

With different approaches to collecting provenance from workflows being developed (not just general programming), an interoperability problem arose with all the different provenance representations being created. These different provenance representations could not be easily exchanged between systems. Furthermore, the data from several different systems could not be easily combined into one provenance representation. Thus, to solve these interoperability problems, the Open Provenance Model (OPM) [81] was created as a first attempt to standardise provenance representations. The OPM provided the foundation for further work by the Provenance Working Group at the World Wide

Web Consortium (W3C [111]) who created PROV [78, 49] a standard provenance model which has a family of notations for recording and sharing provenance information.

Summary: Workflow Provenance

1. Example applications of workflow provenance include:
    - Experimental science
    - Business
    - Data Analytics
    - General programming
2. There are different approaches for collecting workflow provenance from general programming languages
    - Develop a workflow application in a provenance-aware environment (Starflow).
    - Pre-execution analysis of a workflow application's code to discover and create provenance definitions, which are then monitored at runtime using an existing monitoring API (noWorkflow).
    - Provenance instrumentation is added to the code of a workflow-based application at compile time.
3. Provenance ontology and standard models have been created to enable interoperability between different workflow provenance collection systems.

2.3.3 Data provenance

A common theme in the motivations for data provenance arises from the growth of the internet around the end of the 1990s and early 2000s. Buneman et al. [15] raise an issue with the ease with which data can be copied and transformed via the internet in a paper published in 2000. As data

moves around the internet it becomes increasingly difficult to determine its origins. The term *Data provenance* is used in the paper to broadly describe the information that would describe the origins of some data. The paper uses scientific data stored in databases to illustrate how data contained in one or more databases can be queried and combined to create a new database. This new database consists of data from multiple sources and if no data provenance is recorded then the origins of the data elements (tuples) are unknown.

Lynch published a paper in 2001 that discussed the impact of the internet on data trust and provenance [74]. The paper notes that information retrieval systems that index data (e.g. internet search engines) often assume the database system being accessed behaved correctly and that it is being administered in an appropriate 'trusted' way. However, the automated approaches for indexing the internet were (are) vulnerable to manipulation by the database systems they seek to index. A database system could be intentionally designed to feed an indexing system false information (data or meta-data) to manipulate the indexing system in some way; for example, to improve position in search results. This paper highlights the trust and provenance of data being important for the consumers of data, who need to make their own informed opinion about data/authors/publishers on the internet.

Data provenance seems to be motivated by the view that data can be easily copied or manipulated on the internet. The internet is portrayed as a collection of interconnected or networked databases sharing data. Therefore, data provenance is often associated with the provenance of data in database systems. This view of data provenance is reflected in some of the provenance literature surveys [54, 93].

The problem is seen as data from different internet-connected databases can be copied or integrated into a new dataset. The resulting new dataset may provide a new homogeneous view of the data that obscures the sources or origins of the data that makes up the new dataset [26]. However, sometimes it is appropriate to know where a specific piece of data in this new dataset came from; i.e. its 'provenance'. Buneman et al. [15] use 'in silico' experiments as an example of this need to know about the provenance of data. Specifically, Buneman looks at molecular biology databases as an example of the importance of data provenance. The example is given that these experiments use and combine data from different sources for new computer-based experiments. The new computer-based experiment would produce a dataset which may have inherited data issues from the original data or any processes that modified the source data before it was used in the most recent work. Thus, knowing the provenance of the new dataset and its source datasets helps inform

the new dataset user's opinions on qualities like correctness, accuracy or trustworthiness.

Data provenance is sometimes called *data lineage* or *data pedigree*: these names are used to describe similar types of data that describe where a database tuple or data element came from [26]. This database-orientated viewpoint has a very different approach to provenance compared to workflow provenance. Data provenance research comes from a more mathematical or algebraic approach with a strong focus on how a dataset was derived using a database query.

Lineage is an early formalism which considered the tuples returned by a database query to have a form of provenance relationship with the input tuples the query was run against. Input tuples that "contribute" a tuple being included in the query output are considered to be the provenance of the output tuple. Around the same time Buneman et al. [16] characterised the data provenance relationships between input and output tuples from queries further. Where-provenance defines where data was copied from or where output data relates to the input data. Why-provenance explains why a tuple was created, as a justification for the tuple appearing in the query output. Both of these approaches used algebraic representations for the provenance of the tuples produced by a query.

In 2007 Green et al. [46] presented a paper on *provenance semirings*, which proposes the use of algebraic structure to communicate provenance. Having reviewed the earlier works, they spotted similarities in the calculations and annotations. They identified communicative semirings as the appropriate algebraic structure for representing rich provenance information. A few years later in 2012, Doan et al. [26] wrote in chapter 14 of the book 'Principles of Data Integration' that provenance semirings formalism (semirings formal model) is the most general representation of data provenance. It provides an algebraically equivalent representation for both the relational expressions and the provenance of data in a database (tuples).

Cheney et al. [21] surveyed the approaches to compute provenance in databases around 2007. There are two distinct approaches to calculating provenance, *Eager* and *Lazy*. Eager approaches (known as *bookkeeping* or *annotation*) compute provenance at the time of query execution, the resulting dataset and provenance are output (this may be to a new database). Lazy approaches do not compute provenance at the time of query execution, this is only done after the query is executed if needed. Obtaining the provenance data via a lazy approach requires both input and output datasets with the transformation (query) that was used. The approaches both have different characteristics. For example, eager adds overhead at query runtime whereas lazy does not. Lazy requires sophisticated reasoning to calculate the provenance after the fact but does not change the transformation process.

Eager may be easier to implement but it requires the transformation process to be altered to add the provenance calculations.

There is a curious point as to where workflow provenance ends and data provenance starts. Herschel et al. [54] considered data provenance as finer-grained than workflow provenance, as it can identify the source of a single data tuple in a database. Workflow provenance being more coarse-grained may describe the source of tuples in a database more broadly, “tuples in database C came from a process that merged databases A and B”. Doan et al. describe the difference between data provenance and workflow provenance in a similar way [26]. A workflow can be considered a pipeline of ‘black-box’ interconnected processes: the details of what happens inside the ‘black-box’ are unknown. Whereas data provenance exploits the knowledge of relational operator semantics used in a process and makes the process ‘transparent’, exposing all the process details. These ideas of workflows, black-boxes and transparency arise in other publications, relating to explanations and accountability [115, 61]. Herschel et al. [54] also comment that fine-grained workflow provenance is similar to data provenance when it exposes an equivalent level of detail. Data provenance is dependent on processes having clearly defined semantics like database queries, which can be used to produce fine-grain provenance data.

The differences and similarities between data and workflow provenance are addressed in a paper by Acar et al. [46]. This paper bridges the gap between ‘workflow-style’ provenance which is described as ‘batch processing steps’ and ‘database-style’ querying. Data provenance is often expressed in an algebraic form, whereas in workflow provenance a directed graph notation is often used. The paper uses a Haskell (functional programming language) program execution as an example for creating provenance in both forms. Dataflow calculus and Open Provenance Model provenance graph visualizations are created for data and workflow provenance representations of a Haskell program execution. While the author acknowledges the work is based on early versions of both standards and there is room for debate on the design; it does demonstrate that the practical workflow provenance approaches and formal algebraic approaches to data provenance are related.

Doan et al. [26] discuss applications for data provenance and present three classes: ‘explanations’, ‘scoring’, and ‘reasoning about interactions’. Explanations for how tuples came to be could be useful for debugging mappings or cleaning data sources. Scoring may be something of interest when rating confidence in a dataset’s accuracy. Reasoning about interactions is where the provenance of tuples is examined to search out dependency relationships between different tuples. This is a surprisingly

rare occurrence in the literature where ‘explanations’ are mentioned with provenance. It is slightly surprising to have not seen more comments like ‘Data provenance explains how a dataset was created’, this is likely because ‘Data provenance describes the origin of a data set’ is a more precise description that is less likely to be misinterpreted.

Summary: Data Provenance

1. Considered the finest-grained provenance and often associated with databases and provenance of data elements (tuples) within databases, which have lineage, or provenance from other databases via a query.
2. Provenance of tuples in a dataset can be represented in an algebraic form as a formal proof of provenance for each tuple, this representation can be calculated from the logic in the query e.g. provenance semirings.
3. Computing Why-, Where-, How-Provenance for a dataset can be done using an *eager* or *lazy* approach.
    - **Eager**, at query execution time which requires a query to be modified to add the provenance computation.
    - **Lazy**, retrospectively by reversing the semantics of the query logic with input and output datasets.
4. Algebraic provenance notations can be mapped to graph notations, this relationship forms a connection between the different approaches to data and workflow provenance.

2.3.4 W3C PROV — PROV-DM

The World Wide Web (W3C) Consortium published PROV, a family of specifications for recording and representing provenance [78, 49]. One of the central ideas for PROV was to create a domain-agnostic provenance model which can be used across heterogeneous environments, such as the Web, where

different PROV notations can be used to record and share provenance information. These PROV notations share a common model that enables the interchange of provenance information between different systems or processes. For example, a human-readable notation might enable a human to input some provenance information. The input provenance data is then transformed into a notation for automated processing by a machine. Finally, the provenance data could be shared with an external system, which could be in another organisation.

Below is a list of some of the W3C PROV specifications documents on their website for provenance notations:

- **PROV-DM**, the conceptual data model for provenance
- **PROV-O**, OWL2 Web Ontology Language provenance notation
- **PROV-N**, a human-readable provenance notation
- **PROV-XML**, an XML schema for provenance

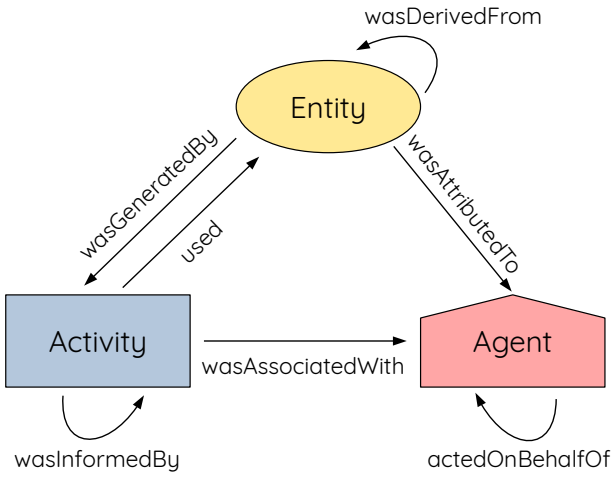

Figure 2.6: W3C PROV-DM provenance graph containing nodes and directed edges

PROV-DM provides the core structure or conceptual data model for the PROV family. It captures the relationships and intrinsic elements of provenance [78]. Figure 2.6 shows the PROV-DM model as a provenance graph. The nodes represent provenance elements: AGENT, ACTIVITY, ENTITY. The directed edges between nodes represent the relationships between them, such as *"used"*, *"wasAssociatedWith"* or *"wasAttributedTo"*. These nodes and edges are combined to form provenance statements about a workflow.

The provenance nodes are defined as:

- **AGENT** is used to attribute responsibility or ownership to a person or organisation, enabling provenance statements such as an AGENT 'Bob" *"wasAssociatedWith"* an ACTIVITY "generate report" or an ENTITY "report" *"wasAttributedTo"* AGENT "Bob".

- **ACTIVITY** defines a process or action that may encompass an entire process or a small step of a process. If working with fine-grained steps, these will likely be chained together with wasInformedBy relationships. An example provenance statement may be that an ACTIVITY "generate report" *"used"* ENTITY "report data".

- **ENTITY** represents things and concepts; things can be physical items such as paper-based documents or digital files. The concepts enable capturing information about less concrete things, for example, capturing states, e.g. "report status is approved for publication". Finally, ENTITY nodes can be used to track versions or instances of things over time as a chain of ENTITY nodes, connected with *"wasDerivedFrom"*; an example of such a provenance statement would be ENTITY "report" *"wasDerivedFrom"* ENTITY "draft report".

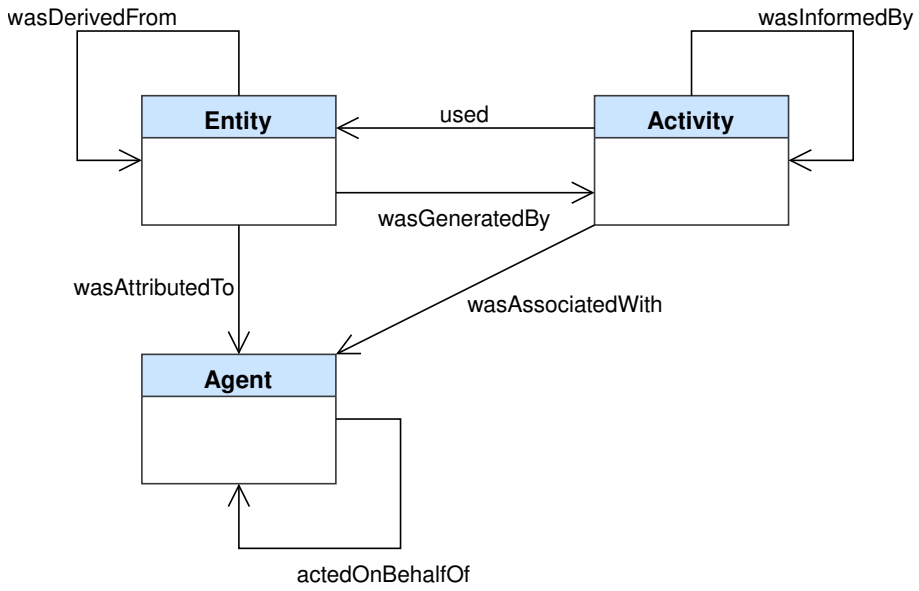

Figure 2.7: UML diagram of PROV Core Structures (Informative) [49]

The PROV-DM specification provides a UML diagram (Figure 2.7) of the provenance elements and relationships, which provides more detailed guidance on the fields within the classes used to represent elements. However, the richer features and detailed specifications that cover the namespaces, collections or extended relationships do not need to be understood for this thesis. Furthermore, the other notations (PROV-O/-N/-XML) are alternative serialisations of the PROV-DM

model, which again do not need to be understood for this thesis. The core structure of PROV-DM described in the UML diagram could be implemented using MDE approaches, which provide their own serialisation methods.

Summary: PROV-DM

1. Provides the conceptual model for PROV, containing elements (nodes) and relationships (directed edges).
    - **AGENT**, responsible person or organisation.
    - **ACTIVITY**, action or process.
    - **ENTITY**, concept or thing (physical or digital object).
2. Elements and relationships are combined to form provenance statements, which further combine to create a provenance graph.
3. Provenance graphs can be visualised as a directed graph or serialised using a notation such as PROV-O/-N/-XML.

2.4 Prior works relating to computer systems and explanations

Computer science has explored the idea of enabling systems with different forms of 'explanations' before. The idea of a computer/program/system needing to explain itself to a human is not new. However, the type of systems and reasons for wanting explanations does change slightly, but the theme of needing explanations seems constant. There are several reasons why a human might want or need to understand what a system has done or is about to do, and several ways the problem might be approached. This background section provides some examples from the literature following an approximate chronological order of publication.

Explaining and Justifying Expert Consulting Programs Swartout [106] identified the need for an expert program to explain what it does and justify its actions understandably. XPLAIN is an approach that uses a domain model and domain principles with an automatic programmer [1] to create

[1] The description of an automatic programmer is similar to that of a code generator: a computer is provided 'knowledge' of a system to build, and produces the code implementation.

a system with a refinement structure. A set of explanation routines uses the refinement structure to retrieve information about decisions made during the creation of a program to provide explanations. Swartout's paper predates Model-Driven Engineering literature by decades, but his discussion of models, domains and automatic programming resonates strongly with MDE.

Swartout (1981) quote 1

"The reason we can't get the sort of explanations we want by producing explanations directly from the code is that much of the sort of reasoning we want to explain has been 'compiled out'. Thus, we are forced into explaining at a level that is either too abstract or too specific. The intermediate reasoning which we would like to explain was done by a human programmer in the case of the Digitalis Advisor."

In the quote above, Swartout makes an astute observation about why explanatory capabilities at the time were inadequate. The human programmer is assumed to have done some intermediate reasoning, which is undocumented. Thus, Swartout argues that an automatic programmer could be used to create a program and provide some descriptions of its code using intermediate levels of abstraction; these intermediate abstractions would then support the program's explanatory capabilities. However, it is not entirely the human programmer's fault, as Swartout questions if automatic programming is too hard.

Swartout (1981) quote 2

"It seems that the primary difficulty in both explanation and automatic programming is a knowledge representation problem and that the kinds of knowledge to be represented in both cases are similar so that a solution to one case makes the other much easier."

Without creating sufficiently powerful representations of knowledge in the design and development of a system, a loss of knowledge or information occurs that makes both automatic programming and explanations difficult. Again, Swartout's comment in quote 2 connects back to similar problems that MDE seeks to solve with modelling languages and models: the use of models and transformation for 'communicating' knowledge between domain experts and software developers. While the terminology

and perspective of concepts like automatic programming have changed since 1981, Swartout seems to be identifying the difficulties with managing knowledge during the design of a system, and that the knowledge required for explanations is not available or lost due to some abstraction at design-time.

Explanations from Intelligent and Knowledge-based Systems Gregor and Benbasat [47] argue that 'Intelligent and Knowledge-based' systems should in principle be able to explain themselves by offering a reason or justification for their behaviours. Their research questions ask:

- Do the users of intelligent systems want explanations?
- What benefits arise from the explanations?
- What type of explanations should be provided?

They conclude that users of these systems infrequently use the explanations a system provides: however, the use of explanations is likely to be highly context-specific. Users tend to need explanations when there is a need to resolve a perceived anomaly, and the user desires to learn more about the intelligent system and its procedures, which could arise from a need to report on production or system debugging. Gregor and Benbasat see system explanations as beneficial for users: a user's perception of a system improves as their learning/knowledge of the system improves. Gregor and Benbasat provide three classifications for existing explanation approaches: content, presentation format, and provision mechanism. The presentation format has the least empirical evidence to support the use of different available methods, i.e. a 'best' presentation format for an explanation has not been identified. An explanation's content and provisioning mechanisms are more important or useful under certain conditions or contexts.

Gregor and Benbasat categorise explanations by content

Gregor and Benbasat presented a table (list below) of classifications of explanation by content type [47], which is adapted from prior works: Chandrasekaran et al. [20] provided types 1–3 (Trace, Justification, Control), and Swartout and Smoliar [107] provided type 4 (Terminology).

1. **Reasoning trace** explaining why a certain decision was made based on data and rules for a specific case.
2. **Justification** using deep domain knowledge to support the reasoning process for a decision.
3. **Strategic** describing a problem-solving strategy or a system's control behaviours.
4. **Terminology** terminological information that supplies definitions for domain concepts

Explanations for Context-aware systems Dey et al. present in earlier work a toolkit for developing a context-aware application [25]. These context-aware systems have some degree of decision-making autonomy whereby they can act independently based on some detected context, e.g. a system can automatically take action without explicit user input based on a set of rules and sensor data. However, these systems lack transparency and can make mistakes that make it difficult for their users to understand them [7]. This lack of understanding and mistrust arises from a lack of application *intelligibility* and can result in the users misusing or abandoning the system [84]. Bellottie and Edwards propose that a context-aware system could be made more intelligible if they could show users what it knew, how it knew, and what it was trying to do [7]. Lim and Dey assessed the demand for intelligibility in context-aware applications [71] in a paper where they connect explanations and intelligibility ideas; they categorize the type of explanations in terms of questions a user may ask into *intelligibility types*. They investigate the demand for each intelligibility type and recommend why, certainty and control intelligibility types should be available for all context-aware applications; In contrast, the demand for others can change or is only useful in a specific context (e.g. why not for a goal-supportive explanation).

Lim et al. presented a more detailed lab experiment that investigated the improvement of context-aware system intelligibility with 'why and why not' explanations [70]. In this paper, they cite Gregor and Benbasat's paper [47] in acknowledgement of the studies of the use of explanations with Knowledge-Based System; Lim et al. identify a knowledge-based system user is typically an expert seeking explanations with expert knowledge. In contrast, Lim et al. seek to provide explanations to novice end-users of context-aware systems explanations, to help improve their understanding of novel context-aware systems. A large controlled study is devised to investigate the value of four types of intelligibility questions a user may ask: why, why not, what if, and how to. They conclude that providing reasoning trace explanations [47] for context-aware applications (for novice users) improves the user's understanding of the system, notably in answering 'why' questions (explaining). Lim and Dey were convinced of the value of enabling a context-aware system with explanations; they presented a toolkit to support intelligibility in context-aware applications a year later [72]. This toolkit is intended to support developers with enabling 'explanation' functionality in a context-aware system to make them more intelligible.

Self-explanation in Adaptive Systems The behaviour of a self-adaptive system (SAS) can be emergent, which can surprise the developer or user of a system with unpredictable behaviour [112]. There are similarities with the novice users of a context-aware system that performs automatic actions, which the user does not understand. Welsh et al. [112] discuss the problems with 'intelligibility' and 'trust', which could cause users to stop using a self-adaptive system citing Muir [84]. They identify the work of Lim et al. [70] who improved the intelligibility of context-aware systems with explanations for a user's 'why and why not'.

Welsh et al. [112] propose the same runtime models used for managing unforeseen changes in a SAS that could be used to enable a system with a form of 'self-explanation'. They characterize self-explanation as *why* = $\{$*what, how, history* $\}$: **why** represents the explanation for **what** was observed. This observation was a consequence of **how** the system satisfied its requirement, after interpreting the **history** of adaptation events. They conclude enabling a SAS to have a form of self-explanation would allow emergent behaviours (not prescribed at design-time) to be diagnosed, understood, and explained. Again, this is relatable to prior works with making context-aware systems more intelligible; extending systems with 'explanation' capabilities to assist someone with a desire to learn and understand a system better. Furthermore, the use of a runtime model as a source of

'knowledge' from within a system for creating explanations is relatable to other prior works like Gregor and Benbasat [47] and Swartout [106].

Explaining AI systems, or Explainable AI (XAI) Adoption of Artificial intelligence (AI) systems into people's daily lives is fast and wide [1]. These AI systems are also creating problems in people's daily lives, as they are hard to explain due to a lack of transparency, and disputes are difficult to resolve due to a lack of accountability [24]. Adadi and Berrdad published a survey on Explainable Artificial Intelligence (XAI) [1]. Broadly, XAI research explores ways to peek inside the 'black-box' of an AI system to try and 'explain' them. Adadi and Berrada provide a list of motivations or reasons for XAI and improving the explainability of AI systems:

Adadi and Berrada's list of motivations for XAI

1. Explain to **justify**, a decision or action of an AI system.
2. Explain to **control**, improve the understanding of a system to gain viability of unknown vulnerabilities or flaws; enabling them to be corrected enhancing control over the AI.
3. Explain to **improve**, similar to control a greater understanding of an AI system through explanations can be used to improve the system i.e. help to make AI systems 'smarter'.
4. Explain to **discover**, asking for explanations can aid the learning of new knowledge, XAI could enable humans to learn new knowledge that an AI system discovers e.g. new/hidden laws in biology, chemistry or physics.

The listed motivations for explanations (XAI research) all seek to extract some knowledge from a 'black-box' AI system that conceals its internal processes and knowledge; the list represents what could be done with the knowledge gained through explanations from an AI. This high-level view of XAI of seeking knowledge from inside a system through explanations connects with the prior works reviewed in this section of the background. However, XAI has a distinct problem that is different to the systems considered in prior work; Burrell [17] explains "When a computer learns

and consequently builds its own representation of a classification decision, it does do without regard for human comprehension". The prior explanation work assumes that the 'knowledge' encoded in a system can be comprehended by a human; Burrell is telling us that a trained AI system's knowledge may not be. Thus, in the XAI survey [1] interpretability extends to enabling an AI's internal models comprehensible by humans, which is an additional step beyond simply extracting knowledge through having a system provide explanations. Explaining these types of AI systems is outside of the scope of this thesis, because explaining the inner workings is a complex problem best left to XAI; AI components in a system can be explained as a black-box in the context of a larger system without knowing its inner workings [61].

2.5 Provenance awareness

Provenance awareness is a name given to a function or property of a system that records the provenance of its processes during execution and can later answer the user's provenance questions about its process [77]. The most recent state-of-the-art work uses the ideas of provenance awareness to build systems that can provide provenance-based explanations for legal/regulatory compliance; this research occurred in the same time frame as the research presented in this thesis.

This section summarises the literature on provenance-awareness and presents the work in approximate chronological order. The subsections group the publications around the development of an architecture for a provenance system, methodologies for applying provenance-awareness to a system, and the use of provenance-awareness for enabling provenance-based explanations.

The provenance awareness literature provides many answers for collecting provenance at runtime as a description of the execution of a system. However, the work focuses on systems with Service-Oriented Architecture (SOA) and collecting provenance for interaction between a network of systems-sharing services. This creates a provenance view of execution where a system is considered a collection of component systems (actors); component systems provide services to one another. Services are accessed through an exchange of messages (data items). SOA systems achieve high-level system goals by combining several services (processes). This thesis is not exploring explanations for SOA, but there are some common provenance-related problems and solutions that it can be compared with.

2.5.1 Architecture for building a provenance system

A *provenance system* is a system for collecting, storing and analysing provenance data recorded from another system. This section of the background of provenance awareness reviews literature that discusses how a provenance system is constructed, and the principles for recording provenance.

An Architecture for Provenance Systems

Groth et al. published “An Architecture for Provenance Systems” in 2006, which uses the term “provenance-aware(ness)” [52] to describe a system that collects provenance during execution (runtime) and produces *process documentation*. A *provenance system* deals with provenance information issues such as recording, maintaining or accessing provenance data. The *provenance architecture* identifies the roles, interactions and kinds of provenance representations a provenance system has. Thus, a provenance system provides facilities to support another system which is processing data and requires provenance information recording.

This paper identifies ‘computational provenance’ in the executive summary as the idea that computer systems (or applications) produce data during an execution: a system could collect provenance information for the data being produced while executing. Systems enabled with this feature would be considered to be *provenance-aware*. These systems create *process documentation* for the provenance of the data being produced while they execute. These process documents are then stored in a *provenance store*. The paper details a provenance system architecture for supporting the creation of provenance-aware systems/applications. The paper includes summaries of work presented in other papers such as PReP and PASOA.

The logical components of the provenance system architecture and its scope are shown in Figure 2.8. These are implemented as software components that would be added to or alongside a system whose execution processes are to be recorded. This section will discuss the refinements to the description of provenance that the provenance system architecture introduces. Then the section will discuss in more detail the higher-level components of the architecture.

Refinements to the description of provenance

The above provenance system architecture from Groth introduced some additional refinements to the description of provenance. These refinements help to outline the motivation and objectives for a

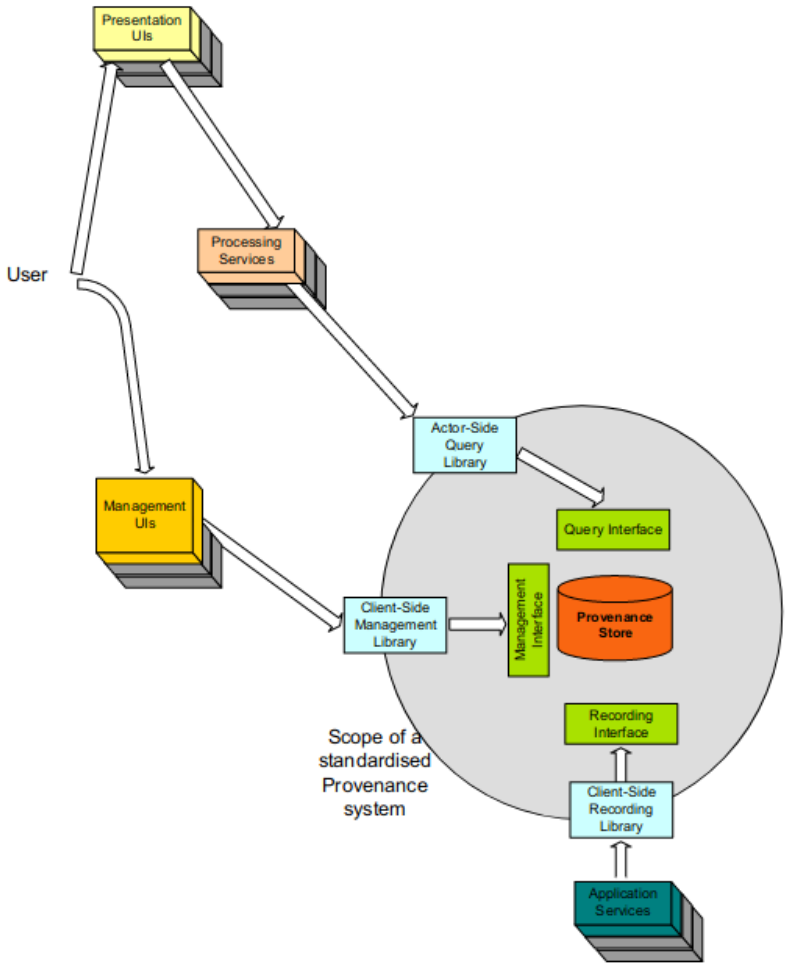

Figure 2.8: Logical view of a Provenance system and its scope [52]

provenance architecture.

Groth described a four-phase **provenance life cycle** for provenance data: creation, recording, querying, and managing. Reflecting on the life cycle phases, the architecture paper offers the following refined views of provenance:

1. Provenance can be seen as a concept for explaining how a result or piece of data came to be.
2. Provenance assertions (*p-assertions*) are created during execution, and constitute documentation of the execution (process) including information representative of output data production.
3. When p-assertions are queried, a sub-set of p-assertions are filtered out and presented in some form as query-time representations of provenance.

Groth and Moreau [48] describe high-quality provenance documentation as having the following characteristics:

- **Immutable**: process documentation is protected from tampering (or deletion) even under the guise of correcting errors.
- **Attributable**: the responsible party for a process can be traced and held accountable.
- **Autonomously Creatable**: a component of a system (which is composed of multiple components) can document its own processes without having to synchronise with other components.
- **Finalizable**: the process and its documentation should be identifiable as being finalized or complete. This is important when process documentation occurs asynchronously with the process.
- **Process Reflecting**: the resulting process documentation reflects the process it represents in a way that enables the reconstruction of the process using the documentation.

Provenance modelling occurs as p-assertions are created for actor interactions. An actor creates a p-assertion to describe a process they are performing from their perspective. These p-assertions are made in the context of an interaction, and form a provenance model of the interaction. The p-assertion can be instantiated using data structures like XML or application-specific binary formats. Three types of p-assertion have been identified: *interaction*, *relationship*, and *actor state*. The architecture provides guidance on identifying p-assertion types and modelling them, but it does not prescribe a specific encoding for instantiating them.

Identifying interactions univocally for provenance collection requires assigning them globally unique keys (*Interaction Keys*). An example of how to compose the key is given for a distributed system, where actors exchanging some messages is identified as an interaction. This approach uses three parts to compose the key: a message source address (*messageSource*), a sink address (*messageSink*) and an *interactionID* that distinguishes the instance a message between these addresses are passed.

Interaction key composition
messageSource + messageSink + interactionID

P-assertions may be created for an interaction involving multiple messages between the actors involved. Thus, p-assertions need a *Local P-assertion Identifier* which uniquely identifies each assertion in the interaction. The asserting actor also needs to identify the *view kind*, or perspective of the interaction that a p-assertion is being made from. For example, an assertion can be from the perspective of a message sender or receiver. A unique *global p-assertion key* can then be composed by combining these new components with the *Interaction Key*.

Global p-assertion key composition
Interaction Key + View Kind + Local P-assertion Identifier

A p-assertion can contain all the assertion data, or optionally a *p-assertion data item* can be used to store the data which can be referenced by several p-assertions. To do this, a globally unique *p-assertion data key* needs to be generated for the p-assertion data item. This key would be stored as a *data accessor* reference that enables the data to be recovered for a given global p-assertion key.

Having specified how to uniquely key or identify instances and p-assertions, the models used for each type of p-assertion can be defined.

Interaction p-assertions: in the context of a distributed system, these assert the content of a message being sent or received by one of the system's actors. Thus, there should be one p-assertion for each message, each with a Local P-assertion Identifier. If several messages are exchanged in an interaction, these should all be marked with the same Interaction Key. The message content itself is then attached to the p-assertion as *content* with a *document style* that defines the styling or type of data (e.g. SOAP). There is a need to ensure all asserting actors that are creating p-assertions for the same interactions are using the correct keys and identifiers. To do this, *context* and *p-header* are used to exchange information between actors. Context can be considered as meta-data about an interaction: this context is encapsulated into a p-header that is sent as a message from one actor to another. Both context and p-headers should include the interaction key.

Actor State p-assertions: an actor can disclose its internal state information, which provides further detail about the execution process being documented. These p-assertions should be identified using the Local P-assertion Identifier within the context of a specific interaction (Interaction Key). For example, an actor receives a message which is an interaction: before processing the message, the actor submits an Actor State p-assertion about itself, in addition to an Interaction p-assertion. The

actor's internal state could be represented using a similar *content* and *document style* data structure. However, the exact representation is highly application-dependent so no generic data structure is provided.

Relationship p-assertions: these are used to represent a uni-directional flow of a process within an actor, mapping inputs to outputs via some process. A triple can be used to record the *subjectId*, *relation*, and *objectId*. The subjectId references a *subject*, e.g. a single data input. The objectId identifies each *object* that is produced from the input (subject), e.g. several data items (objects) may be created from a data input (subject). The objectId should be paired with a parameter that identifies the operation that was performed on the input (subject) to create the output(s) (objects). Thus, the relation of the tuple simply maps sources to objects, and the process description is provided by the object description for an output.

Actor-side libraries for provenance

Actor-side libraries are needed to facilitate the provenance functions required for the different provenance actor roles: asserting, recording, querying, and management. They should be designed to enable easy integration with new and legacy systems. These libraries should support common provenance tasks, such as creating common p-assertions. They may also include facilities to help developers enforce the desired actor behaviour rules (these rules are discussed later in this section). These libraries are an important part of the architecture, but the details of their implementation are sensitive to a system's implementation. Thus, in Groth et al.ś provenance system architecture guide, the libraries are discussed as recommendations, and a paper by Jieang et al. titled "Client Side Library Design and Implementation" is cited as providing further details [63]

An actor-side library handles the communication with each provenance store interface (querying, recording and management). The actor-side libraries provide an application developer with functions for interacting with the provenance store from the provenance-aware application. These libraries should ideally shield an application developer from the complexities of implementing actor behaviours and rules.

Actor Behaviour is largely *voluntary* as this architecture can identify types and expected behaviours, but cannot force an actor to behave as specified. To do so would add additional overhead to the system which may be excessively inefficient. Therefore, a limitation is acknowledged and accepted that actor behaviours are described as constraints that must be followed to enable the

correct recording of p-assertions. If actors conform with the constraints then execution provenance will be captured correctly, and subsequent provenance questions can be answered. A set of *Architectural Rules* are used to inform a designer of a provenance-aware system as to how actors should behave.

The P-assertion Recording Protocol (PReP)

Groth and Moreau presented PReP in a separate paper called 'Recording Process Documentation for Provenance' [48]. PReP defines the communication and expected behaviours of *actors* when recording *p-assertions* in a distributed system. The provenance recording process produces a *process document* that is stored in a *provenance store*. A process document is constructed from several p-assertions, and each p-assertion represents a small step in a system's process that must be recorded. PReP defines how this recording process should occur as it is important for producing high-quality provenance that has the characteristics listed previously: Immutable, Attributable, Autonomously Creatable, Finalizable, and Process Reflecting.

The protocol allows p-assertions to be recorded to a provenance store at any time; recording occurs asynchronously with the interactions/process being documented. All p-assertion messages are *stateless*, meaning that another actor (e.g. recording actor) can understand and act upon a message without prior information. This enables messages to be recorded on arrival, even if they arrive out-of-order or relate to unfinished interaction records. In addition to the p-assertion data being self-contained, all messages include the interaction key that links p-assertions to a specific interaction.

A provenance store can receive any number of record messages containing p-assertions from multiple actors. P-assertions from an actor should always be unique in the context of an interaction, as the local p-assertion identifier and view separate them. Once the provenance store has recorded a p-assertion it cannot be overwritten or modified; asserting actors cannot overwrite their assertions or retract them. However, should an actor submit a duplicate p-assertion the provenance store should not record the assertion, blocking an actor from overwriting a store p-assertion and preserving the idempotence [2] of the PReP protocol.

Finally, the protocol *terminates* to prevent actors from indefinitely recording p-assertions about a single interaction. Termination is supported by *submission finished* messages where an asserting actor notifies the provenance store of the number of p-assertions that relate to an interaction. The

[2] *Idempotence* is the quality of something that if used one or multiple times has the same effect.

timing of this message (i.e. whether it should be sent before or after the p-assertions) is not strictly defined. The *submission finished* message gives the provenance store a way to check if an actor has finished submitting p-assertions for an interaction. Upon receiving a p-assertion message, the provenance store should respond to the asserting actor with an *acknowledgement* message. The emphasis is on the asserting actor to be responsible for tracking the success of p-assertions being recorded. Thus, a provenance store and asserting actor should be able to establish how complete the recording of an interaction is, by comparing the count of acknowledgement messages and the number of p-assertions reported in the *submission finished* message.

Provenance store

The provenance store is central to Groth's architecture of a provenance system. It is expected to persist the provenance data beyond the life of the system whose execution provenance is being captured. There may be more than one provenance store, to manage a high volume of provenance information. The architecture prescribes no specific technology, but as an example, a large provenance store might use a distributed database system that can scale up easily.

P-structure defines a structure for storing the p-assertions in a provenance store. P-assertions need to be organised and related in a way that enables them to be queried or retrieved from the store. A hierarchical structure with *Interaction Records* at the top level is used. Each interaction record contains all the associated p-assertions and identifiers relating to an interaction. This is achieved using the interaction keys from p-assertions: creating these interaction keys and ensuring they are globally unique is left up to the asserting actors. Within the interaction record, p-assertions are further sorted by View Link, i.e. p-assertions are grouped by the sender's view or receiver's viewpoint of the interaction.

Scalability Architecture provides a set of recording patterns that can be used to address the need for a provenance store to scale.

- **Separate Store pattern**: a provenance store is separately deployed from the system to preserve provenance information after a system has run so that it is available for querying. The store can be provisioned on separate resources from the system, which may not have the capacity to store the p-assertions. See Figure 2.9.

- **Context Passing pattern**: when two or more actors are interacting, they may need to share

some provenance *context* information. This context should include IDs that both actors use when referring to the interaction in their p-assertions. Thus, when the two actors' p-assertions are stored in separate provenance stores, both stores contain a common ID for the interaction. See Figure 2.10.

- **Shared Store pattern**: several actors can share a common provenance store. There is no requirement to use a different provenance store for each actor making p-assertions. See Figure 2.11.

One or more of these scalability architecture patterns could be applied to a system (Figure 2.12), as different actors in a distributed system could record p-assertion to different provenance stores. The need for this may be for network locality to actors where the type or volume of processing an actor is performing would overwhelm a limited bandwidth between sites. The flexibility to apply one or more of these patterns multiple times when deploying provenance stores aids scalability.

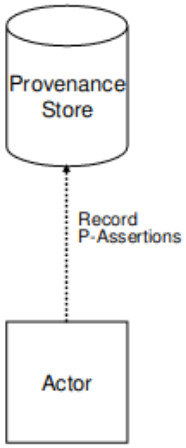

Figure 2.9: Separate Store pattern [52]

A provenance store also needs to provide interfaces for servicing provenance data:

- **Recording Interface** functionality is based on the P-assertion Recording Protocol (PReP) [48], and defines how actors should record p-assertions.

- **Management Interface** is necessary to enable a provenance store's administration and maintenance. The architecture does not prescribe the requirements for a management interface, as generic data storage technologies can be used for a provenance store. There are no proven-

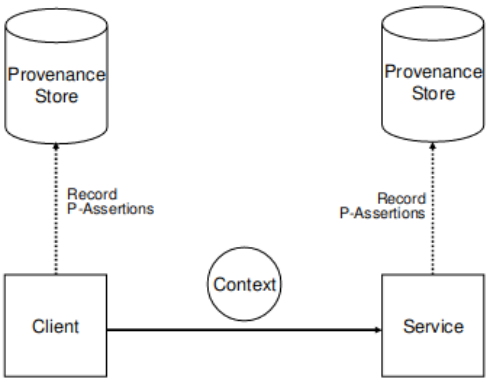

Figure 2.10: Context passing pattern [52]

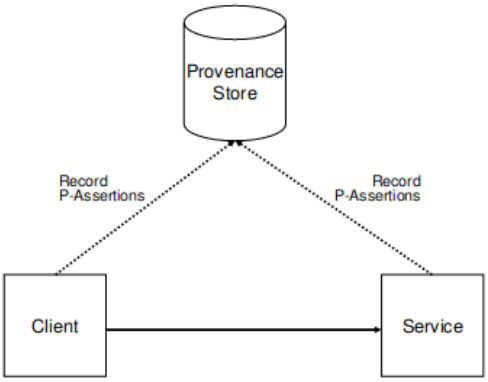

Figure 2.11: Shared store pattern [52]

ance storage-specific issues that need to be addressed but some features are suggested that could be useful e.g. notification of usage, setup and management of index.

- **Query Interface** is needed to provide access to process documentation via queries for p-assertions. An actor searches process documentation using queries which retrieve parts of the p-structure in a provenance store that relate to a process or entity of interest. The following query scopes should be provided: i) to retrieve data for a single p-assertion (global p-assertion key), ii) to retrieve all p-assertions for relating to an interaction (interaction key), and iii) to retrieve p-assertions relating to an interaction with additional filtering to an actor or view. A provenance store is only expected to return results for the p-assertions it holds. There should be a facility for allowing iterative retrieval of results in chunks, as a query may return large amounts of data. Several existing query languages can be used for this purpose and particular applications (systems) will have different requirements, thus, the query interface is not fully or

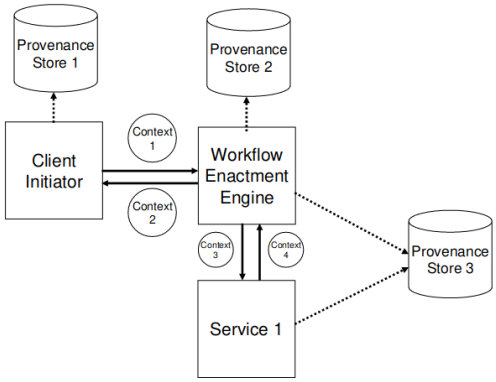

Figure 2.12: Storage patterns can be combined [52]

strictly specified.

2.5.2 Methodology for applying a provenance system

Having built a provenance system to record provenance from another system, i.e. make a system provenance-aware, the provenance awareness research moves to formalise the application of provenance awareness to another system. A methodology can be used to guide developers through the process of making a system provenance-aware. The literature reviews in this section discuss the gathering of provenance requirements, identifying sources of provenance information within a system, and adapting a system for provenance-awareness.

PrIME: A methodology for developing provenance-aware applications (2011)

Using the guidance provided in the architecture for provenance systems (Section 2.5.1), it is possible to create a provenance system, and use it to make other systems *provenance-aware*. However, when making a system provenance-aware, there is a challenge to define what provenance should be collected from the system. A system has a virtually infinite amount of provenance that could be collected, as recognised by Groth.

PrIME [77] is a methodology that assists with identifying and understanding what provenance needs to be collected from a system, to address a set of provenance requirements. The methodology provides support for soliciting provenance requirements from users. SOA design views of the system (based on actors and message interactions) are used in the PrIME methodology to identify where provenance information can be collected for each provenance requirement. Finally, PrIME provides

guidance on adapting a system for use with a provenance system to record the required provenance information.

Not all systems are based on SOA, so their design documentation may not include an SOA design view. PrIME acknowledges this problem, and guides the mapping of the actor and message interaction concepts seen in an SOA system to a non-SOA system. This mapping helps to generalise PrIME and the provenance system architecture (also based on SOA) to other system architectures. PrIME is applied in three phases to create a provenance-aware system:

1. Provenance question capture and analysis
2. Actor-based decomposition (required for non-SOA systems)
3. Adapting the system (or application)

Each of the phases for applying PrIME is now explained in further detail.

Phase 1: provenance question capture and analysis Elicit the provenance questions that users will ask. The provenance of computational systems needs to be explained to the users, so they understand the types of questions that can be answered. It may also help to provide examples of generalised provenance questions, with examples of how a provenance-aware system could answer them. After explaining to users how provenance questions are expressed, they should be able to provide a list of (provenance) questions they would like the system to answer.

Analysis of the provenance questions is required to implement them as provenance queries. A provenance query has a starting point that needs to be identified; this is likely a data item whose origins are of interest. The query needs to be scoped to return what is considered to be relevant parts of the system processes. Finally, the query results are evaluated to establish the suitability of the answer to the original provenance question; there may be a need for additional processing or extracting more information from the system.

Phase 2: actor-based decomposition Phase 2 applies to systems that do not have an SOA design view. This phase creates the required design view to enable an SOA-based provenance system to be applied. Superimposing an SOA design view on the original system design to answer provenance questions can cause a problem. The resulting provenance records for a non-SOA system may not directly map to the original components in the system design. This is because the actors identified in

this phase of PrIME may cross-cut some functional aspects of the system, or connect distant system components (when viewed from an actor's perspective at runtime these components appear closer together).

Defining the granularity of the provenance to be collected is an iterative process. The decomposition of the system into actors affects the granularity of the collected provenance. Actors may group several system components, or a single system component may be divided into multiple actors. Actor decomposition is achieved by iterating the following steps until an acceptable level of granularity is achieved:

1. Identify initial high-level actors for major functional sub-divisions of the system.
2. Map out interactions between these actors.
3. Identify causal relationships between these actors, where a data item's production is explicitly clear as a sequence of processes.
4. Ensure the data items needed as provenance query start points exist within the interactions: if they are missing, iterate with finer-grained decomposition.

A system's major structural components might be from the initial high-level actors of a system. The methodology extends the description of an actor to further assist in identifying components as actors. System components that send data or receive data within major structural components can be defined as actors; this would decompose a single major component actor into some smaller (finer-grained) actors. Doing so would expose more interactions between actors that contain more discrete data items. Identifying the interactions between actors is achieved by following these steps:

1. Discover where an actor sends/receives data.
2. Model the data exchange as a type of message, grouping messages by kinds of content that are exchanged in the interaction.
3. For each received data item (effect), discover the other data items such as actor state (causes) that create a causal relationship for an actor's output data item. In other words, this mapping should define a causal relationship for an actor's output data item, identifying inputs and causes that would affect the output data item.
4. For each cause-effect relationship, identify the type of that relationship.

The user's provenance questions require specific data items to answer them, and these data items need to be exposed for capture. Each data item should have one or more *knowledgeable actors* whose interaction contains the data item. The following steps can be followed to aid in identifying knowledgeable actors and exposing data items:

1. For each data item, determine if it is communicated (sent/received) between two actors in the current model.

2. If a data item is not exposed, check to see if it is created and destroyed within an existing actor: if so, decompose the actor further.

3. Some data items may be difficult to expose, but could be computed from more accessible data items; in some cases, it may be appropriate to collect accessible data items and compute the required data item for an answer.

4. A system may contain inaccessible data items that cannot be computed: these instances need to be documented as 'not currently accessible within the system'.

Hidden Actors may be encountered: these are components of the system which cannot be adapted for provenance collection. There may be a number of reasons why a component cannot be adapted, and this can be a problem when it is a knowledgeable actor for a data item. For some data items, it may be possible to collect the data item from another adaptable actor: for example, the other actor in the interaction. It may be that the hidden actor is the only knowledgeable actor for a data item, and also needs to be documented as 'not currently accessible within the system'.

Ideally, the SOA design view of the system should expose all the data items required for answering the user's provenance questions. However, there are three reasons why it may not be possible to record the needed provenance of a data item:

1. The actor decomposition of the system components is not sufficiently fine-grained to expose the data item.

2. A data item is never present in the system, i.e. it may only be outside of the system on paper.

3. Part of the system cannot be adapted for provenance collection (hidden actor), and is the only place a data item could have been captured.

Phase 3: adapting the system (or application) The final phase involves adapting the system to record provenance data using a provenance system. This provenance system provides storage for the provenance data, and enables the provenance queries that answer the user's questions to be executed (Section 2.5.1).

In the previous phases of PrIME, some data items have been identified. A system designer needs to determine where in the world these items are, and which actors can be modified to record them. The required modifications to an actor for exposing the data item must be specified. In some cases, this may require the designer to introduce new actors or interactions into a system design or extend current ones to include new data items. In addressing hidden actors outside the system who hold knowledge (data items), a more extensive change to the system's processes may be required; for example, an existing paper-based process may require computerising to expose the hidden actors and data items.

Finally, *wrappers* are applied to the relevant knowledgeable actors (system components) inside the system. These wrappers will contain suitable functionality to record provenance data for the data items needed to answer user questions. The wrappers conform to the principles discussed in the provenance architecture: a wrapped actor must behave as an asserting actor, and record p-assertions for its internal states, relationships and interactions. Client-side libraries from a provenance system can be used to provide the provenance recording facilities for the wrapper implementation.

In summary, the PrIME methodology approaches applying provenance awareness to a system over three phases:

1. Identify the use cases for provenance collected (question to be answered) and the data required for the provenance use cases (to answer the questions).

2. A system/application is decomposed into a design view made of actors and messages; data flows in the system are represented as messages passed between actors. Knowledgeable actors can then be discovered, these handle or contain the data for a provenance use case.

3. The system components represented as a knowledgeable actor can then be adapted to expose the use case provenance data for recording.

PrIME requires the identification of the provenance use cases before a system's provenance awareness is designed or applied. However, the methodology only provides high-level advice on how

these provenance use cases could be discovered. Broadly, the approach is to discuss the concepts of provenance with users and provide them with some examples of provenance questions. Failing to capture sufficient provenance data will mean that some questions of provenance cannot be answered; poorly collecting the provenance use case information could negatively affect provenance-awareness, resulting in user questions not being answerable.

Classification of provenance questions (2013)

The provenance system architecture from Groth et al. (Section 2.5.1) identifies how provenance can be recorded, and that an infinite amount of provenance data can be recorded. PrIME tempers the recording of provenance by having a developer identify some specific provenance use cases (questions) that a user requires. The PrIME methodology provided basic guidance for soliciting the use cases for provenance from a user. However, this could result in over- or under-recording of provenance information: too much provenance increases the runtime costs of a system unnecessarily, and too little provenance will not meet the use case requirement. A greater understanding of provenance requirements is needed to achieve a good balance of provenance data recording and the trade-offs with runtime costs for recording provenance data.

Understanding a user's provenance requirements and the resource implications for recording the provenance is important for designing a system with effective provenance-awareness. Such a system would answer all the user's questions with minimal resource overheads and no 'wasted' run- or design-time resources. A paper by Zerva et al. [116] provides *provenance question categories* for SOA systems. These categories group *kinds of provenance data* that answer *realistic questions*. Each category is broadly evaluated in the paper, identifying the kind of provenance data that can be recorded, the runtime resource implications and the provenance-awareness properties that can be expected. The paper is working towards an approach to designing provenance awareness through categories of questions: the paper terms these as *facets for provenance awareness*.

While the paper by Zerva et al. [116] is early work, it provides 9 facets for organising provenance questions for SOA systems. A user's provenance questions about an SOA system should fall into one of the following facets:

1. **Service and provider identity** questions about the identity of something on a network.

2. **Data flow** questions for inputs, outputs, and processes on data.

3. **Resource and physical deployment** questions relating to what was running a service on the network.

4. **Time** questions like when did an event/data/processes occur, or similarly duration-related questions.

5. **Routes not followed** questions are seeking to explore the alternatives at some juncture in a systems process.

6. **Past history** questions examine provenance over multiple executions of the same process, e.g. "how often does the system produce this output when running this process in a day?".

7. **Non-functional properties and Quality of service** questions may be asked if a system design includes high-level abstractions for them, e.g. "what was the average wait time for a response to a request?".

8. **Actors** or the responsible parties that might own/manage/monitor parts of the system or a service connected to the system.

9. **Design information** questions about how a service or system works in the context of design; are answered using design time documentation.

When organising questions into facets, Zerva et al. [116] make a distinction between *real question* and *provenance question*. A real question is more of an everyday type of question a user might ask, and these might require data from more than one facet of provenance. To answer a real question like "Why is the system running slow?", several provenance questions need to be asked, and these can relate to different facets from the list above. For example, provenance questions for process duration (time) might be asked to find examples of slow processing (facet 4). Then, the resources of the physical deployment (facet 2) reveal the health or CPU/memory of a node hosting a service. Thus, facets can be complementary and two or more facets may answer more real questions than the sum of their facet of provenance questions.

Zerva et al. [116] also argue that the recording of provenance data for each facet would likely impose different overheads on the runtime resource costs, although they do not validate those arguments empirically. The list highlights the different runtime cost implications a designer should consider. For example, "design information" questions require no runtime resources, only the

provenance information recorded during the design-time activities. Data for a system's past execution history could have a significant storage impact if the data is detailed and held for long periods.

Implementing provenance-awareness with Provenance-Templates

Provenance-Templates are proposed in a paper by Moreau et al. [82] as an aid for developers to implement provenance-awareness in an application or system. Neither the provenance system architecture nor PrIME provides a developer with tools for performing practical implementation tasks. Provenance-Templates are a more recent toolkit to fill this gap, which is based on other previous toolkits: ProvToolbox [80], and ProvPy [58].

How provenance is collected (instrumentation of code, or retrospective log processing) will depend on the specific details of the system's implementation. But, regardless of this diversity, there are common tasks involved in record provenance. This is where Provenance-Templates aim to assist and build on the prior tool kits. Recording provenance requires generating the provenance documents (in memory), and then serializing them for transmission and storage (as files). ProvToolbox [80] and ProvPy [58] can be used by a developer to record provenance, but the developer needs to write code to assemble the provenance representations (graphs) which requires them to know the technical details of provenance.

Provenance-Templates are proposed to separate the responsibilities of a developer and a *knowledge engineer*, to address some of the challenges with implementing provenance-awareness. A knowledge engineer designs and maintains a set of Provenance-Templates, which represent the various provenance documents that need to be produced. These templates are then used to produce a set of *bindings* that a developer will use to record provenance at the required points in code. The bindings reduce the developer workload by providing the provenance building functionality and serialisation: the developer simply needs to pass in data (variables). A knowledge engineer can update or change the templates as a system evolves: new bindings are provided to a developer, who then updates the code to align it with the new data requirements.

The use of bindings to handle provenance recording relaxes the limitation of what programming languages a system is developed with; Provenance-Templates do not explicitly require any particular language to be used. Furthermore, bindings produce their provenance documentation using PROV notation directly in a *template expansion*: the paper uses JSON for this. Template expansions provide an intermediate representation of provenance to decouple the bindings from the specific types of

PROV serialisation (notation) a provenance store might use. Additionally, template expansions might be more convenient for a system to consume itself at runtime; a system might have a feedback loop based on its knowledge of the past that is read from the template expansions. For long-term provenance storage, template expansions can be transformed into other forms of provenance documentation (PROV formats) at any time, making the data compatible with other existing PROV querying and storage technologies.

Provenance-Templates and the mentioned toolkits give some insight into the challenges with implementation of provenance-awareness, and the additional overheads with the design/maintenance of a system. The crafting of provenance records in code is discouraged in this paper, as developers may be unfamiliar with provenance recording intricacies which might result in errors. There is also a level of complexity in managing the recording of provenance that might not be expected. Separating the work between a knowledge engineer role and developer spreads the workload which would be required when developing a larger system. The knowledge engineer is responsible for mapping the system concept to the provenance recording templates (bindings), the developer then aligns the variables in code as required to the same. A developer does not need to change the code to build or serialise the provenance recordings when aligning the code to the template.

2.5.3 Provenance awareness for explanations

This last subsection of provenance-awareness reviews the literature connecting provenance-awareness with designing explainable systems. Applying provenance awareness to a system enables *provenance-based explanations* to be created for a system's execution. Creating provenance-based explanations requires carefully establishing *explanation requirements* that identify what needs to be explained and how. The explanation requirements point to the provenance information that must be collected to provide suitable explanations. Further understanding of the explanation requirements for provenance-based explanations leads provenance-awareness research towards proposing a methodology for designing systems to explain their execution.

Explanations for automated decision-making with provenance: PLEAD

Building on the prior works presented in this section of the background, the most recent developments occurred in 2019 when Huynh et al. published provenance-based explanations for automated

decisions [60]. This paper is significant as it connects the same ideas of concern for automated decisions and the need for explanations around the same time this thesis' research began. Automated decision-making and legislation such as GDPR set out high-level requirements for accountability or a person's right to explanation. The legislation does not give specific details for creating compliant practical applications. The authors assert that technology is part of the solution to address the concerns that legislation is highlighting, and advocate *explainable computing* is needed, which has a broader scope than just explaining 'Artificial intelligence' (AI) systems. Provenance-based explanations are asserted as an important source of data for creating the desired explanations.

In the context of GDPR, PLEAD makes a case for all provenance-related information being valuable and serving a purpose for the "data protection goals" of GDPR. The value of provenance information is seen when a *data controller* uses it to demonstrate the system is compliant with the data protection principles or regulatory requirements. Using provenance in service of providing explanations can be seen as *detective controls*: these are described as a facility for a data controller to detect or measure potential compliance issues, or a *data subject* having an opportunity to exercise a right. However, raw provenance data itself is not a suitable representation of an 'explanation' for all audiences. The paper explores the needs of the *audience* of an explanation using data controller and data subjects with a hypothetical loan assessment scenario. Both data subjects/controllers have a *rationale* for an explanation: this is either an obligation or right to an explanation of some *context*. Through careful analysis of these, it should be possible to assess if an explanation reaches an acceptable level of usefulness.

Using a hypothetical loan assessment scenario, the paper proceeds to demonstrate the application of provenance to the system and the use of provenance information for explanations. The explanations required are categorised by identifying the following requirements:

- **Audience** the explanation targets
- **Questions** the audience may have
- **Rationale** for providing the explanation
- **Examples** of the explanation that should be generated

A prototype of the loan decision system pipeline was written using Python to demonstrate provenance for explanations. PROVpy was used to create PROV representations of the provenance

of the loan decision pipeline. In the paper, provenance representations are presented for each of the various stages of the pipeline process. These are drawn as small provenance graphs (Provenance-Templates). These provenance representations are created and stored during the execution of the pipeline. While this paper does not fully explain how the instrumentation or design of the provenance was performed, it looks to be based on the aforementioned Provenance-Templates, PrIME and provenance system architecture. The resulting prototype system for the experiment is provenance-aware, and each decision made by the pipeline is documented in terms of input, process, and output.

The provenance-aware loan system produces a full provenance record of each loan decision process. These can be recalled in their native provenance form with extensive detail and data, which will likely overwhelm or confuse users. The paper identifies a risk with applications that process extensive amounts of data or run for long periods: these systems could record a large amount of provenance data. Moreau recommended that the raw provenance data should be further processed to create representations of relevant information; these could be summaries of key data, or common patterns to identify outliers [79]. Depending on the question to be answered or the explanation to be provided and the audience, the summary process should bridge the communication gap between the provenance data and the explanation audience.

Methodology for establishing explanation requirements

After the initial proof-of-concept paper was published, Huynh et al. published another paper 'Addressing Regulatory Requirements on Explanations for Automated Decisions with Provenance - A Case Study' in 2021 [61]. Furthering the work of the prior paper, they present an *explanation elicitation methodology*. This methodology was used with their case study based on the automated loan decision system. Experts from the UK Information Commissioner's Office (ICO) and legal experts from the University of Southampton were engaged in two workshops, to discover various explanation types for stakeholders in the scenario.

The methodology has four steps:

1. **Identifying user questions** using GDPR and other applicable legislation, the potential questions for the loan decision system are considered.

2. **Categorising questions** from the previous step: grouping the questions and associating them

with relevant regulatory obligations and rationale for why an explanation is required.

3. **Crafting example answers** for the question categories: the example given provides textual answers, but non-textual representations might be applicable in some cases.

4. **Identifying provenance data requirements** for the answers in the previous steps. The provenance data requirements are used to drive the design of provenance data recording requirements (provenance awareness).

Before constructing explanations, requirements for each explanation need to be gathered; the paper defines these as explanation requirements (ER):

- **ER1** Explanations should be generated from provenance recorded from the loan decision system.
- **ER2** Explanations should address one or more legal requirements from GDPR.
- **ER3** The explanation must be computationally traceable.
- **ER4** The intended audience for the explanation must be able to understand it.

Provenance needs to be recorded from the system to build explanations. The provenance recording should be designed to conform to the following provenance requirements:

- **PR1** Identify the type of data in the system used in the decision-making process, e.g. loan application, appliance, automated or human-based decision.
- **PR2** Trace outcomes back through the system, and identify the influencers.
- **PR3** The software and human actions or outcomes there needs to be an attribution or assignment of responsibility.
- **PR4** The activities, timings and contributions to an outcome need to be identified.

The methodology is demonstrated using the automated loan decision system from the proof-of-concept. A provenance model of the decision pipeline was created to meet both ER and PR requirements. PROV-DM (the provenance ontology discussed in Section 2.3.4) is used to create the provenance model as a set of Provenance-Templates. The provenance of each stage in the pipeline process is recorded using a Provenance-Template.

The loan decision pipeline contains a Machine Learning (ML) component, whose internal process cannot be directly exposed to provenance. Provenance-based explanations would require solutions from explainable AI (XAI) to expose the internal workings of AI components (Section 2.4). However, not being able to expose an ML component's inner processes can be tolerated in the context of workflow provenance. The ML component can be treated as a 'black-box' (internal processes remain unknown); provenance recording can capture the input/output data, and associate it with an activity that attributes responsibility to the ML component. This approach records the conditions of an ML component when it makes a decision. In theory, the conditions could be reproduced and the same ML component can be set up to make the same decision while being observed, which could then explain the decision process.

After the automatic loan decision process has been made suitably provenance-aware, the collected provenance can be used to generate explanations. To create the explanations, *explanation templates* are used: these templates structure the explanation per the explanation requirements. An explanation requirement question is formed as a provenance query to extract the required information from the recorded provenance of a loan decision. An explanation template is populated with provenance data using a natural language generator; this produces an answer that the intended audience should be able to understand.

A technical demonstration of the system is available online[3]. From a web browser, a user can simulate the submission of a loan application for the system to consider. Having decided on the application, the system returns the decision outcome with explanations. A user can explore the explanations by clicking on a question and revealing a natural language answer based on the provenance information. Overall the system achieves accountability for answering specific questions about its decision-making process. A systematic approach to explanations in the context of regulatory and legal requirements for providing explanations is needed. However, a systematic approach requires clear, unambiguous, definitions of the explanation requirements: what questions must be answered, what is considered an appropriate answer for the audience, and what supporting data/evidence is expected.

[3] `https://explain.openprovenance.org/loan`

Explainable-by-design methodology

To close the provenance awareness background, a set of papers is explored that provides a start towards a methodology for creating explainable systems. The work is based on prior provenance awareness research, and is part of the wider PLEAD research project. PLEAD identified that provenance-based explanations require particular provenance information to be recorded from a system. The process for recording the provenance information has so far been addressed retrospectively, by using an existing design or modifying an existing system. Huynh et al. published the paper 'Explainability-by-Design: A Methodology to Support Explanations in Decision-Making Systems' in 2022 [59]: it presents technical details for one of the three phases of a methodology that would design a system to be explainable from the start.

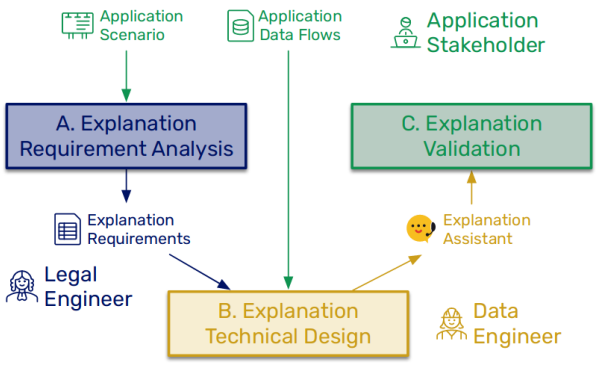

Figure 2.13: Phases of the Explainable-by-Design methodology [59]

There are three main phases to the Explainable-by-Design methodology, as shown in Figure 2.13:

A. **Explanation Requirements Analysis**: given a scenario with an application and stakeholders, the requirements for explanations should be captured. This process may require expertise in legal and regulatory matters to help identify specifically what part of a system's decision should be explained for an outcome. As described in the PLEAD work, this phase results in a set of *Explanation Requirements*, or the intended goals of the explanation system.

B. **Explanation Technical Design**: a set of technological steps that use the explanation requirements to implement an *Explanation Assistant*. This could be described as the process of making a system provenance-aware with suitable Provenance-Templates and queries: these provide provenance data that can be used with explanation templates.

C. **Validation**: having designed and created an explainable system, there is a need to ensure the

explanation provided by the Explanation Assistant meets the requirements set out in phase A. Explanations need to be suitable for the intended audience (stakeholders) and effectively provide for the intended explanation goal identified in the requirements.

Phases A and C can involve completely different contexts depending on the domain or application, e.g. a robotics application and a legal decision process have very different contexts. A follow-up paper by Tsakalakis et al. [110] (which is reviewed later in the section) provides a taxonomy of explanations which could be useful for Phase A. Huynh et al. focus on phase B (the technical design) [59], making the assumption the explanation requirements analysis has been performed and produced the following:

- **Minimum requirement context** that informs the designer of the minimal amount of information that a satisfactory explanation requires.
- **Intended audience** that the explanation is for.
- **Exemplar narrative** as an example of the text to be supplied for a specific explanation: this may be crafted by a legal expert ("legal engineer" in Figure 2.13).

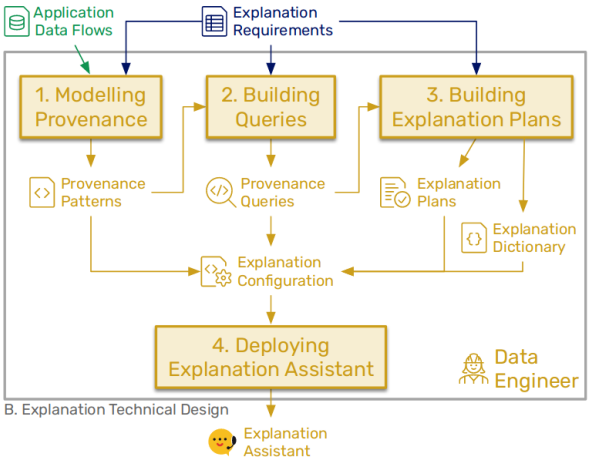

Figure 2.14: Explainable-by-design methodology: outline of the technical design phase [59]

The tasks that make up phase B of the methodology are shown in Figure 2.14. The Application Data Flows and Explanation Requirements are fed into the design process that produces the Explanation Assistant. A designer or developer performs the 4 tasks to produce the Explanation Assistant design or a direct implementation. The provenance awareness background section has explained some of these tasks already: the relevant section for each task will be referred to.

Modelling the Provenance of a Decision that the system or application is making: in this paper, *Application Data Flows* are used to represent the decision process. This modelling process goes back to the initial provenance system architecture (Section 2.5.2), and its actor-based view of a process used to record provenance. The important information or data items must be identified and found in the system for provenance recording. To identify the important information, PLEAD (Section 2.5.3) and PrIME (Section 2.5.2) should be applied for gathering explanation and provenance requirements. A designer might identify facets of provenance questions (Section 2.5.2), where overlapping sources of provenance data can provide data for answering multiple provenance questions. A data model like PROV-DM (Section 2.3.4) could be used by the designer, who can then specify the provenance representation to capture *Provenance Patterns* for the system's processes that provide answers to the provenance questions as *Provenance Traces*.

Building Queries The provenance modelling process should result in all the required data being collected. A data engineer needs to find the relevant information for an explanation in the Provenance Patterns/Traces. They then construct provenance queries to extract the required data and construct a suitable generic provenance graph representation of the extracted data for the explanation requirement. This process connects with the ideas expressed in the classification of provenance questions paper (Section 2.5.2): to answer 'real questions', answers from one or more facets of provenance questions may be required.

Building Explanation Plans was introduced in the initial PLEAD report. which identified that raw provenance alone may not be a suitable 'explanation' for an audience (Section 2.5.3). Therefore, the results of the provenance queries need to be processed to some extent. While there are different ways in which the data could be processed (e.g. provenance summaries), the example in PLEAD sought to provide natural language responses to questions. Responding to a 'real question' expressed in natural language with an answer in the same form is a likely expectation. As such, PLEAD provided the architecture of explanation templates, which would be populated with data from provenance queries using a natural language processing component.

Deploying Explanation Assistant as a composition of the artefacts from the prior tasks. Depending on the situation, there may be a need to modify the existing system or extract provenance from existing logging mechanisms. If a new system is being built, then it may be appropriate to instrument the system code (wrappers) for the provenance recording. The existing system may impose limitations such as not having source code available for recompilation with provenance instrumentation. These

sorts of issues were discussed in Phase 3 of PrIME (instrumenting a system, Section 2.5.2). Like a provenance system, the Explanation Assistant could be realised as a separate reusable component that runs in parallel to the system making decisions. This stage should include deploying the Explanation Assistant with a test system (and data) to gather samples of the explanations it produces.

The methodology presented in the paper was evaluated with two concrete systems: a credit card application process, and a school placement allocation process. Both systems are real systems with stakeholders and sensitive data: to present findings, the paper includes links to demonstration versions.

To evaluate the cost of the approach for each system, the authors present metrics for the number of artefacts produced, counting Provenance-Templates, queries, explanation plans, and so on. The time taken is presented as a brief account of the events for tasks 1 to 3 and the days spent on them. A need for 'Iterative Development' was found, as information required for an explanation was sometimes not readily available using queries or in some cases not recorded from the system.

There were opportunities to reuse some of the provenance queries as multiple explanations needed the same data returned from a query. Similarly with the explanation templates, in some cases, the same template could be used for more than one audience or system with minor modifications to the wording (e.g. 'the bank' to 'the school'). For the practical implementation of the technologies with both systems, existing components could be used and re-used; in both cases, the explanation assistant from PLEAD was used and external to the system.

A Provenance Template approach was used to record data from the system and ship values to the explanation assistant through a 'logging API'. 'Explanation Narratives' are then exposed from the Explanation Assistant with an 'Explanation API': the API triggers the required provenance queries and natural language processing routines to prove explanation on demand for the system.

Taxonomy of explanations: supporting PLEAD and Explainable-by-design

When designing a system to be explainable or to provide explanations for the decisions it makes for legal or regulatory reasons, there is a clear need to understand and analyse explicitly what the requirements are for providing 'explanations'. This need to establish explanation requirements is the first step set out in the methodology for explainable-by-design systems, but was outside the scope of the paper [59]. A pre-print presenting a Taxonomy of Explanations was published by Tsakalakis et al. [110] (which is yet to appear on a peer-reviewed venue at the time of writing): this taxonomy is

provided to help design explanation requirements.

The taxonomy provides a set of high-level dimensions for explanations:

- **Source**: the origin of the explanation requirement, for example, legal/regulatory or internal for an organisation and each requirement is either explicit or implicit.
- **Perspective**: time the explanation was generated to a triggering event, the explanation is either for what is about to happen or for what happened in the past.
- **Autonomy**: distinguishes between different explanations that are generated upfront (proactive) and those which require some input from the audience generation, e.g. in response to a question (reactive).
- **Trigger**: the event that initiates the generation of an explanation. Reactive explanations would be triggered by a user's question. Proactive explanations would trigger when the system detects that the next process requires an upfront explanation before being performed.
- **Context**: the detail of how an explanation should be formulated, as the provenance data may contain sensitive or confidential data. Thus, there might be a need to anonymise or omit/redact some of the data from an explanation when there are overriding factors. For example, data for a 3rd party in a decision has protection rights outside of the user's and system owner's right to an explanation.
- **Scope**: the reusability or applicability to an individual or group. There are local explanations which are specific to an individual in a context: the explanation is only applicable in a scenario. Universal (global) explanations are applicable across more scenarios; given specific conditions or individuals, the explanation does not change (e.g. a contact address does not change when explaining how to contact someone by post).
- **Explainability goal**: the purpose for generating an explanation as a combination of *Understandability* goals and *Intervenability* goals. These goals have further dimensions, but broadly understandability is the knowledge an explanation should impart to an audience, and intervenability is how an audience could respond or act upon the knowledge; given some explanation, the audience can intervene or exercise a right.

- **Intended recipient**: the audience the explanation is being provided for. This aspect is divided between outward-facing and inward-facing; this in/out direction is taken from the perspective of an organisation.
- **Priority**: producing the explanation may be mandatory or discretionary. A mandatory priority will often correlate with a legal or governance framework requirement.

The explainability goal has two more extensive dimensions that break down into smaller goals. There are 11 goals of *Understandability* identified in the paper (covering the type of knowledge an explanation is intended to provide), and 12 goals of *Intervenability* (related to what a recipient of the explanation is expected to be able to do in response to the knowledge provided).

Chapter 3

Initiating the research project

3.1 Motivation

This chapter discusses the motivation for researching an approach to enable systems with a form of self-explanation or why anyone might want to use such an approach. The topic is wide and there are many unique and interesting ideas to chase, so the chapter provides a summary at the start. The further sections discuss motivations at different levels, curious desire, society's call for regulation, and emergent surprises.

Summary

Systems with autonomous decision-making are becoming increasingly important for society, which needs explanations about their behaviour; one solution to the problem would be to enable a system with a form of 'self-explanation'. To build those explanations, we need a system to be able to answer some facets of provenance questions. Some facets of provenance questions can be addressed with a general approach to provenance awareness, such as data flow and past history. In contrast, other facets require provenance data which contains some specific system concept or domain knowledge. This presents a problem when trying to create a generalised approach to provenance-awareness as a framework or tool, not a methodology. Model-driven engineering and runtime models are examples of how model abstractions can be created at design-/run-time to represent system concepts or domain knowledge; these knowledge models capture some facet of information about how a system behaves or makes decisions. As an alternative to existing provenance awareness approaches, a generalised approach that applies provenance awareness to the knowledge models within a system may address

some facets of provenance questions that are system or domain-specific: this data could then be used to create provenance-based explanations for some real questions about the system's behaviours or decision-making.

3.1.1 A desire for a computer to explain its process

In the background section 2.4 some examples from the literature show past work that sought to enable computers with a form of self-explanation. The desire to have a computer that can communicate with a user what it did or will do in the form of an 'explanation' has existed for a long time. The motivations for enabling a computer system with self-explanations vary over time as the focus of computing science moves around. The approaches taken to self-explaining systems also change depending on the motivation, often the approach is specific to a type of system, domain or problem.

The meaning or definition of an explanation is broad and interpreted differently, this is a challenge for the philosophical literature to define 'what an explanation'. This thesis needs to frame the kind of explanation or self-explanation features for a computer system that motivates the research. Many different aspects could be explained about a system's size, function, architecture, data etc. The explanations of interest for this research are the ones that convey some information about a system's action/processes at runtime.

Think of a computer system as a black box, in which data can be input and some output data is returned, what happens within the black box is unknown. When a user inputs some data to the system they may have some expectations for the output data; if expectations are met with convenience and ease the user may never be motivated to understand what happens inside the black box. However, when a user's expectations do not align with the output data there may be a need to reconcile the difference between expectation and reality. One aspect of reconciling the difference between expected and actual output data is to offer the user some information about what happened inside the black box. The information offered could be considered an explanation if the system has a function to provide the information, that function enables a system with a form of self-explanation.

For a system to offer information about what happens inside the black box there is a need for a system to be aware of the contents. The types of models, symbols, or representations in the system's black box are also important. Section 2.4 discussed the XAI research which addresses the complex problems with systems that contain models that need translation for humans. Conversely, if

a system contains an intelligible process and suitable representations, like the knowledge base or context awareness systems. The system might be able to reflect on its internal knowledge to produce information readable to humans. Systems like the knowledge-base or context-aware are the ones that this thesis seeks to enable with a form of self-explanation as a form of self-reflection based on the content of the system's 'black box'.

There is a question of when and how a system should provide information about its runtime activities. This thesis is looking at a post-hoc or after-the-system-has-acted type of explanation, where a user will engage and lead an investigation into what the system knows about its past. To do this a system will need to have documented the processes and data it has worked with, then provide relevant information from the documentation in response to a user's questions or queries. Intuitively a human might assume that this exchange would be in natural language, but it could be more abstract. Knowledge, information or data can be communicated in different forms and levels of abstraction, this idea is relatable to the concepts discussed in Models Section 2.1.1 of the background. A system could keep a record of its actions as a model, it would build this model itself using its internal representations, and then when asked questions it would use the model to explain or show answers.

A system using a model to describe its past actions factually might support a range of use cases where explanations are required. The motivation for a system designer to enable a system with self-explanations could be considered on a scale with requirements ranging from relaxed to strict reasons. A relaxed requirement for explanations would be for user experience, a system causes no harmful outcomes the explanations are to foster trust and acceptance of its users. These relaxed requirements might be less precise in describing the goal for the explanation a system needs to provide. The strict requirements for explanations will be expected for the systems that can cause harm to humans or the environment and there is a need for accountability or justification; these strict requirements may be enforced with laws or regulations formed by society. Depending on the motivating requirement the suitability or qualities expected of an explanation change, meaning a generalised approach to self-explanations might not fit all use cases.

3.1.2 Society needs computers to explain their automated decisions

A motivation for enabling systems with self-explanation arises from the increasing amount of automated decision-making using computer systems and the impact these systems are having on society.

Artificial Intelligence (AI) and Machine Learning (ML) techniques can be used to create autonomous decision-making systems, which have caused several social impacts as reported by Crawford et al. [24]. A lack of accountability or access to explanations for the outcomes of automated decisions these AI systems make has sparked some regulatory reactions. In Europe General Data Protection Regulation (GDPR) [41] includes provisions for citizens to have 'the right to an explanation' when they are the subject of an automated decision.

There are vigorous discussions about what sort of explanation people are entitled to under GDPR concerning automated decisions. For example, Selbst and Powles [101] consider meaningful information about the logic involved in the automated decision process to be included, not just the data the decision was based on. To expose the logic and data within a system a designer could make a system transparent which is closely related to explainable. To do this a system designer can look to standards like IEEE P7001 Transparency of Autonomous Systems [115]. Winfield et al. proposed standards for system transparency that should be measurable and testable [115]; in this paper, they also identify stakeholders around a system have different requirements for transparency. Each stakeholder requires a different view of a system based on their motivation to look into a system to understand something.

Simply making systems transparent has further complications when AI or ML components are used as part of the decision-making process. In particular with ML, Burrell writes 'When a computer learns and consequently builds its own representation of a classification decision, it does so without regard for human comprehension." These systems present a unique challenge as they develop or learn solutions to problems, by manifesting some internal structure that defies human understanding. Providing explanations for these types of systems is the research work Explainable Artificial Intelligence (XAI), which is surveyed by Adadi and Berranda [1].

The PLEAD work (Section 2.5) takes the legal requirements challenge; an answer to society's call through regulation for explanations and accountability for automated decisions. The authors of PLEAD propose that provenance-based explanations can explain a system's decision process. In their approach, the authors acknowledge that AI and ML components will need support from the XAI community to expose their internals. However, the function of these AI/ML components in a larger workflow can be captured in the context of a larger system using provenance.

The argument raised by the authors of PLEAD is that explaining the content of AI/ML components is only part of a larger explanation for a system's decision. There is a need to include information

from the wider context of the entire system, which XAI alone does not address. Their view is that a system's decision pipeline should be made provenance-aware; a form of transparency. The provenance-awareness of the decision pipeline should be defined by *explanation requirements* as a set of specific questions or desired explanations.

PLEAD offers a holistic approach to enabling a form of self-explanation that is well-suited for a system that needs to comply with regulations. The methodology calls for the explanation requirements to be defined; what information must be included and what a suitable presentation of the information looks like. To do this experts on regulations/law should be consulted with system stakeholders to form an agreed interpretation of what is required. By understanding what is required as a suitable explanation of the system,'s processes, it is possible to locate, expose and record the provenance data to support explanations. Without these well-defined descriptions for the explanation requirements, it is hard to assess if a system's explanation reaches the required level of compliance.

3.1.3 Understanding run-time surprise with a computer's explanation

This section looks at a curious problem with surprise and emergent behaviours when a system at run-time does something beyond the design-time assumptions. In the previous two sections, the concept of a computer explains what it was doing as an explanation and the serious view from society on wanting explanations for automated decisions. In both these situations, the person wanting to know what happened may not have access to some design knowledge of the internal workings of the system; this 3rd situation is interesting because even with design-time knowledge expectations and reality of the actions of a system differ.

The unavoidable complexity of some software systems and their contexts make their runtime behaviour unforeseeable sometimes. For example, concurrent systems are intrinsically non-deterministic: their runtime processes are a complex interweaving set of processes, which makes them difficult to test [11]. Another source of complexity can also come from the body of information a system uses when making decisions; knowing what information affected an outcome can be difficult [7, 98]. Overwhelming complexity can cause the design-time expectations to be wrong or underestimated, some uncertainty arises in the system design unknowingly.

Self-adaptive autonomous systems (SAS) use a form of automated decision-making to alter their behaviours, or processes to reach a function goal. These systems might adapt to overcome some

design-time uncertainty for example the system's operating environment is not fully understood. This design-time uncertainty is compensated by enabling a system with some self-adaption, this can result in emergent behaviours that have unintended consequences [112]. The emergent behaviours might not be negative; a system may achieve the intended functional goal. However, the system may be erratic or counter-intuitive, which could cause the user to have some reservations about its ability.

Now there might be a desire arising from curiosity to have a system explain its process, or a need to explain a system's processes to others in society. The motivation for self-explanations for a SAS connects with the motivations presented by Adadi and Berrada when they surveyed the XAI literature [1].

- **Explain to Justify** why an outcome occurred; the process details might be summarised or excluded from the explanation, for example, an assurance of correctness is given with some key information.
- **Explain to Control** a system by understanding the processes within a system we hope to mitigate against vulnerabilities, flaws or errors.
- **Explain to Improve** a system's internal processes by observing, understanding and continuously developing them.
- **Explain to Discover** new facts and information by learning from the process a system developed.

An interesting challenge arises for the system's self-explanation function to explain its runtime processes when those processes are not fully understood at design time. There are some assumptions about what the states in the system might be and the order in which different processes might occur, but in reality, these assumptions might be wrong. Therefore, the kind of self-explanation functionality needed requires a system to keep a record of the processes/states from its perspective; not a prescribed description based on design time assumptions.

Conceptually, a computer system would need to have some basic expressions to describe processes, states and the relationships between them. A system's overall functional process would need to be broken down into various sub-processes; these would interact with sub-sets of states or data within the system. The adaptive nature of the system may change the order of processes or

alter weights represented as states. The basic expressions a system has to document its runtime need to be able to reflect the dynamic changes that can occur as a result of adaption.

The representation of the runtime that a computer builds to document its runtime processes could be seen as a type of model. This model provides a system with an awareness of its actions or knowledge of its past, to access this information we might ask the computer questions to get answers. However, the questions it can answer will be restricted to some level of abstraction/detail that the documentation function permits (e.g. the basic expressions of processes and states have a level of abstraction). The system would only answer questions about states/processes it was aware of and documented during runtime.

3.2 Problems

People have looked to create self-explaining computer systems in the past, each with their unique approach for some purpose. Some attempts have been related to trust or acceptance, communicating knowledge from a system to a human about a system's actions that have or will happen. In some instances, a strict requirement for some information relating to a system's processes must be recorded to explain a single instance or all outputs a system produced. More recently, where a system creates its solutions or processes, the explanations seek to discover knowledge or gain an understanding of a system. These all lead to some broad questions about the processes and states within the system at run-time, or a need for the system to reflect on an awareness of its run-time, recall and present information about its run-time on request as a form of 'self-explanation'.

The problem of explaining the inner workings of the AI or ML systems is best left to XAI researchers, but these are not the only systems that need to explain what happened at run-time. As seen with knowledge-base and context-aware systems (Section 2.4), there is a need for a system to automate the communication of information about run-time actions using a system's internal representations (which a human can comprehend). However, these approaches are intended for specific system explanations like 'why a recommendation was made based on some knowledge' or 'if this option is selected in this context, the following may happen'.

Provenance-awareness describes a property of a system that is made aware of its processes at run-time and records the provenance of its execution (Section 2.5). This is a very close description of the information needed to explain a system's decisions/behaviours at run-time. PLEAD makes the

case for provenance-based explanations for systems that need to comply with legal or regulatory requirements. The PLEAD work provides the ground for a methodology for 'explainable-by-design' which extends prior work of the PrIME methodology for applying provenance-awareness to a new system.

The methodologies presented for making a system provenance-aware present two sources of potential problems for creating a generalised approach to provenance awareness for explanations. Firstly, provenance awareness was developed on top of a provenance system architecture [52] intended for use with a service-oriented architecture (SOA) system. This influences a view of system execution provenance based on actors exchanging messages to perform a process. The subsequent methodologies generalise their approach for other system architecture by having a designer create an orthogonal system design as an SOA view of the system. This second design view of the system is then used to identify important processes/states and apply provenance-awareness instrumentation to a system's components.

Creating an orthogonal SOA design view of the system introduces a separation of concerns in a system's design; a design for the system's intended function, and another for documenting provenance. This separation can be exploited to divide work between a software developer concerned with function and a knowledge engineer concerned with provenance. There is a potential for these two designs to drift out of alignment and introduce errors. As with most complex software development, the challenge is overcome with good project management, documentation, communication etc. However, requiring an SOA design view solely for implementing provenance may not be acceptable, or may discourage the adaption of provenance-awareness.

The second source of problems comes from the requirements-driven approach to designing provenance awareness that is likely not reusable. The targeted nature of the provenance awareness developed will collect a precise amount of provenance in a form that serves a purpose. Soliciting explanation requirements as part of a design process for a system that must meet some regulator or legal requirement is very important. It requires the regulator/legal body to be clear in their requirements for an explanation as a definition of data to include, suitable presentation etc. A system designer can then build a system per the explanation requirements and demonstrate or validate the explanations produced against the same.

The methodologies for applying provenance awareness give some assistance to the soliciting of explanation requirements from stakeholders. In some cases, it may be enough to introduce

provenance concepts to stakeholders and discuss how to reform their questions into a question of provenance. However, even with an extensive list of one-to-one question-and-answer pairs targeting specific parts of the system's processes, this does not lend itself to creating a generalised form of provenance awareness [116]. An approach to developing (explanation) requirements for generalised provenance awareness is to consider *facets of provenance questions* (Section 2.5), or sets of questions that might be asked around some aspects of the system.

Zerva et al. identified in [116] a classification of provenance questions and noted that the answers to those questions were interwoven and overlapped. For this reason, generalising an approach to provenance awareness based on a few specific questions is not practical. Facets of provenance questions for an SOA are examined, as well as the run-time resource cost implications collecting the required data to answer different facets, with run-time resource usage estimates for storage space (for provenance data) and performance impact (system execution slowdown). This perspective of facets of provenance data and questions is an important consideration for generalising approaches to provenance awareness and by association generalised provenance-based explanations.

Using provenance facets as a lens, we can try to frame the explanation requirements for the general XAI motivations for explanations (explain to [justify/control/improve/discover]). There is a facet of questions relating to a set of processes and states within a system that by design enables a system to produce a decision/behaviour as an output, which we want to explain to [justify/control/improve/discover]. Therefore, a few facets of provenance information need to be collected about the system. These facets include information about system execution but exclude other facets of information about things that we do not want to explain (e.g. provenance of the manufacturing processes for the hardware components).

A selection of provenance questions over several facets could be used to apply the PrIME approach to provenance awareness. The provenance collected for each question would provide data for some additional unasked questions. These unasked questions might be multi-faceted or derived from the answers to other questions. The provenance data required to answer a question is available if a system component has provenance-awareness and questions are used to establish this coverage; failing to ask the right questions could result in potentially important parts of the system not being covered.

Applying provenance awareness based on questions with PrIME also drives the design of predefined provenance representations for a system's execution. One of the tools to assist in imple-

menting provenance awareness is the provenance template; a predefined provenance representation of a process a system completes with data at run-time. If a system dynamically changes its processes it may be difficult to provide these predefined templates for a system to record provenance. This extends from the earlier orthogonal design problem, where the design of provenance runs parallel to a system design; the provenance awareness records a description of execution derived from an SOA view of the system design, not the execution of a design.

Provenance awareness and the methodologies to apply the approach (such as PLEAD or the explainable-by-design approach) provide a way to achieve explanations for compliance. This view of a system's execution being captured as provenance being a source of data for explanations is compelling. However, there is a rigidity in the approach which makes generalising the approach as a set of tools difficult. This imposes additional design-time costs, with the requirements and analysis of what/how to explain, which is valid for legal/regulatory requirements. The resulting provenance awareness is derived from a second design view of the system, which also poses a challenge if a system re-configures itself at run time.

Providing a generalised approach to provenance awareness which produces provenance representations based more directly on the execution of a system design presents its own problems. Some facets of provenance questions are going to be domain- or system-specific: these questions start from some information that is not general across all systems. The provenance question categories from Zerva et al. [116] identified a facet of data flow questions: data flow generalises as data in and out of the processes in a system. Questions relating to quality of service only apply to some systems. A generalised approach would need to accommodate a mechanism that enables a designer to tailor the general execution provenance to include domain/system-specific information; PLEAD does this by developing provenance for explanation requirements.

3.3 Objectives

The provenance awareness line of work provides a compelling argument for collecting the provenance of a system's execution to enable self-explanation in a system. The reviewed methodologies give instructions on how to adapt or extend a system design to include explanations/provenance awareness for some specific requirements. There is a challenge with adapting these approaches into a generalised/reusable tool for applying provenance awareness: such tooling could reduce the

design time costs for applying provenance awareness and explanations.

There is a challenge with a system presenting explanations or interacting with an audience seeking explanations. This is identified in the PLEAD work, which highlights provenance data may not be a suitable explanation for some audiences. There is a need for a layer between provenance data and the audience, which should convey some information derived from the provenance data as a suitable explanation. Acknowledging this point is important, as a generalised approach to provenance awareness for explanations may produce provenance data which some experts might consider 'an explanation'; others may reject it as just data and not a suitable explanation.

Society's desire for explainable systems is broad and not completely understood; this is partly evidenced in the PLEAD mythology including the gathering of explanation requirements for clarification of the explanation needed or expected from a system. A generalised approach to allow a system to explain its behaviour or decision process in some abstract form, for a low design-/run-time cost, could fill some of those requirements or aid the further discovery of what those requirements should be. However, we need to try and carefully understand what that description of an approach is trying to achieve as an objective. A system's ability to explain is the enabling of functions or features associated with the idea of a system being explainable or providing explanations (Section 2.4); for example, transparency or the ability to answer some 'why' questions regarding its processes. The explanation being in an abstract form is about the types of abstractions described in modelling (Section 2.1.1); some knowledge or information is represented with a model, using a modelling language at some level of detail.

The search space for a solution is in an overlapping space among the ideas covered in the background chapter 2. Modelling provides a conceptual view of communicating through abstract representations. Provenance (provenance awareness) is a means to keep a record of processes/states that created a product (output). Lastly, automated self-explanations using some properties of systems internals, its *self*, which might be a model containing knowledge or representations of the system.

The following research questions and sub-questions outline the objective of creating an approach to provenance awareness based on runtime models or a shared-knowledge base. These questions seek to tackle problems that arise when using the existing provenance awareness methodologies for non-SOA systems.

RQ1 *How can we implement an approach to provenance awareness for systems with a shared-knowledge base, which can answer a facet of provenance questions to support explanations for a system's decision-making (or behaviour) in terms of its execution without having to create an orthogonal design of the system to implement the provenance awareness?*

- **RQ1.1** *How can provenance data be structured and stored in a way that enables storage costs to be managed by retention of time- or event-based periods?*
- **RQ1.2** *To what extent can the reuse of the existing infrastructure for the shared-knowledge system also reduce the costs of providing provenance awareness?*
- **RQ1.3** *For which common facets of provenance questions can the collected provenance data provide some information to support an answer?*

Answering the initial research questions will produce an approach which can be applied to a limited range of technologies/systems. These limitations are likely caused by reducing the complexity of the problem and the practical implementation. However, it is also possible the approach produced by RQ1 has some dependencies on how a system's shared-knowledge model is designed or built. For example, the approach could be dependent on certain design documentation/views or particular coding practices being followed. These would also further limit the kind of systems that the approach to provenance collection could be applied to. Given the product of the initial research question, the second question sets the objective for extending the approach.

RQ2 *How can the approach generated by RQ1 be used with systems that have manually-written (less consistent) code implementing their shared-knowledge model?*

- **RQ2.1** *Using the new approach, can the results of the original evaluation experiment be reproduced, and does this approach add any new capabilities?*
- **RQ2.2** *What problems arise when applying provenance awareness to a system with a manually written shared-knowledge model, and do these problems relate to any known issues in the literature on provenance awareness?*
- **RQ2.3** *How can the approach answer a facet of provenance questions for program execution, using the shared-knowledge model and processes identified in a system with a manually-written shared-knowledge model?*

The last research question applies to all of the experiments and development for the prior questions. These questions seek to provide some knowledge of the approach's effects on system design- and run-time costs. Presenting these findings is important for others, while the data might reflect a limited example in the context of experiments on small case-study systems. Having some data enables others to consider the trade-offs or suitability of the approach for their system/use case.

RQ3 *What new knowledge of provenance awareness costs was discovered by the design and development of an artefact to answer RQ1?*

- **RQ3.1** *What are the design-time activities for using the new approach, compared with existing provenance awareness methodologies?*
- **RQ3.2** *What runtime overheads were seen on the systems used to evaluate the implementation, and how do they compare to the results published for provenance awareness?*

Design time and run time definition

There will be many references to design time (design-time) and run time (run-time): these terms are defined or refer to the following activities.

Design time Refers to time that is spent creating or maintaining a system. In the process of creating a system, many activities take place. Some or all of the following may occur: the gathering of requirements or production of a specification, the creation of system models or design views to communicate a system's design, and the developers writing and testing the system's code.

Applying provenance awareness to a system increases the design time costs of a system; one or more of the above design time activities requires more effort or work to be performed. For example, if a system does not have an SOA design view, then an additional design view would need to be created, which is more work in the modelling/design view stage of the design time process. Similarly, adding provenance instrumentation to a system would likely involve writing additional lines of code. If these lines are manually written by a developer, then the programming work is increased.

Run time When a system has been created, at some point, it will be deployed into an environment and run; this is a system's run time. Run time is when a system is executing and in operation.

During run time, a system requires some computing hardware resources to operate: providing these resources has a cost. The cost of providing the resources is described as increasing or decreasing if more or fewer resources are required. A lean system, which requires very little processing power, memory and storage has a low run time cost. Conversely, a system that consumes large amounts of processing power, memory and storage has a high run time cost.

Chapter 4

Design and development: shared-knowledge system provenance collection concept

Provenance awareness or provenance collection can be implemented using PRIME, but this approach requires an SOA design view of the system. An SOA design view enables the identification of specific actors and messages in a design view of the system. A set of explanation requirements are then applied via this SOA design view. A shared-knowledge system's design may not have this SOA design view, representing the system as a set of actors/messages. To reduce design time costs for implementing provenance awareness, an approach that uses the system's existing design views (models) would be advantageous. In this chapter, a conceptual approach to provenance awareness is designed for use with a shared-knowledge system.

4.1 Conceptual design of shared-knowledge provenance collection

4.1.1 Motivation

Systems can be made provenance-aware by following the PrIME methodology: this can be used to enable a system with a form of self-explanation, using provenance-based explanations. However, applying PrIME requires an SOA view of the system which can result in additional design-time costs to maintain more design documentation. The SOA view of the system is needed to enable the design

processes of provenance awareness. The provenance models produced for execution are based on actors exchanging messages to interact with each other, which is one way to view execution in an SOA.

It is desirable to reduce the design-time costs of provenance awareness in systems that employ shared-knowledge bases (e.g. runtime models). Creating and managing the SOA design view adds to the cost of designing/maintaining a system with provenance awareness. A new approach could be designed that reuses more of the existing system design, to enable provenance awareness without creating a separate orthogonal design.

Certain facets of provenance questions will only apply to some systems: these are probably questions starting from a system architecture feature or domain concept. PrIME implements a provenance awareness design that provides coverage for these facets, by adapting or wrapping the design of system components. This approach does not directly reuse the existing system design. An approach to provenance awareness that derives the facets of system-/domain-specific provenance from a system's design-time or runtime models could be implemented as a set of components that is reusable between systems, offering further value for the investment in design costs.

The motivation to reduce the design costs of provenance awareness also reduces the scope of the approach to a selection of systems. At the moment, provenance awareness generalises to other architectures by imposing the SOA view of execution as message exchanges. To offset the loss of some system architecture generalisation, some other benefits or additional reductions in design costs might come from developing a view of execution specific to shared-knowledge systems.

A shared-knowledge system is designed around processes that read and modify a shared model to achieve a goal. For example, a MAPE-K system architecture (Section 2.1.4) has MAPE processes interacting with a K model to share information. A view of execution for this type of system could be the evolution of its runtime model. System agents evolve a runtime model by performing activities (MAPE) that use/change the model (K); this is the execution a provenance model needs to reflect.

Assuming the system output (behaviour/decision) is a product of processes interacting with its runtime model, then the provenance should answer a facet of questions (Section 2.5.2) about the origin of the system outputs.

4.1.2 Problem

An approach for applying provenance awareness for shared-knowledge systems (using a runtime model architecture) that can remove the need for the SOA view could reduce some design-time costs. However, removing the SOA view requirements presents several problems, as the view influences both the methodology and the architecture for provenance awareness.

The provenance system architecture provides a provenance model based on the SOA view of actors exchanging messages to perform processes. Actors in this case are systems that exchange messages, and these interactions are captured in a provenance document. The provenance document can be structured through a provenance template at design-time, and populated with the data about each interaction at run-time. The current provenance model would require a shared-knowledge system to be built using SOA or an SOA design view.

One solution for using the SOA provenance model would be to prescribe a general SOA view of model interactions in a shared-knowledge system. This view would need to provide a generalised design that describes model interactions as two actors interacting through a sequence of messages. However, this approach may create misleading provenance representations; for example, the 'actor' representation of a system process interacting with a model implies that the process and the model are both actors with responsibilities, like a system providing a service. A more accurate view of the runtime model would be a passive data store with no responsibilities.

The SOA design view of the system in the PrIME methodology is important in designing where provenance awareness is applied to a system. As part of a system design process, some explanation or provenance requirements are established to determine the provenance data to be recorded. The location of the provenance data in the system is discovered by a process of actor decomposition, using the SOA design view. An approach that does not create an SOA view will need to provide a replacement for the design processes in PrIME that identify and control where provenance awareness is applied to a system.

A reusable tool set for an approach to provenance awareness would also help to reduce the costs of applying provenance awareness to a system. However, the later provenance awareness literature [116] identified a challenge with creating generalised approaches to provenance awareness based on specific questions. The same paper [116] proposes the idea that provenance questions exist as facets of questions; there is a complex many-to-many relationship between provenance data

and questions with overlapping dependencies. The provenance data collected to answer one question may also answer or contribute to answering other questions. Thus, the existing methodologies based on collecting provenance data to answer specific questions can not be directly applied to the creation of a reusable tool set for enabling a generalised approach to provenance.

4.1.3 Objective

The objective is to create a new provenance model for the shared-knowledge systems that use a runtime model-based architecture, which is common for SAS (Section 2.1.4). The provenance model will drive a new generalised approach to provenance awareness for these shared-knowledge systems. At this stage of the development, a conceptual design of an approach should be created that can answer the following local research questions:

LRQ1 *How can we model the provenance of processes interacting with a runtime model of shared-knowledge?*

LRQ2 *How can the new provenance model capture domain-/system-specific facets of provenance, without a separate design view for provenance awareness?*

The answers to these local research questions contribute some answers towards RQ1 of the thesis (Section 3.3):

RQ1

How can we implement an approach to provenance awareness for systems with a shared-knowledge base, which can answer a facet of provenance questions to support explanations for a system's decision-making (or behaviour) in terms of its execution without having to create an orthogonal design of the system to implement the provenance awareness?

4.1.4 Design and development

To develop the approaches for conceptual design, an assumption will be made that the system follows the MAPE-K architecture (Section 2.1.4). This abstraction helps to design a provenance awareness

approach, while mitigating against inadvertently creating dependencies on specifics of the system or domain.

MAPE-K knowledge bases need to contain representations for the domain concepts or system components that are important for the intended function (goal) of the system. The system's physical sensors and actuators must be modelled in its Knowledge base ('K'), for the system to be aware of them. Modelling a sensor can enable the system to access other aspects of the sensor beyond its data signal. For example, a sensor may have an adjustable sensitivity range, which a system could reconfigure to get more fine-grained data readings; additionally, the model may include identifiers to map them to the physical sensor. The same applies to actuators, which may have similar adjustable settings for power or precision. Finally, the knowledge will be treated as a runtime model, maintaining a causal connection between the model and the physical system created through the MAPE phases which synchronise them.

Walking through the MAPE-K execution of the system, the following sequence occurs in a continuous loop:

- **Monitoring** of the environment is performed by the agent, by reading a set of sensors and recording the data into the knowledge base (the sensors must have model representations in the knowledge base).
- **Analysis** of the data collected from the sensors occurs, using some configuration in the knowledge base. Model representations for some domain concepts are then updated, which represent the situation the system is in (e.g. 'too hot', 'too cold').
- **Planning** is then performed to determine how the system should react to the situation: this may be an exploration of possible outcomes for different strategies. A plan of action is created in the knowledge base.
- **Execution** of the plan occurs: the last process uses the system's actuators (modelled in the knowledge base) to interact with its environment and effect some change in the physical world.

For simplicity in this specific example, we assume a single agent (actor) will perform the 4 processes (activities). These processes interact with data (entities) stored in a shared-knowledge base, implemented as a runtime model. The provenance system architecture [52] tells us that an (asserting) actor should make provenance assertions (p-assertions) to report events from the

perspective of the actor performing the process. The assertions can be structured using PROV-DM (Section 2.3.4).

Provenance model elements

The old names in the provenance system architecture (Section 2.5.1) map to the recent PROV terms (Section 2.3.4), and describe the following features of a MAPE-K system execution:

- **Actors map to PROV Agents**, who are responsible for performing processes (activities) in an execution.
- **Processes map to PROV Activities**, which define a process being performed (executed) by an agent, that may use/change the shared-knowledge.
- **Data items map to PROV Entities**, the values or states in the shared-knowledge base (assumed to be a runtime model) that a process used/changed.

Having mapped the terms, a more detailed description is provided below for each PROV-DM term, to relate them to the provenance model of the provenance system architecture.

Agents The SOA provenance model considers agents to be system nodes in the network, where each system node could be owned by a different person/organisation. In this context, the responsibility represented by the AGENT at a system level is a suitable level of granularity to identify a system node and its owner. Execution in SOA is viewed as a sequence of interactions between system nodes. However, the execution layer being modelled for MAPE-K is at a finer level of detail; this could be thought of as execution that occurs within an SOA system node, that would act upon the content of a message and perform the work for the service advertised.

A single AGENT (representing the entire system) does not provide sufficient granularity for the provenance model of the MAPE-K execution. There is a need to attribute responsibility for each process/model access occurring in the system. For example, a MAPE-K system might adapt in response to a user interaction, and the provenance of the adaptation needs to attribute some responsibility and activities to the user. Another feature of a MAPE-K execution that needs to be considered is how software executes. It is common to have software run multiple concurrent execution threads. A system execution can be made up of multiple concurrent processes: responsibilities for these processes can be assigned to the software thread of execution.

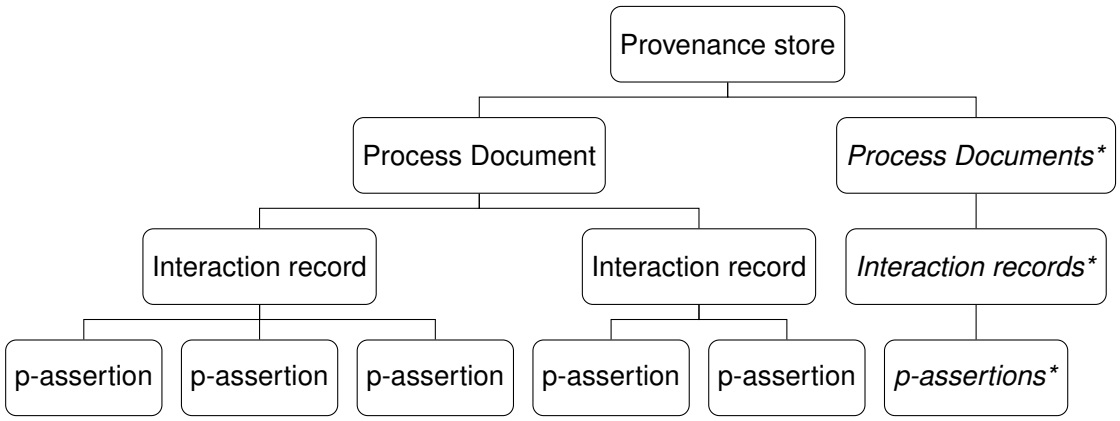

Figure 4.1: Tree model example of the p-structure in a provenance store, see background Section 2.5.1

Activities In the original SOA approach, the execution of an activity occurs between the input and output of data. This description fits different levels of detail that a provenance model can represent in a provenance store. As shown in Figure 4.1 (page 96), a provenance store could be viewed as a set of documents that describe processes applied to some input dataset to produce an output dataset. This can be applied recursively to more fine-grained process documents containing interaction records, where each interaction record is a collection of p-assertions. The p-assertions are the fine steps in a process, and include descriptions of the data interactions involved.

Interactions are identified as a unique exchange of messages between two actors. P-assertions are made in the context of an interaction, and the p-structure organises these into interaction records (see Figure 4.1, and the concept of p-structure in Section 2.5.1). The type of p-assertions recorded depend on the relationship between the data and interaction (the concepts of interaction, relationship, and actor state, are explained in Section 2.5.1).

In the execution of a shared-knowledge system, an agent accesses the runtime model, performing reads and writes. These accesses occur within the context of an activity, so an agent would make assertions (p-assertions) for each access of the runtime model in the context of an activity.

Agents in the shared-knowledge system start an activity, which may access the runtime model. This activity might be divided into sub-activities, which an agent might perform once or repeatedly. The execution of activities in the system would be similar to the execution of a program.

A code function could be considered an activity, and functions can call each other or themselves in an iterative loop. Thus, activities can nest similarly to interactions in an SOA, where an agent interacts with other agents to complete an initial interaction.

Entity SOA data is passed in messages or stored as internal states of agents; both are documented as entities. Messages received by the SOA system are usually consumed and not persisted after use; the provenance system is responsible for persisting their data in the provenance records, for provenance queries. In contrast, with a runtime model architecture, data is persisted in the model until it is changed/deleted by a process. The provenance model for the runtime model needs to provide a similar persistence to the SOA provenance model. An ENTITY tracks the versions and states of the shared-knowledge data as agents perform activities that change the runtime model.

Available p-assertions Based on the earlier description of the MAPE-K execution and the above definitions of agent, activity, and activity, we can now make the following p-assertions from the perspective of an agent performing the execution:

- I am AGENT, I started ACTIVITY.
- I am AGENT, performing ACTIVITY, I wrote to ENTITY.
- I am AGENT, performing ACTIVITY, I read from ENTITY.
- I am AGENT, I ended ACTIVITY.

Provenance model element identities

The provenance system architecture says that the p-assertions should use unique identifiers (ID). If more than one agent refers to the same instance of something in an assertion, the same unique ID must be used. Ideally, IDs should be generated by an agent, to remove the time overheads that a centralised ID system would introduce with agents requesting IDs to make assertions. This section proposes approaches for generating these IDs.

Agent identifiers The AGENT representation needs to contain a globally unique identifier for the thread of execution it represents. This is globally unique in the context of all provenance-aware threads of execution in the system.

For example, consider a distributed system running on two networked computers. Each computer runs an Operating System (OS), where the OS runs some of the distributed system's threads, and each thread is assigned a unique *OS thread ID*. The OS instance on each computer can safely duplicate the OS thread IDs of the other computer: the OS thread ID only needs to be unique within

each computer's OS, not the network. Therefore, to make these OS thread IDs globally unique for provenance collection, we would need to ensure each computer's thread IDs can be distinguished from one another; e.g. 'globally unique thread ID' = 'computer' + 'OS thread ID'.

Unique Agent ID composition

AgentID = 'Globally unique ID for a thread of execution'

Activity identifiers The same activity may be performed more than once, and each instance should have a globally unique identifier. As an example, each SOA interaction has an interaction key for this (see Section 2.5.1, "Identifying interactions").

A simple approach for composing a unique ID combines the agent ID with a simple counter, incremented each time the agent starts an activity. Each generated activity ID needs to be tracked, considering that nesting of activities may occur as sub-activities start. When a sub-activity is completed, the activity context needs to revert to the interrupted activity. A stack structure (Figure 4.2) could be used to track activities: the top of the stack always represents the current activity, and the stack layers track the nesting of activities.

Figure 4.2 shows one possible execution of an agent thread, where the agent is performing *activity A* which is interrupted by *activity B*. The agent's activities are tracked using a stack. A token is placed on the stack for *activity A* when it starts. When the agent starts *activity B*, a new token is placed on the stack: *activity A* has not yet finished, and its execution can be thought of as paused while the agent focuses on *activity B*. The agent completes *activity B* and reverts back to *activity A*: popping the top element from the stack reveals the token for *activity A*, which matches the agent's current activity.

Unique Activity ID composition

ActivityID = AgentID + 'Activity count'

Entity identifiers A system's runtime model can be structured and implemented in different ways. The different techniques and technologies for creating a runtime model can use different terminology to describe the parts of a runtime model. For this reason, the term *model element* is introduced to describe the unit of division of a runtime model that is tracked with provenance. For example, a Java

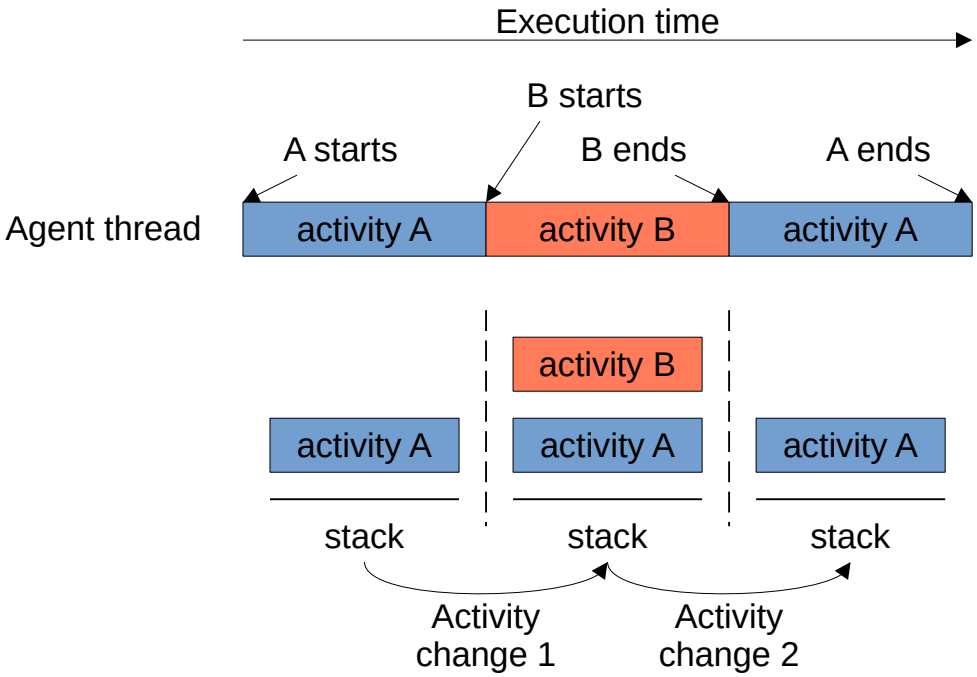

Figure 4.2: Tracking instances of Activities with a stack

programmer might create a runtime model using objects and object attributes, and each object's attributes would be the finest details of the model to be tracked; in this instance, object attributes would be referred to as the *model elements*.

There is also a need to clarify the term *object attribute*, as the terminology for describing an object's smaller units can differ: attributes, fields, or features. In EMF, a model object is considered to have features, with a distinction between attributes whose value belongs to a data type, or a reference to another object. However, in this dissertation, *object attribute* will be used in the broadest object-oriented programming sense: thus, an object attribute could contain either a value or a reference. For provenance collection, the value of a model element involved in an activity is recorded; the specifics of the implementation as a value or reference are not.

The runtime model representing the shared-knowledge in the system is something that more than one agent will refer to when recording provenance. The Entity IDs need to be unique for each model element and its different states during execution. However, more than one agent might interact with a model element in a given state: for example, two agents might read from the same version of the same model element. In this situation, creating or storing an entity ID based on information from the runtime model (which may implement its own IDs and versioning) makes more sense, instead of each agent counting entities or exchanging entity ID information.

Unique Entity ID composition

EntityID = ‘Model element ID’ + ‘Model element version’

Establishing unique IDs for agents/activities/entities is important for creating a provenance graph that does not have nonsensical loops. Suppose we had an activity A use an entity E on its first execution, and then modify it on its second execution. Figure 4.3 shows on the left an example of the nonsensical loops that can occur if the assertions do not include instance or version information: the original sequence of events has been lost. On the right, the same two assertions are made with versioning information for the entity (E.1, E.2), and instances for the activity (A.1, A.2). It is important that assertions differentiate new versions/instances, and also re-use the same versions/instances when re-use occurs.

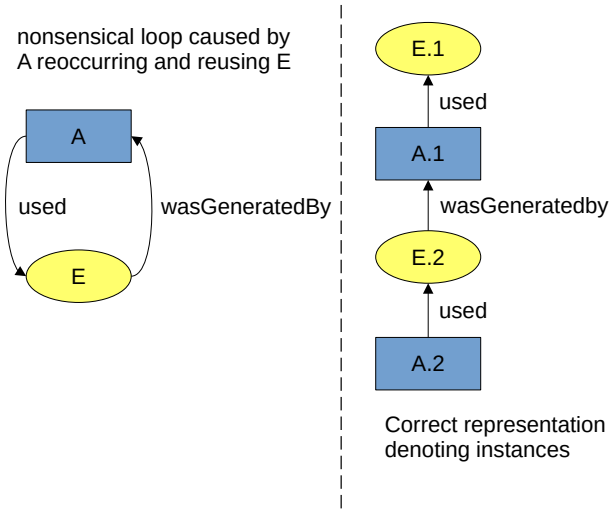

Figure 4.3: Instances/versioning of PROV nodes prevent nonsensical loops that can occur with non-unique IDs that do not separate instances/versions.

Representing provenance assertions with the model

The provenance store recording interface (Section 2.5.1, “Recording interface”) based on PrEP [48] provides further guidance for communicating and recording provenance as assertions. Assertions are usually only made once, and should be immutable once recorded. Each assertion should also be self-contained in terms of the information being recorded: this enables stateless processing of the assertion to a complete record of the event, without prior knowledge. However, there is scope for some flexibility in creating an assertion record: several incremental assertions could be

made, which add data to the record as it becomes known. If the incremental assertions repeated known information, an element of redundancy would be added to the recording of the assertions. The redundancy would reduce the errors caused by lost or out-of-sequence assertions, at the cost of increasing the bandwidth costs for recording. Given the model established so far for making provenance assertions, the following minimal provenance graph representations can be drawn for each of the four types of assertions proposed above.

Asserting an Agent starts/ends an Activity Figure 4.4 shows an Agent starting and ending an Activity. Only the ID and a marker for the state (started/ended) of the activity need to be present to record the event. The marker for the activity state could be based on a timestamp, using the asserting agent's clock. However, timestamps from multiple agents with different clocks can introduce a distributed clock problem: all the agent clocks would need to be synchronised, or aligned to a global clock to establish a timeline of activities based on the timestamps.

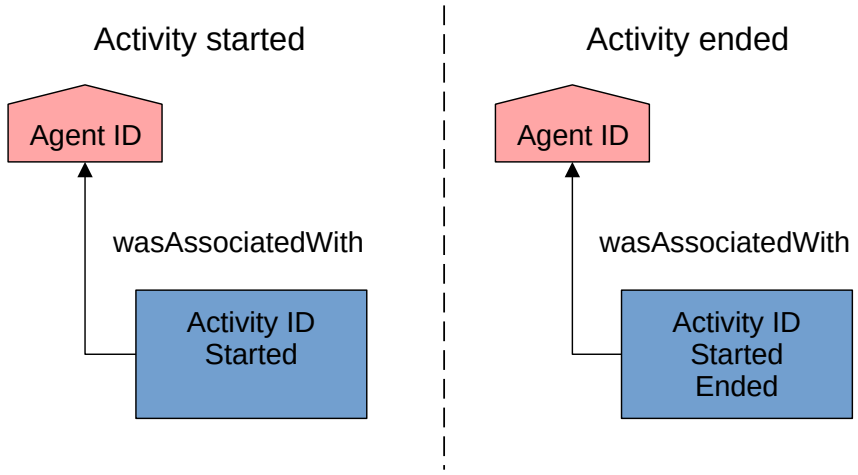

Figure 4.4: Provenance representations for starting and ending activities

Asserting an Agent performing an Activity reads/writes an Entity Figure 4.5 includes the same agent/activity graph description for the activity, with additional entities for the model interaction. Reading a model can be shown with a single entity and a *"used"* relationship: the value reflected must be the value the agent accessed. When an agent writes to the model, two entity representations are required. The first entity reports the model element and version that is about to be overwritten. A second entity indicates the new value and version for the model element that was written. A *"wasDerivedFrom"* relationship from the second entity connects to the first: this represents the version

change that occurred with the model element's state. The *"wasDerivedFrom"* relationship between these entities speeds up the retrieval of the provenance of a model element's versions or states: without this relationship, finding these versions would require costly traversals of the entities touched by the various activities.

Writing to the model is more complicated in a system with multiple agents that access a shared runtime model. There is the potential for model access conflicts to occur between agents if they try to access the same model element. The strategy for managing these model access conflicts is the system's responsibility. Provenance assertions for model accesses should be made based on the outcome of the strategy for handling model access conflicts: successful accesses are asserted, and failed accesses are not asserted.

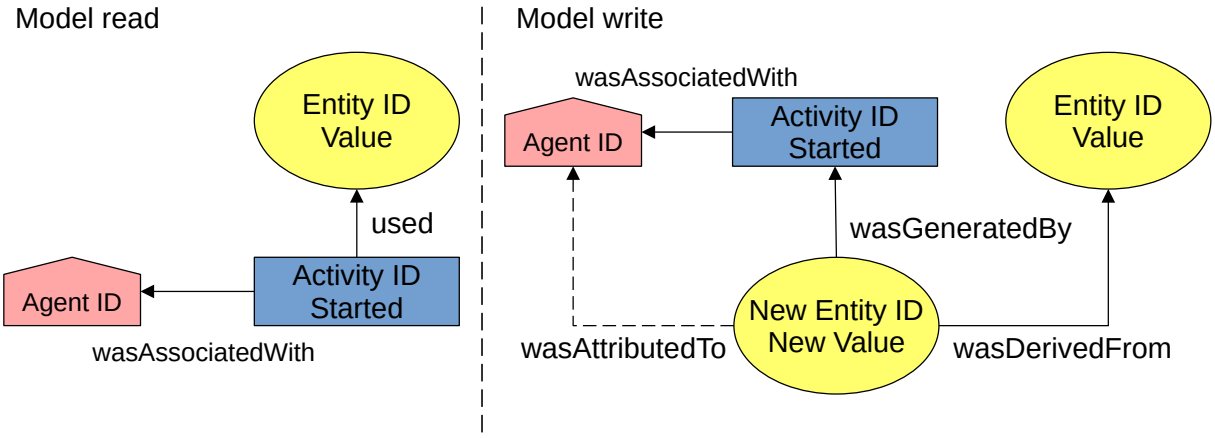

Figure 4.5: Model read/write PROV-DM descriptions

Assembling the provenance model of execution Provenance assertions of each agent could be handled as a message. Each message received could be processed to assemble a provenance graph that forms a flow diagram model of the execution of activities performed on the runtime model.

A simplified example of the provenance model that would be created is shown in Figure 4.6 (agent nodes removed for clarity). The provenance model can be assembled by matching the IDs on agent/activity/entity nodes in each provenance message. In some cases, the details of the node would be updated with new/additional information. The order of the messages does not affect the shape/content of the final graph. The flow of data can be seen in the layers of activities and entities.

SOA provenance templates use the *"wasDerivedFrom"* relationships to connect the output entities with some of the inputs, but this assumption does not hold for shared-knowledge systems. During the shared-knowledge system execution, an activity may read/write more than one model element.

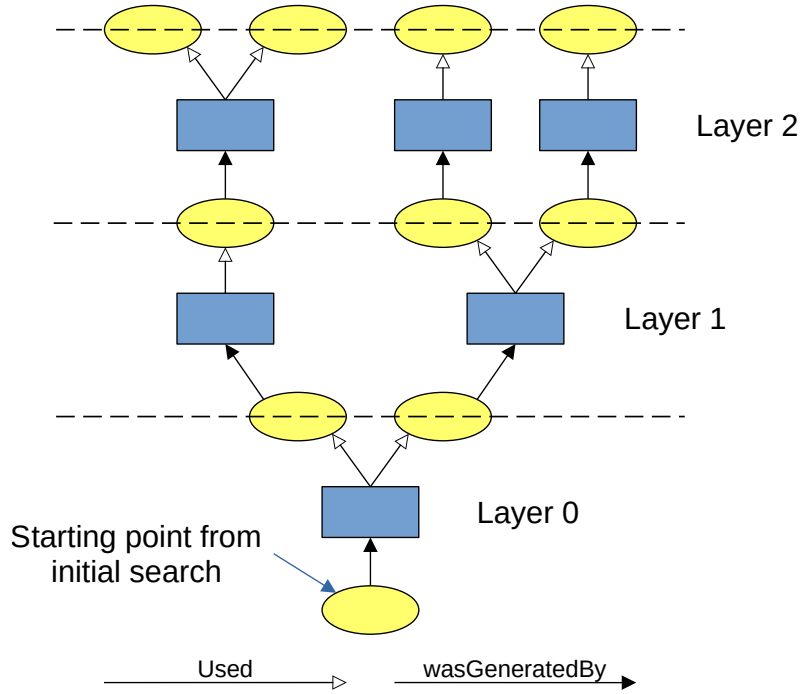

Figure 4.6: Layers of provenance formed by activities and entities (agent nodes removed for clarity)

However, the minimal start/end description of an activity lacks the context to indicate which inputs relate to which outputs, and detecting this relationship would potentially require significant use of static analysis, which would be outside the scope of this dissertation.

4.1.5 Demonstration

The design view of a hypothetical heating system's MAPE-K architecture is shown in Figure 4.7. The Monitor and Execute processes synchronise the runtime model with the physical heating system hardware (sensors and actuators). Analysis and Planning processes are intentionally simplistic. The Analysis process checks the difference between the desired temperature (dial position) and room temperature (heat sensor) that the monitor process recorded; based on this, the concept of 'Too Cold' (a Boolean value) is updated. Planning determines the state the heating should be in based on the state of 'Too Cold'; the Execute phase will synchronise the heating state with the boiler hardware.

The system runs through the MAPE loop and a change occurs with the thermostat setting for the desired room temperature (Figure 4.8). As the agent executes the MAPE phases (activities), a sequence of provenance messages is created and assembled to create a provenance model (Figure 4.9). This model of provenance identifies the agent (system), the 4 activities (MAPE phases) and some entities representing the runtime-model accesses (K model).

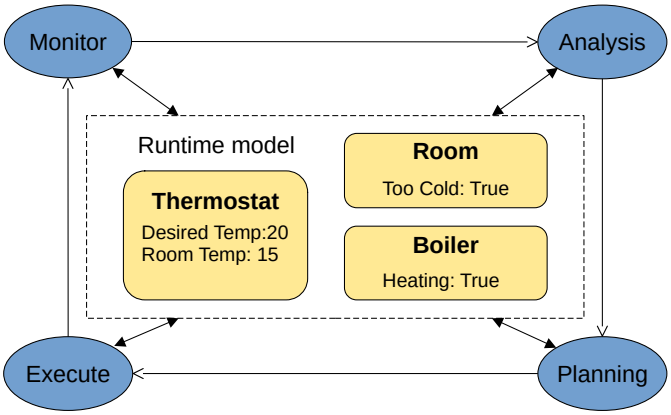

Figure 4.7: MAPE processes interacting with a runtime model of the hypothetical heating system.

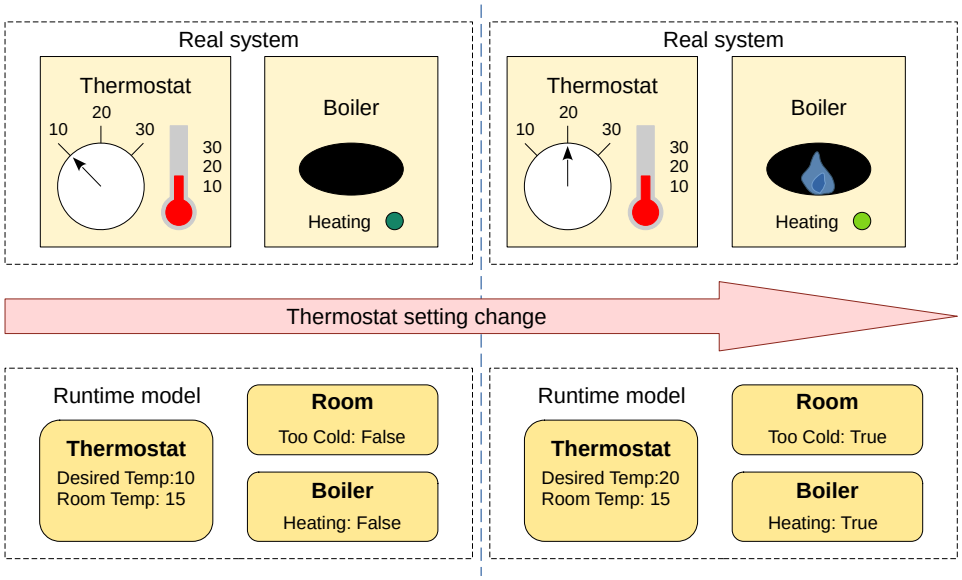

Figure 4.8: Hypothetical heating system with its runtime model representation: changes are synchronised between the real and model components

A provenance graph of a change in desired room temperature can answer some facets of provenance questions. While this graph represents only a single loop execution for simplicity, multiple executions of the loop would be captured. In a much larger graph, it would be possible to answer a facet of questions about data flow and past history. Some facets of provenance questions could be answered that start from MAPE concepts, such as “What knowledge did the monitor process update?”. Similarly, some facets of domain questions can be answered as they are present in the runtime model, such as “Was the room too cold?”.

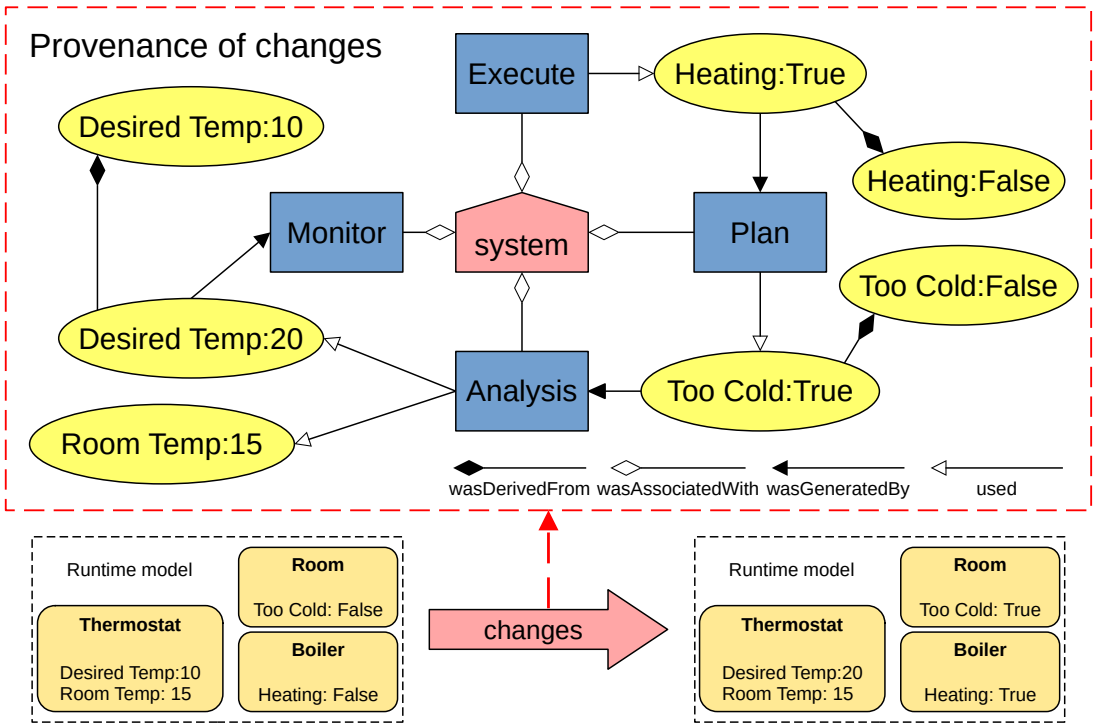

Figure 4.9: Provenance created by the MAPE loop responding to a change in the desired temperature

4.1.6 Evaluation

LRQ1

How can we model the provenance of processes interacting with a runtime model of shared-knowledge?

LRQ1 answer A provenance model for the execution of a shared-knowledge system based on a runtime-model architecture has been designed. The new approach to modelling provenance does not require the application of an SOA message/actor view of the system. This model has some similar properties to the SOA provenance in representing agent/activities/entities interacting, but these are in the context of runtime-model read/write operations. The provenance system architecture requirements for unique IDs to separate out the agent/activities/entities in assertions can be met.

LRQ2

How can the new provenance model capture domain-/system-specific facets of provenance, without a separate design view for provenance awareness?

LRQ2 answer As shown in the demonstration, provenance data can answer some facets of provenance questions: data flow, past history, and actors. These are some of the more general facets of provenance questions. However, there are some examples of provenance questions relating to facets for MAPE-K (system-specific facets) and for heating control (domain-specific facets). Some facets of provenance questions can be answered because they are represented in the system's runtime model or activity descriptions.

Processing the provenance messages from agents to a single provenance graph (in a provenance store) would result in a large graph. Within that provenance graph, there would be a single node representation for each agent and the unique activity instance/entity versions. The underlying technology used to store provenance may impose some limitations on the size of a provenance graph that can be created. Breaking up a provenance graph into smaller sections, and distributing the provenance over several graphs could be desirable. Distributing provenance over several provenance graphs might offer some interesting advantages for querying, as well as storage. However, the process of breaking up a large provenance graph could become computationally expensive, negating the possible advantages.

4.2 Conceptual design of a provenance store

4.2.1 Motivation

There is a need for a provenance store that can record the provenance messages at runtime in a structure that can support querying and management of data. The provenance messages from the agents to the store contain duplicate information, which motivated the idea of provenance being stored in a single large graph for storage space efficiency.

Having recorded the provenance of execution, there is a need to recall the state of the system's runtime model and use the provenance to explain the origins of its changes. However, we may not want an explanation for all the changes since the start of execution; we may only want to see recent changes or changes after some point. A similar view of the most recent provenance data having the highest value could be applied to data management, by deleting the oldest provenance data first.

4.2.2 Problem

Processing the provenance messages to a single provenance graph would reduce the storage space required to store the messages which contain duplicate information; a single graph node representation can be used to represent data present in several messages. However, the large graph could develop complex or long chains of dependency relationships between graph nodes. This large graph could become computationally expensive to process when making changes or performing queries. For example, recalling the state of the runtime model by replaying changes recorded in provenance since execution started, or retrieving a set of entities relating to some point after execution started.

Being able to establish the state of a runtime model helps to identify prior provenance information that could be deleted, based on a data management policy that deletes the oldest data first. However, if provenance data is deleted without considering later data that depends on it, it may not be possible to recall the state of the system's runtime model for some points in time after the deleted provenance record. This dependency will impact the possible provenance questions that can be answered.

4.2.3 Objective

Storing the information in the provenance messages requires a provenance store that balances two competing requirements. Efficient storage can be achieved using graph data structures that reuse unchanged nodes. However, these structures can impose additional processing costs when trying to establish the state of the runtime model at some point after the start of execution.

The ideal provenance store should try to achieve both fast recall of past runtime model states, and efficient use of storage. This store would enable the recall of the runtime model for some points during execution. These would be supported through small provenance graphs describing the detailed changes between these points. This approach would exchange some storage space efficiency for reduced processing times when recalling the state of the runtime model.

These objectives raise the following local research questions:

LRQ1 *How can provenance messages be stored efficiently in small provenance graphs that do not create long or complex dependency chains?*

LRQ2 *How does the provenance store structure support the recall of the state of the system's runtime model at some point after the start of execution? And how does it support storage space management (deleting) of the provenance data?*

The above questions are related to research question RQ1.1 from Section 3.3:

RQ1.1

How can provenance data be structured and stored in a way that enables storage costs to be managed by retention of time- or event-based periods?

4.2.4 Design and development

The PrIME approach to provenance awareness, based on an SOA view of a system, exposes data inside a system for provenance collection. Data can be exposed from messages between agents, or an agent can expose data from its own internal state. The data in an SOA may not be persisted outside of an interaction; i.e. messages are consumed and the data within the message is discarded. Therefore, the data required for provenance questions needs to be recorded in the provenance system for recall later. By contrast, in a shared-knowledge system, the data could be persisted in a runtime model and its state could be recorded using alternative methods to provenance awareness.

As mentioned in Section 2.1.3, model versioning is used to track changes to a model. Capturing the state of a model over time as a system executes is a problem that others have explored in the literature. A *filmstrip model* [55] is another approach for recording the changes to a model over time. A filmstrip model is a series of *snapshots* taken of a model's state, associated with some temporal information. The captured snapshots are interconnected using temporal relationships such as "precedes"/"succeeds".

A filmstrip enables the state of a model to be recalled at any point in time when a snapshot was taken. Replaying or recalling the state of the system after execution could enable an expert to reason about the system's execution. Clark et al. [23] created filmstrips of agent-based simulation models: these filmstrips were replayed after the simulation was executed to try and understand the emergent behaviours in the simulated system. In the instance of a simulation based on steps, each step can be captured in a snapshot: this enables any point in time of the simulation to be recalled.

Capturing the state of the system periodically or based on events can be used to replay or recall

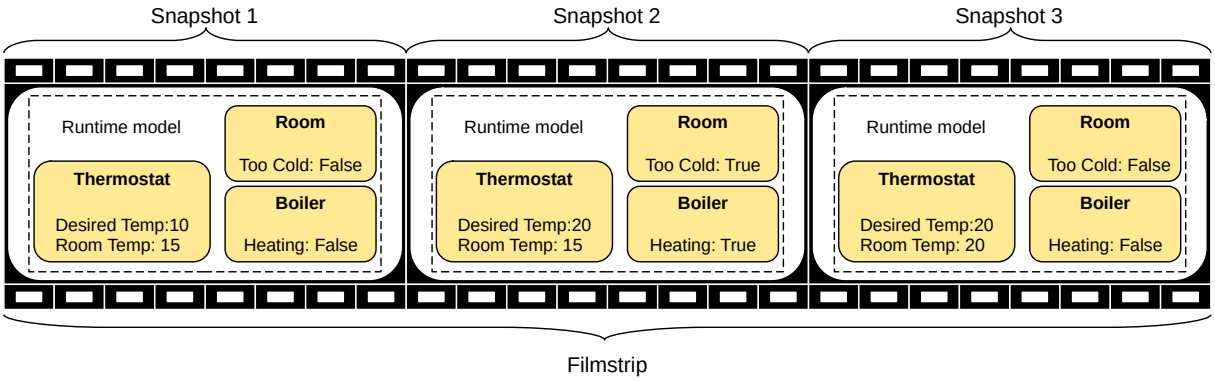

Figure 4.10: Conceptual model of a filmstrip of snapshots taken of the hypothetical heating system's runtime model.

the execution of the system. However, filmstrips have some limitations: the most obvious is that they only represent the state of the system, which lacks the context of processes/activities. If snapshots are not triggered by events or state changes, then depending on how often they are taken, there is a risk that intermediate states between snapshots could be missed.

Using the hypothetical heating system as an example, Figure 4.10 shows a set of snapshots of the system's runtime model. The snapshots are arranged in chronological order to make a filmstrip. If these snapshots were taken whenever a change occurred (one per change), we could replay the filmstrip and see each change in sequence. However, this process could quickly consume storage space if many changes occurred and created snapshots. There are ways to reduce the space required to store snapshots, like storing only the deltas that change between snapshots, but these fine-grained snapshots would lack the reasons for the changes.

History model proposal

We propose an alternative solution, which takes snapshots less frequently, and instead records the changes between snapshots using provenance. Used in this way, provenance captures both the state changes and the causal information. Additional snapshots of the full runtime model can be stored with a provenance graph, describing its state after each set of fine-grained changes.

Time windows Combining a provenance graph description for small runtime model changes with process information, and a snapshot of the initial state of the runtime model, creates a *time window*. A snapshot could be loaded into memory from a time window, to provide a *base version* of the runtime model. Using the provenance graph, the small changes can be applied to reconstruct

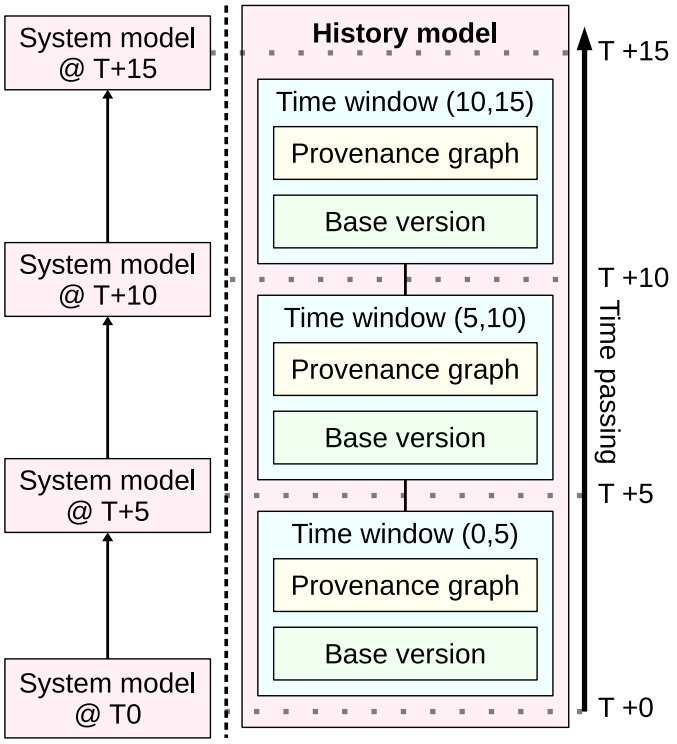

Figure 4.11: Proposed history model

all the intermediate states of the runtime model between snapshots. This effectively replaces the need to take a snapshot for each model change, and adds change process documentation to the filmstrip. A chronologically-ordered sequence of time windows, each containing a snapshot and provenance graph, would form a *history model* (Figure 4.11: a way to replay runtime model changes with documentation of the origin of each change.

The history model divides the system execution into time windows, and avoids creating a single large provenance graph of a system's execution. A time window starts when a snapshot is taken, and ends when the following snapshot is taken (starting a new time window). The time interval between snapshots can be variable or constant; alternatively, a snapshot could be triggered by the accumulation of a certain number of changes. However, the sequencing of the snapshots and the provenance messages used to build the provenance graph of small changes must be carefully aligned. This requires identifying which snapshots and provenance messages relate to a given time window. If done correctly, a time window becomes a self-contained representation of the system execution for a period, and is consistent with the data in neighbouring time windows.

The data across two adjacent time windows is consistent if the snapshot of the first time window can be matched with the snapshot from the following time window, after applying the changes

recorded in the provenance graph of the first time window. When time windows are consistent in this way, it would be possible for the provenance graphs of neighbouring time windows to be merged (and the intermediate snapshot between graphs removed). This may not be desirable for storage purposes, but a merged provenance graph could simplify some provenance queries. For example, a provenance question might ask about a long-running process which spans multiple time windows. A complicated query would be needed to trace provenance across time windows and query multiple provenance graphs. Alternatively, the provenance graphs covering the execution of the long process could be merged, and a single query executed. In effect, time windows avoid scaling issues with a large provenance graph, but their use does not prevent a large provenance graph from being generated if required.

How do we handle provenance for a system process that spans multiple time windows?

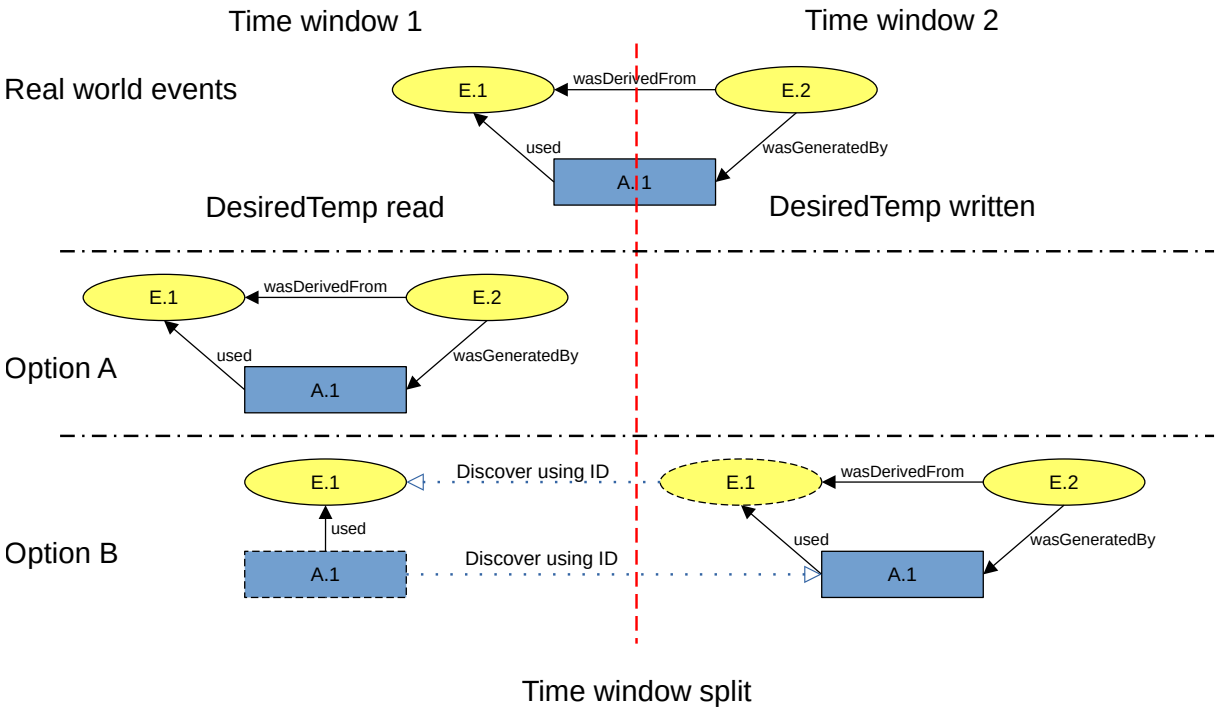

Figure 4.12: Options to manage activities spanning time windows

A complication with the provenance collection and recording occurs when a system's process (Figure 4.12, real-world events ACTIVITY A.1) runs over multiple time windows. The sequence of provenance messages for the process is interrupted with a new snapshot opening a new time window. The provenance collected for the activity could be stored using one of two options:

Option A: *time window 1* would remain open until the activity *A.1* is completed, with all provenance

being stored in it. However, this approach has some flaws: for example, if the activity runs until the system completes its execution, the time window would never be closed, continuing to take up an increasing amount of system memory.

Option B: distribute the provenance across multiple time windows as events occur. In this case, each window represents the data available at that point in time. In the example, the activity *A.1* in *time window 1* would contain only partial information for the activity (only the start flag is set). In *time window 2*, a complete representation for the activity *A.1* can be stored, containing start and end flags. In addition, the instance ID for the activity *A.1* should be the same in both windows, enabling the provenance in both time windows to be merged into a single provenance graph. Therefore, it would be possible to search forward from *time window 1* to discover later provenance information relating to the same activity.

A similar problem occurs with the entity representations over multiple time windows. In Figure 4.12, the ENTITY *E.1* requires a representation in both time windows. The *time window 1* ENTITY *E.1* would lack the *"wasDerivedFrom"* relationship from *E.2* (indicating the change of version/state from *E.1* to *E.2*). There is a need to traverse nearby time windows to find other occurrences of *E.1*, to discover prior and later accesses. In *time window 2* there is a *source* ENTITY *E.2*: it has a *"wasDerivedFrom"* relationship (indicating the prior version/state), and a *"wasGeneratedBy"* relationship with an ACTIVITY (indicating the activity that created the state). The ENTITY *E.1* is not a *source* ENTITY: it is missing relationships that establish its origins (*"wasDerivedFrom"* and *"wasGeneratedBy"*). Upon detecting the missing relationships, it would be possible to search backwards through time windows to find other occurrences of *E.1*, until a *source* ENTITY is found. Figure 4.12 does not contain a *source* ENTITY for *E.1*: its origins are unknown, as the oldest provenance record only accounts for its use by *A.1* in *time window 1*.

How can the storage space consumed by a history model be managed?

As mentioned above, time windows are self-contained representations of the system's execution for a period. Time windows allow for controlling how much storage space is used, as they and their constituent parts may be selectively discarded. Figure 4.13 show some of the possible discard strategies.

Discard strategy 1 is the simplest, whereby the oldest time windows are completely discarded. Depending on the available resources, the retention policy may keep n *time windows* of a given

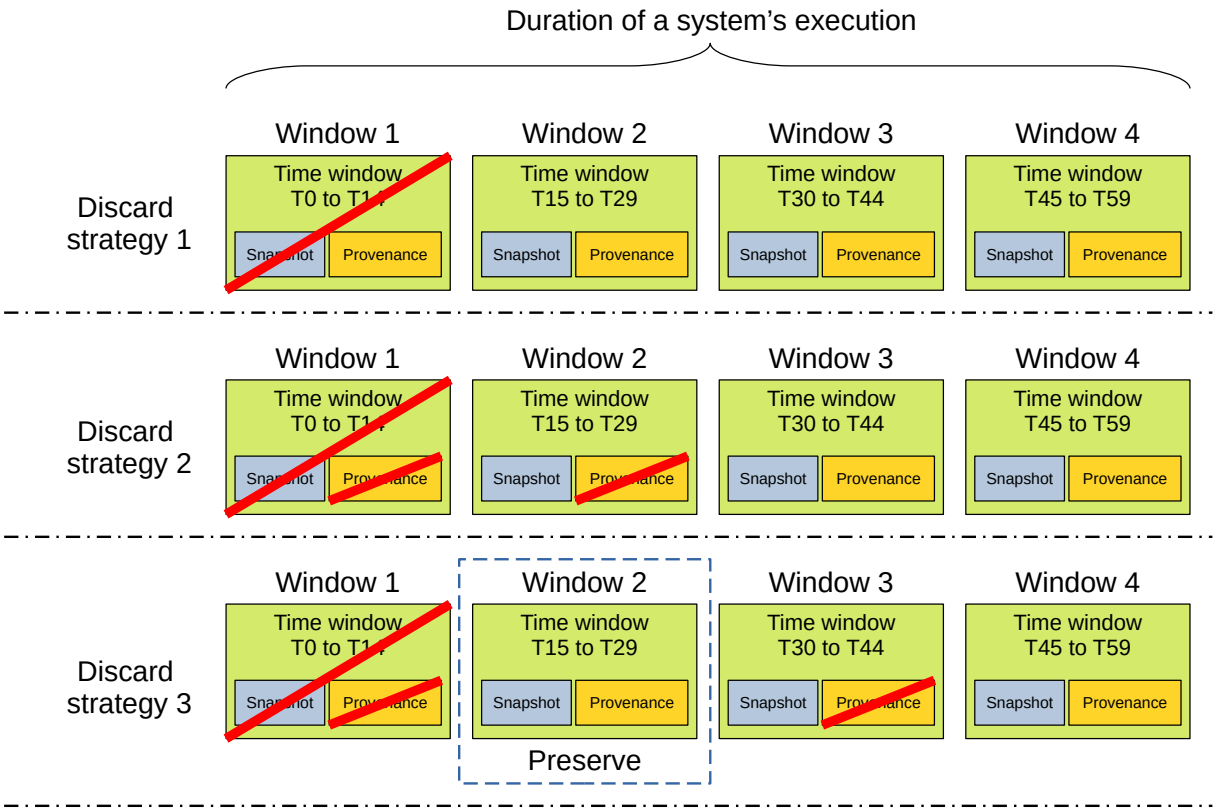

Figure 4.13: Strategies for discarding time window data

duration or may apply a time-based policy that retains d days of data. Thus, *discard strategy 1* is a time window-level decision-making process to retain/discard data.

Discard strategy 2 introduces an intermediate level of data between complete and discarded time windows. The partial time window (*Discard strategy 2*, Window 2) contains only a snapshot: the provenance graph (detail) is discarded. Thus, the approach would enable recovery of the system state at a point in time when the snapshot was taken, but would lack detailed causal information. Therefore, a loss of information has occurred, but not a total loss. It may be possible to answer some questions using only the snapshot. The independence of the time windows allows the windows to contain different levels of information.

Discard strategy 3 extends *discard strategy 2*, introducing the concept of a *preserve* flag. For example, a single time window may capture data for an event of interest previously defined by a policy or filter. Therefore, a preserve flag is set to keep the time window in the store, which excludes it from any automated discard strategy. Again, the flexibility to keep/discard part/all of a time window does not require that prior or sequential time windows be kept.

4.2.5 Demonstration

To demonstrate the concept of a history model, the heating system from Section 4.1.5 can be used. The change from the previous example (shown in Figure 4.8 on page 104) is shown as a time window in Figure 4.14. Four time windows capture the execution of the system from T0 to T59: each time window contains a snapshot and a provenance graph. In this example, the policy for taking snapshots is to take one at a fixed time interval, with each time window recording 15 T units of system execution in a provenance graph.

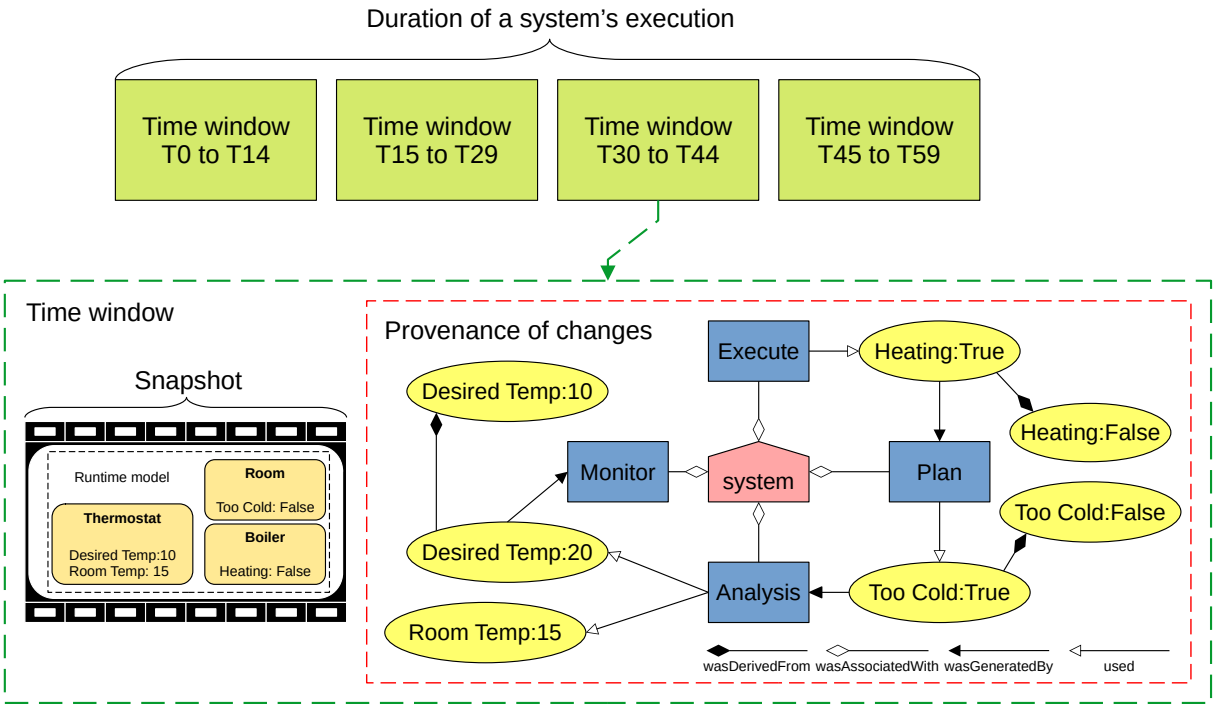

Figure 4.14: Time windows demonstrated with the hypothetical heating system

The system's execution from T0 to T59 could be recorded in a single large provenance graph. Assuming the full state of the system's runtime model is known at T0, the changes recorded in provenance can be applied to a copy of the system's T0 runtime model. The copy of the runtime model would represent the state of the system's runtime model at any point between T0 and T59. For example, recovering the runtime model state at T17 requires replaying the changes recorded using provenance from T0 to T17.

The full state of the heating system's runtime model can be recalled using the snapshot in a time window. Depending on the snapshot policy, the system's runtime model can be visited at some points without needing to access the provenance data. In the example with the heating system, the full state

of the system can be explored at an interval of 15 T units without using the provenance data.

Recalling a point between snapshots is possible by applying the provenance of changes to the snapshot within the same time windows. For example, to recover the runtime model at T17 only the time window from T15 to T29 is needed. The runtime model snapshot would be loaded, and the provenance graph of changes that occurred for 2 T units would be applied to the runtime model state up to T17. This approach omits the processing of the provenance graph covering T0 to T14 which would be required if a single large provenance graph had been created.

Time windows can reduce the amount of provenance that needs to be processed for some provenance questions. Using the snapshot policy logic it is sometimes possible to filter out provenance from some time windows. For example, a provenance question might ask about the state of the runtime model which is expected to be found between two snapshots. All time windows outside these points can be excluded from a provenance query process. In the case of Figure 4.14, if asked "Was the boiler on after T15 and before T29?"; only 15 T units of provenance need to be loaded and processed. Performing the same query on a single large provenance graph (for T0 to T59) requires all 60 T units of provenance to be loaded and processed.

There are scenarios where there may be no entities for a model element in a provenance graph. If the same question was asked ("Was the boiler on after T15 and before T29?") while the boiler state had never been accessed in that time period, the provenance query might return no entities for it. In this instance, the snapshot would be used to determine the boiler state for the duration of the time windows. If we were using a single large provenance graph to achieve the same, we would need to process the provenance for T0 to T14, to establish the state of the boiler at T15.

Using the snapshot to establish the full state of a system's runtime model at points in time can reduce the amount of provenance processing to answer some questions. There is no need to process all provenance from the earliest records of execution up to some point after to establish the state of the runtime model. This also helps reduce the processing when there is a need to 'forget' or delete the oldest provenance data for data management. As discussed in the above discard strategies, an entire time window can be safely deleted. To remove the same provenance from a single large graph would require some processing. The provenance graph would need to retain at least one entity for each model element: if all entities were deleted and no further access occurred, the state of the model element would become unknown.

4.2.6 Evaluation

This iteration has produced the concept of a history model, which combines filmstrips and provenance awareness. A history model contains a sequence of time windows, with filmstrip-like snapshots of the runtime model that provide a base version of the model state, which changes during the time window. Changes that occur within a time window are recorded using provenance awareness, creating a graph that can answer provenance questions. Recalling the state of the runtime model can be achieved using a snapshot and then applying the changes recorded in the provenance graph, up to the desired point in time.

The iteration has provided the following answers to its research questions:

LRQ1

How can provenance messages be stored efficiently in small provenance graphs that do not create long or complex dependency chains?

LRQ1 answer The system execution can be divided into time windows: the snapshot in the time window acts as the starting point of the execution captured in its provenance graph. Thus, the provenance of a long execution can be represented as several smaller isolated provenance graphs.

LRQ2

How does the provenance store structure support the recall of the state of the system's runtime model at some point after the start of execution? And how does it support storage space management (deleting) of the provenance data?

LRQ2 answer Being able to recall the full state of the runtime model at points in time can save significant amounts of provenance graph processing. This can be seen with the discarding strategies and with answering some provenance questions. Depending on the snapshot policy, some provenance questions can be answered by processing a filtered set of time windows. For example, a provenance question might seek an answer that can be found between two time window snapshots: only the time window provenance graphs between these would need to be processed.

Chapter 5

Design and development: MDE shared-knowledge bases

In this chapter, the design research process for Cronista is presented. Cronista was developed incrementally from the conceptual design for collecting provenance during a system's execution. A series of exploratory experiments were carried out to evaluate how the provenance data collected by Cronista could be used to enable provenance-based explanations for a system's execution.

5.1 Initial implementation of Cronista

5.1.1 Motivation

The design and development process described in Chapter 4 has produced some conceptual designs for provenance collection from a shared-knowledge system (Section 4.1), and designed a history model structure for a provenance store (Section 4.2). To evaluate these conceptual designs further, an implementation is required for evaluation through experimentation.

Experimentation may reveal details that had not been considered. These details may be problems with the design, or limitations that require the design to be extended. Similarly, some assumptions about the re-usability of system components and the scope of generalisation across domains or system types could be incorrect.

5.1.2 Problem

There is a collection of different problems surrounding the motivation of creating an implementation for evaluation. The design and development of an artefact for evaluating the concept could provide more knowledge than a validation of the design concept. There are problems to be addressed with creating an implementation that presents some 'value' for its use, or more broadly, with applying provenance awareness to a system. The resulting implementation may not be valuable in terms of practical use in a commercial or production environment, but the knowledge of its design and development could be valuable.

Problems with provenance awareness increasing system costs

Applying provenance awareness to a system requires adding some components, like a provenance store. Instrumentation throughout the system must be added to collect and record provenance data. Deploying instrumentation to a system could require a developer to make changes all over the codebase of a system. These repetitive manual changes to a system raise the risk of introducing software defects, which can further increase system development costs.

A system's runtime resource requirements are increased when provenance-awareness is applied. Therefore, an approach that can tightly integrate with a system and reduce these overheads is desirable. The inverse problem is that coupling the integration too tightly can negatively affect the reusability of the components. Reuse of component implementations is one way to increase the return on the investment of the time and resources needed to create them.

Problems with developer buy-in/learning a new tool or technique

Getting developers to use new tools and techniques is generally a challenge. If a developer has to overcome a steep learning curve without clear rewards for their efforts, they are less likely to invest time and effort in learning something new. Furthermore, if a tool or skill is not widely reusable, the return on learning efforts is reduced compared to those generalising to different tasks. Ideally, tools and skills that can make a developer more productive over a wide array of software projects are more appealing.

Systems with a shared-knowledge base may not share a common approach to implementing their runtime model. Each system may use a different technology or data structure for its runtime model,

and could require a different approach or tool applying the instrumentation. As such, an approach to instrumenting these models needs to be adaptable to each implementation; the resulting instrumentation may then be reusable across systems that share a common runtime model implementation. Developers can then be presented with a reusable and unified approach to instrumenting runtime models; as opposed to a fragmented collection of different tools for each type of runtime model.

An approach to provenance awareness that targets widely used frameworks could reduce the issue with developer adoption, as well as the costs of creating provenance-aware systems: if it works for one system using a certain framework, it may also work with other systems built with the same framework. This would require a provenance awareness approach to be implemented as an extension of that framework.

Problems with managing the research project costs (time)

There is a need to find a framework that can support the creation of both a shared-knowledge base and a provenance model. Using a common framework should enable a close integration of the two systems, which could reduce the costs for both the research project and when used for applying provenance awareness. The selected framework would need to supply structures which could be extended with provenance collection instrumentation; allowing a system to be built (implicitly) with an awareness of the interactions between agents, activities, and entities during execution.

Provenance-related problems

The rest of this section will discuss the various provenance-related problems that need to be addressed during the design and development of the implementation as a framework.

Agent A computer system can execute many threads at runtime, and some of these threads may not be part of the system being made provenance-aware. A system can dynamically create and destroy threads of execution as it runs; a system's execution could start and end on different threads. A developer needs a way to identify some threads that perform the execution of activities that are subject to provenance awareness. Provenance recording should only apply to the threads that a developer denotes as AGENTS.

Activity The system's codebase defines the processes or *high-level activities* the system will perform at runtime. A developer needs to define the scope (start and end) of activities: this *activity scope* would need to be detectable by the provenance collection system at runtime. The activity scope would create an ACTIVITY node that would refer to a section of the system's code being executed by an AGENT.

Activities could be mapped to the execution of specific language constructs, e.g. the execution of methods or functions. Thus, a system's processes relating to a high-level activity (activity scope) could be aligned with a language construct. However, there may be cases when the high-level activities and language constructs do not align: for example, retrofitting provenance awareness to an existing system code-base where language constructs have been mapped to other concerns. The options are 1) accept the misalignment of the activities that may hinder understanding of the provenance, or 2) restructure the system code to match high-level activities, which could be a costly exercise, and in the worst case may result in a less maintainable code base.

An alternative approach could require programmers to add some annotations or lines of code to the system's code base, reducing the need to restructure code. A developer would place start and end markers for each activity scope in the system code. At runtime, the system's provenance awareness would detect the agent's execution passing these markers, and record that an agent started/ended an activity accordingly. A challenge with such an approach is ensuring that agent execution cannot enter or exit an activity scope unexpectedly and skip the start or end markers.

Entity Entities will describe the model elements that the system agents interact with during activities. A system's runtime model implementation will have data structures that form its collection of model elements. The instrumentation needs to wrap or integrate with the runtime model implementation to enable an awareness of the model element accesses. The provenance description (ENTITY) for each access must include information for correlating and versioning the model element accessed by an agent performing an activity.

The code for a system interacting with its runtime model has two aspects that could be instrumented to report model interactions: the code that uses the model, and the code that implements the model. A system's code should ideally have consistent implementations for the model and for the code using the model, assuming good engineering practices are followed. However, there are no guarantees that i) the system code follows consistent patterns that help identify model elements

or model interactions, or that ii) the system's runtime model implementation provides information to assist with identification and versioning of model elements. The approach to instrumenting a runtime model must address several common requirements:

Model element access A system may contain one or more runtime models. The developer needs to identify which runtime models need to be provenance-aware. All interactions with a provenance-aware runtime model (i.e. accesses to its elements) must be checked to see if provenance should be recorded. If an agent's activity is the source of a model interaction, then provenance must be recorded. However, some detected model interactions do not require recording provenance, as they do not relate to an agent activity and would only create *access noise*. For example, a runtime model may be loaded or unloaded from memory as part of an external background process outside the system's internal processes. Thus instrumentation of the runtime model needs to distinguish interactions requiring provenance recording from those that do not.

Model element correlation Having detected access to a model element, information to enable the ENTITY to be correlated with the accessed model element is needed. The runtime model implementation needs to contain some identification system that can provide unique IDs for the model elements. For example, a system's runtime model could include an internal ID system, which enables a specific model element to be distinguished from all others through the allocation of a unique ID which remains consistent for the lifetime of that model element. Instrumentation could hook into existing model element IDs, or could provide its own mechanisms to correlate the model element and entity representations.

Model element versioning The identity of a model element remains the same during its lifetime. However, the state or value of a model element will change over its lifetime. There is a need to track these changes, and distinguish between different values or states of the same model element.

Versioning of the model elements is one way this can be achieved, e.g. by adding a counter to each model element that increments with each change. Each model access can include this version counter to indicate which state (version number) was accessed. Instrumentation could extract versioning information from the model element implementations if it exists, or provide its own version tracking.

Model snapshots The runtime model of the system will need to be implemented in a way that can be captured as a snapshot. This could be achieved by copying the runtime model data to the history model. The process to produce a snapshot must result in a consistent copy of the runtime model, i.e. the snapshot should accurately reflect the runtime model's state at this point.

A snapshot of the runtime model must include ID and version information that enables the model elements represented in the snapshot to be correlated with the provenance graph entity nodes. For example, some model copy operations might create new instances of model elements with new IDs: the elements of this new model copy would then have IDs that do not correlate with the original provenance graph.

5.1.3 Objective

Enabling a system with provenance awareness requires instrumenting some system components for provenance collection. The approach for instrumentation will depend partly on how the component has been implemented and what features it has that may be useful for provenance awareness. For example, instrumenting a system model that includes an ID system requires the instrumentation to expose the IDs; if there is no ID system present, then the instrumentation would need to provide one. Providing developers with reusable instrumentation requires that the systems being made provenance-aware share some common implementation features, to which the instrumentation can be applied.

Frameworks are one way for a developer to reduce the cost of implementing a system, and can provide consistent code patterns for common features or functions, e.g. runtime model implementation. An approach to provenance collection for systems created using a framework should be simpler than one aiming to support any system built using a general-purpose language; uncertainty with the system's implementation is reduced if the framework's structures are targeted for provenance instrumentation.

The implementation in this development cycle is a proof-of-concept focusing on capturing the provenance of a runtime model. As such, the aim is to track the provenance of an in-memory runtime model as a single-threaded system interacts with it. This is sufficient for evaluating the system instrumentation approach to record execution provenance using agent, activity, and entity provenance information collected at runtime. To collect this provenance information, the following local research

questions need to be answered:

LRQ1 *How can a system be instrumented by extending the framework used for its shared-knowledge base, to enable a reusable approach to provenance awareness?*

- **LRQ1.1** *How can a developer instrument system execution threads (Agents), for provenance awareness?*
- **LRQ1.2** *How can a developer instrument a system to define where Activities start and end, for provenance awareness?*
- **LRQ1.3** *How can a developer instrument a system's runtime model to record accesses to model elements (Entities), for provenance awareness?*
- **LRQ1.4** *How can snapshots be taken that preserve the relationship between runtime model elements and entities?*

LRQ2 *How can a provenance store for a history model be implemented?*

- **LRQ2.1** *How can the concerns between the Curator and storage be separated, enabling the use of different technologies for history model storage without requiring changes to the Curator?*
- **LRQ2.2** *Could an existing model-driven engineering (MDE) technology be used to implement the storage for a history model to reduce development costs?*
- **LRQ2.3** *How can the history model inside the storage component be accessed using existing technologies?*

The local research questions should contribute to RQ1.2:

RQ1.2

To what extent can the reuse of the existing infrastructure for the shared-knowledge system also reduce the costs of providing provenance awareness?

5.1.4 Design and Development

The models for both the shared-knowledge base and the provenance graph could be implemented using Model-Driven Engineering (MDE) tools (Section 2.1.3). The Eclipse Modelling Framework (EMF) is an open-source MDE toolset for developing modelling languages and models [32, 104]. EMF has a code generator that can create code implementations of model components with consistent implementation patterns, for use in systems. The model code includes convenient features from the EMF libraries such as reflection, object IDs, and persistence methods.

How a system's runtime model is implemented can significantly impact the practical approach to creating instrumentation for provenance awareness. It could make sense to reduce the complexity of the initial runtime model instrumentation by requiring the system's runtime model to be built using EMF.

However, native EMF models do not have some of the features required to reliably track model elements with provenance awareness as ENTITIES, e.g. versioning and thread-safe model access. Existing tools in the EMF ecosystem, like the Eclipse CDO [28] model repository, potentially provide solutions to some of these problems. For simplicity of the initial implementation, the model element versioning system can be included with the model instrumentation. A versioning system for the native EMF model could be triggered by the instrumentation applied to a model to track provenance. These components/extensions can be packaged as a library, for adding to other EMF modelling projects that extend the native EMF model with versioning. Regarding thread safety, the initial implementation will temporarily remove this requirement by assuming the system is single-threaded.

EMF can be used to implement PROV-DM so developers can avoid learning a provenance-specific technology like ProvToolbox[80]. Whereas the knowledge of a provenance-specific technology is reusable for a scope of projects requiring provenance, modelling with EMF may apply to a wider scope of projects (both those requiring provenance, and those that only require modelling system concerns). For a developer who already knows EMF, they may only need to learn some additional provenance concepts.

An EMF PROV-DM implementation can be used to create EMF models of provenance, for use with other EMF-compatible modelling tools. This could ease the integration of provenance-awareness with model-based systems built using EMF, as both systems would share a common underlying modelling technology. A developer might then be able to use a common set of MDE tools for working

with a system's different models: design models, runtime models, and provenance models.

Software Architecture Overview

For the reasons explained above, EMF and CDO were selected as the base technologies for the initial implementation. The overall architecture of Cronista was defined as shown in Figure 5.1 (page 126). There are some similarities to the provenance system architecture discussed in the background (Section 2.5.1). Cronista provides some actor-side libraries, known as the *Observer* components. These include Java and EMF components for instrumenting a system with provenance awareness.

In the provenance system architecture [52], the logical view of a provenance system (Figure 2.8 on page 49) shows the various supporting components that need to be considered. These include several interfaces around a provenance store. Our approach to managing provenance data (PROV-DM) as an MDE model enables us to use existing MDE tools (Section 2.1.3) for some of the supporting components the provenance system architecture describes. For example, Cronista's provenance store uses an EMF model repository (CDO), which provides model persistence and APIs for recording, managing, and querying the model data it holds. Plugins for the Eclipse IDE, like CDO Explorer [28], can be used to view and interact with the history model in a CDO model repository. Similarly, the same history model in CDO can be queried using the Epsilon toolset, which provides the Epsilon Object Language (EOL) for model querying.

Cronista's *Curator* and *History Model Store* (HMS) components (Figure 5.1) are similar to a provenance system consisting of a provenance store with several API interfaces (Figure 2.8 on page 49). The provenance system components store the collected provenance and provide access to the stored data. Cronista's Curator and HMS provide the same functionality, and they run in separate threads of execution from the system's regular processes. This allows concurrent execution of the system, and Cronista threads, reducing the impact of provenance storage on the system. To enable this concurrent execution, provenance messages must pass from the system thread containing the Observer(s) to a Curator thread without blocking execution on either thread: to do this, a message queue is used to decouple the threads.

This section will now discuss the design of the components implemented for Cronista. The discussion will start with the Observer components used to instrument a system for provenance awareness, followed by the Curator components that are external to a system. These Curator components manage and store the provenance data from the Observer components.

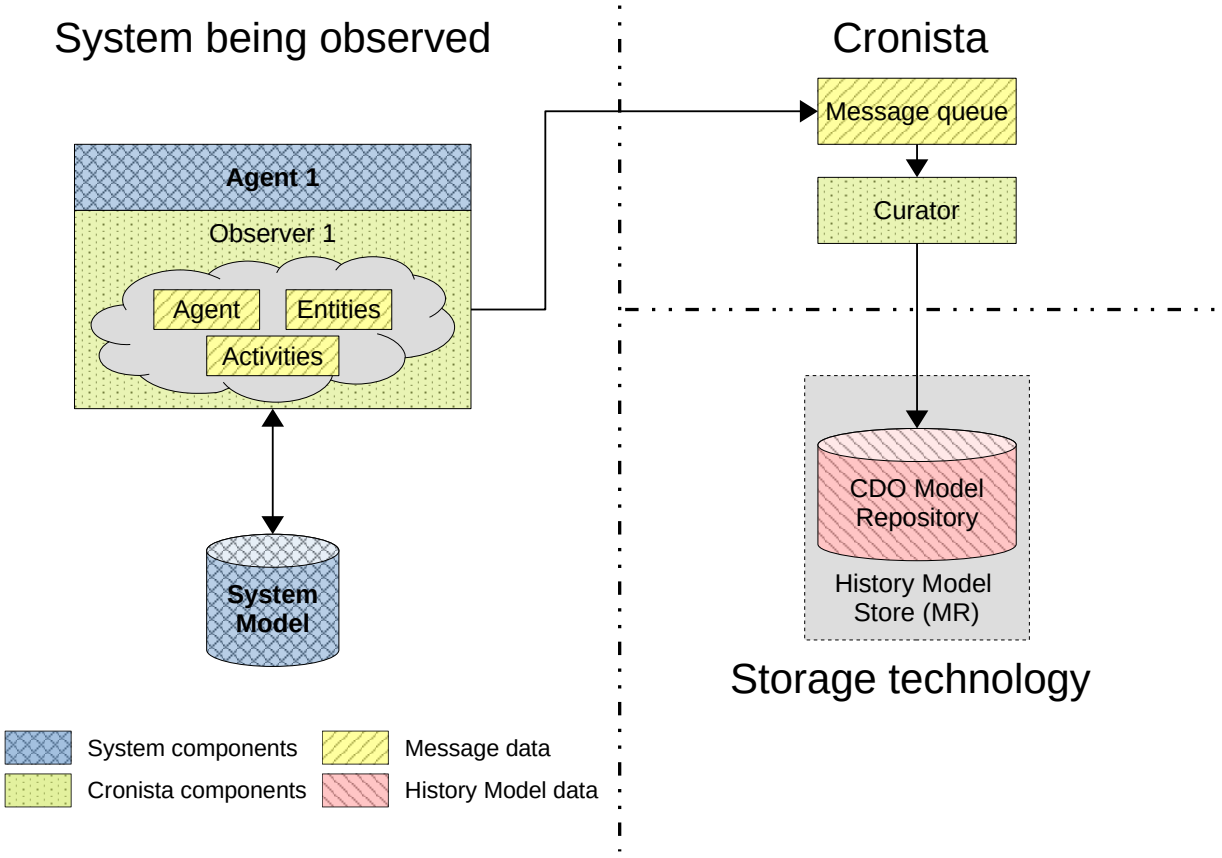

Figure 5.1: Initial architecture of Cronista: top-level components and interconnections.

Implementation of Observer components

The Observer components in Cronista are packaged as a Java library, for use with different target systems that use EMF models. Each thread in a target system is considered an agent, and requires an Observer object instance to enable provenance awareness. This Observer will queue provenance messages for a Curator. These provenance messages describe the start and end of agent activities, and successful model accesses that occur during an agent's activities.

In addition to the Observer components that attach to execution threads, there are some EMF model extensions. The EMF code generator is configured to produce code that uses these extensions for the system's runtime models. The extensions introduce the provenance instrumentation needed by Cronista, and provide default implementations of some required features for a system's EMF model, e.g. a simple model versioning system.

Observing agent activities

The components required to instrument a system with provenance awareness of agents and activities are not overly complex. These are implemented using standard Java features. This section describes the design for these components.

Agent (LRQ 1.1) All system threads that need to be made provenance-aware require an Observer that will collect and record provenance. A developer can leave some threads of execution outside of the provenance awareness by not creating observers for them. The Observer identifies the thread it is in as the AGENT responsible for the provenance it reports. All provenance collected and recorded by that Observer is in the context of that thread's actions, and must therefore be associated with an Agent ID.

The Java thread name can be used as the Agent ID, assuming a developer does not duplicate thread names within the same execution of a system. A fixed Agent ID can be used for repeated executions of the system, if the same AGENT should be identifiable across multiple executions. However, using a unique system-assigned thread ID as an agent ID (or part of the agent ID) provides a slightly more robust approach to creating unique AGENTS representations in the same execution of the system when required.

Activities (LRQ 1.2) To detect when activities start and end, Cronista observers require a developer to wrap the code for a given activity with an *activity scope*. Activity scopes are implemented with Java *try-with-resource* blocks (Listing 5.1), which automatically acquire a resource when reached, and release the resource when execution leaves the block (whether normally or through an exception). Thus, resource acquisition for an activity scope signals the start of the activity, and the release of the same resource signals the end of the same activity.

An `ActivityDescription` object resource is acquired and released by the activity scope. These are tracked by the Observer using a stack. The currently running activity is always on top of the stack, and the objects below mirror the nesting of activities (started but not yet finished). The `ActivityDescription` class contains the information needed to populate the ACTIVITY node: Activity ID and start/end times. An additional "name" attribute was added to provide a meaningful name for the activity, which would be visible to a user reading the provenance graph.

Enabling provenance awareness of the system's runtime model involves more than simply

```
1 try (var aScope = new ActivityScope("ActivityName")) {
2     // ... model reads and writes ...
3 }
```

Listing 5.1: Java try-with-resources for activity scopes

capturing the activities executed by agents. There are also several aspects to collecting the data required to describe a model element as an entity. The following section describes the design of model instrumentation.

Observing model interactions (LRQ 1.3)

A system's runtime model must include some supportive features to allow Cronista to describe accesses to the runtime model using entities. These supportive features will aid the detection of model elements being accessed, and their identification. Given the use of EMF for creating a system model, we know the EMF code generator implements the model using an *EObject* Java interface. Extending *MinimalEObjectImpl* (the minimal implementation of this interface in EMF) is one way to add custom functionality to an EMF model's implementation. The initial model instrumentation will seek to extend EMF's *MinimalEObjectImpl*, and this class extension will be reusable with other EMF models based on *MinimalEObjectImpl*.

The instrumentation to be included in the extended *MinimalEObjectImpl* will address the 3 requirements for provenance collection: detecting model element accesses, collecting model element information for correlation, and versioning.

Model element access (LRQ 1.3) The EMF Notification API was initially considered for sending notifications to an Observer when the model was accessed. Unfortunately, it does not send notifications for accesses that read a model (calls to *eGet* or *eDynamicGet*). Thus, notifications for recording provenance would result in only the write model interactions being recorded (calls to *eSet* or *eDynamicSet*). The solution was to create a custom access detection method: this is applied to a model by the model code generator using an extended version of the *MinimalEObjectImpl* class.

To reliably detect model accesses, we need to ensure that all access occurs through a pair of common accessor methods. Cronista requires the *dynamic feature delegation* flag to be set on the EMF code generator configuration. This configuration causes the code generator to produce model code that contains *eDynamicGet* and *eDynamicSet* methods. These methods are invoked when the

system model is accessed, thus providing ideal interception points for the instrumentation code.

Cronista provides a customised *MinimalEObjectImpl* (*LoggingMinimalEObjectImpl*) to override the default *eDynamicGet*/*eDynamicSet*; the customisation adds instrumentation code to notify the Observer about model accesses. Upon model access, the Observer is notified of the type of model access (read/write), the ID of the model element (object-attribute pair), its version, and its value. Model writes include both the old and new versions and values. The specific details of object correlation and versioning are discussed later in this section.

Instrumenting the model's accessor methods for intercepting model accesses revealed an unexpected source of access noise. This noise can occur if the code that captures the access has to call the same accessor methods to retrieve attribute values for provenance: without appropriate safeguards, this could start an endless loop of intercepted model accesses.

Model element correlation (LRQ 1.3) When a model access is detected, the Observer inspects the access to identify the model object and feature (object-attribute pair) being accessed as a unique *model element*. Model elements are represented using an ENTITY on the provenance graph. All accesses to the same model element must be correlated and recorded as related to the same ENTITY. EMF provides a way of identifying the model elements using its internal ID system for *EObject*s and features (attributes or references).

Identifying model objects and features is not a trivial task. A system's runtime model is implemented as an EMF model, which uses the abstract syntax defined in another EMF model (the metamodel). Navigating the EMF framework requires some working knowledge and experimentation to find a reliable way to identify the model elements being accessed by agents.

Model objects can be identified using the EMF Persistence API [1]. The API provides access to the URIs which identify resources, objects and packages. A URI and fragment are used to reference the *EObject*s inside a resource, making it possible to access an *EObject* without traversing the model's full tree structure. The API provides a method *getURIFragment(EObject)* which returns the fragment as a string for a given *EObject*. A runtime model object being accessed by an agent can be identified with a call to this method.

To identify the attribute of a model object that an agent accessed, EMF provides feature identifiers. Feature IDs [2] are integer constants assigned to the features of an *EClass* by the EMF code generator.

[1] EMF's Persistence API is described on page 447 in the Eclipse Modeling Framework book [104].
[2] EMF Feature IDs are described on page 278 in the EMF book [104].

As they are part of the code generated by EMF, feature IDs should remain consistent at runtime. However, design-time changes to the runtime model classes and subsequent code regeneration may change the feature IDs. EMF's reflective API grants access to the *EStructuralFeature* for a model attribute being accessed, which provides additional metadata like their name or data type. The feature ID of a model attribute can be captured in the *eDynamicGet*/*eDynamicSet* method call that occurs within the *EObject* implementing the model object.

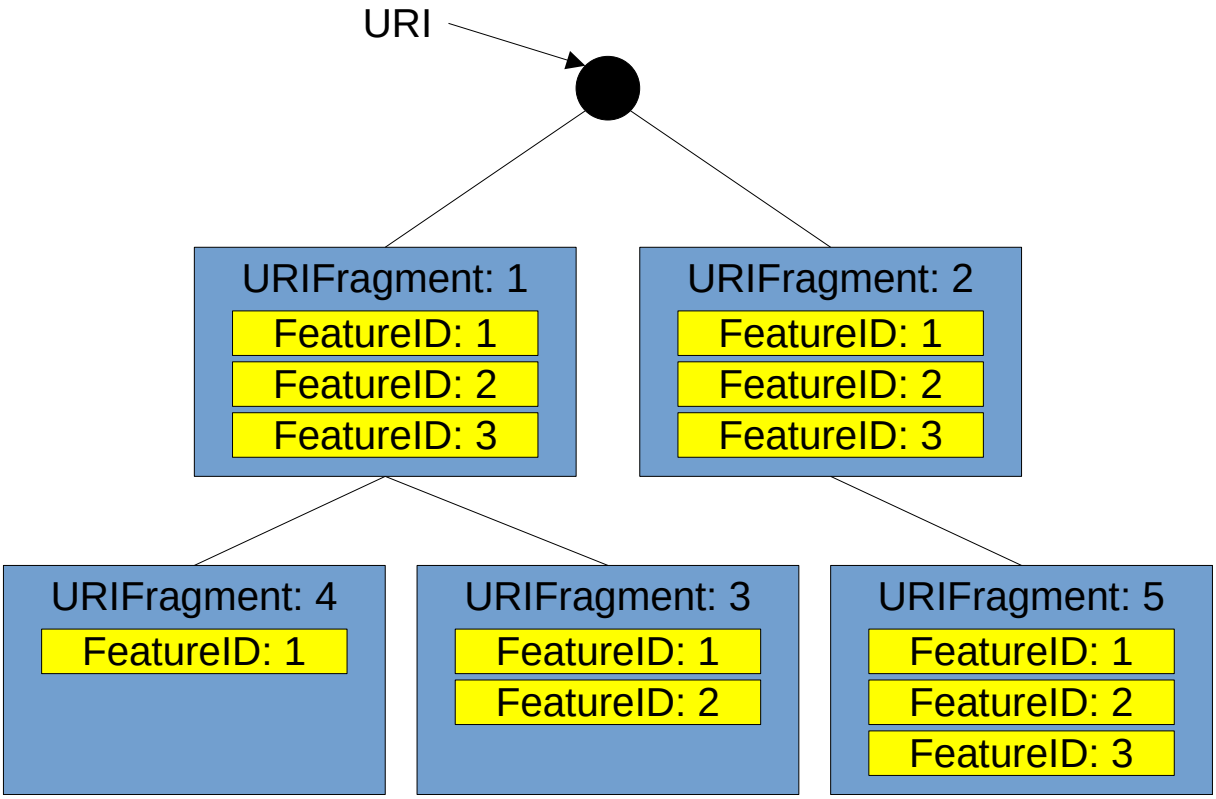

Figure 5.2: Containment tree of a runtime model, with the EMF Resource object at the root and model objects as blue boxes, showing URI fragments for model objects. Yellow boxes are features (e.g. attributes), showing their feature IDs.

Figure 5.2 shows a runtime model as a tree structure. A URI identifies the EMF *Resource* at the root, each model object is represented as a blue box, and yellow boxes are the model object features. Given a *URIfragment* and a *FeatureID*, it is possible to identify each model element (object-attribute pair) in such a runtime model.

Model element versioning (LRQ 1.3) In this initial implementation of Cronista, the system model is a native EMF model, that is only kept in memory (for simplicity). A version controller was provided to track model elements, with sequential version numbers for each change made to a model element.

With the initial single-threaded (i.e. single-agent) system, model versioning could be tracked by the thread Observer, as no other agents could change the model. However, in a large multi-threaded system the versioning system needs to account for all agent model accesses, these accesses must also be thread-safe. The problems with versioning an EMF model in these multi-thread systems will need addressing in the next design cycle (Section:5.2).

Additional observations The modification of the *eDynamicSet* method is slightly more involved, as it must collect provenance data before and after the model feature is set. When the Observer reports a model access to the Curator, it cannot be assumed that the Curator will have prior knowledge of the reported model element. The provenance system architecture [52] identified the need for "provenance recording to be stateless". To achieve this, the Observer reports to the Curator two ENTITIES when a model attribute is changed: one ENTITY describes the model element before the change, and another ENTITY describes it after the change. This reporting of before/after states enables the Curator to fully record a model element change as two ENTITIES connected by a *"wasDerivedFrom"* relationship, without prior knowledge of the model element state. Between inspection of the old and new model attributes, the *eDynamicSet* method triggers an increment of the attribute version number if the model attribute change is successful.

This design addresses the minimum information required to create provenance representations for AGENTS performing ACTIVITIES with model elements (ENTITIES). The provenance instrumentation for collecting provenance data can be further extended to collect more detailed model information. For example, the model objects and features might have name labels, including these on a ENTITY could make the provenance graph easier to read/query. The type of detailed model information recorded in provenance depends on the specific use case of the system and provenance system.

Implementation of Curator components (LRQ2.1)

The Curator processes and stores the provenance messages from Observers into provenance graphs on time windows. The time windows are arranged chronologically in the History Model. The Curator operates in its own thread as a loop that processes messages from a blocking queue that receives Observer messages. This concurrent provenance Curator process avoids causing delays to the system's execution when performing slow operations, like writing to disk storage.

Messages from the Observer(s) contain all the information the Curator requires to create a

provenance graph representation. As such, the Observer's messages contain duplicate data, which provides some mitigation for lost messages that would cause gaps in the provenance graph. As the Curator processes the messages, the duplicate information is removed as existing graph nodes are re-used whenever possible. This re-use of existing nodes causes the Curator to connect the fragments of provenance in each message together.

Reusing existing graph nodes could result in considerable amounts of graph searching, potentially slowing the Curator's graph writing process. To solve this problem, the Curator maintains a transient in-memory index from the IDs of the AGENTS, ACTIVITIES and ENTITIES to storage references, as it creates them. Tracking the IDs for each provenance node can help to reduce the delays by removing provenance graph searches; the in-memory index is quick to search compared to a search involving storage I/O. However, to avoid excessive memory consumption, this index is discarded when the current time window is closed; as the new (empty) time window will not contain any nodes, making the index information redundant.

The Curator must process messages from the queue quickly to keep pace with the messages from a system being observed. Therefore, storage I/O time costs need careful management. I/O time cost will depend on the underlying technology that is used for storing the History Model. One approach to managing storage I/O time costs is to reduce the number of small storage operations, as each I/O operation includes some small time overhead on top of the data transfer time. One way to reduce the number of small storage operations is to create the time window's provenance graph in memory, then write to the History Model Store when the time window ends in a single large Storage I/O. This in-memory approach to building a provenance graph is used with Cronista's CDO History Model Store.

A History Model Store is a simple data repository: the data structure of the history model is created by the Curator. The implementation of the History Model Store is at the end of the section, as the Curator processes described should be agnostic to the technology used to store a history model. The following sections describe the Curator's behaviours/processes for creating a history model composed of time windows, provenance graphs, and snapshots.

Provenance message processing

The logic for processing the 4 types of messages the Curator receives is largely the same, regardless of the storage technology. A future developer's work to create a History Model Store using a

different storage technology can be reduced to implementing storage adaptors. A developer would need to implement storage adaptors for basic functions such as adding a node or creating an edge/relationship between two nodes. All the logic for processing each provenance message type (Section 4.1.4 page 100) is handled by the Curator. This logic is as follows:

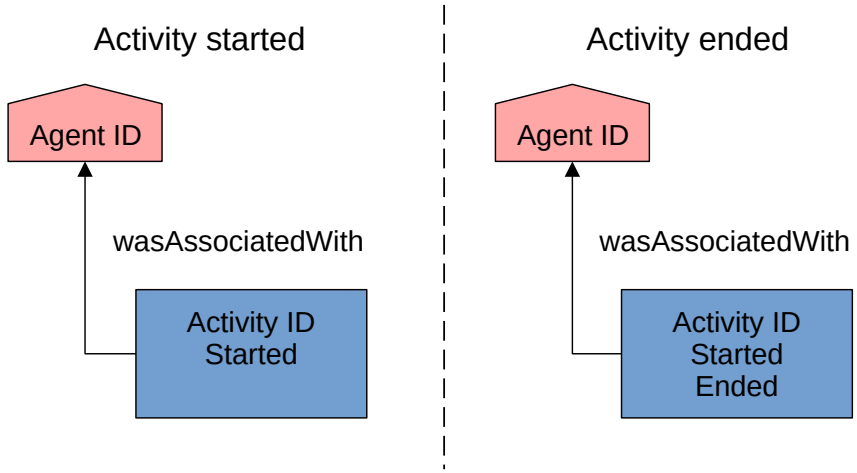

Figure 5.3: Provenance representations for starting and ending activities (repeated from page 101)

Activity started The Curator ensures an ACTIVITY exists: this will be a new node, as the Activity IDs are unique. If this activity is an agent's first, then a new AGENT is created for the agent. Finally, the AGENT and ACTIVITY are related with a *"wasAssociatedWith"*, as shown in Figure 5.3.

Activity ended If there is an existing ACTIVITY node with a matching ID on the time window, the Curator updates the node to reflect the additional end details (e.g. end time). The graph produced would conform to Figure 5.3. However, if an activity start message was missed by the Curator (for example, due to network failures), then the subgraph for the activity start would be missing. As such, the end activity message would repair the missing subgraph, as new nodes/relationships would be generated from the end message which would include the activity start details.

Model read This produces the graph in Figure 5.4, which identifies the attribute depicted with an ENTITY node, and states that it is *"used"* by an ACTIVITY that *"wasAssociatedWith"* an AGENT. Again, a minimal model write might be needed if all 3 nodes are reused: in that case, only the *"used"* relationship needs to be added between an ENTITY and ACTIVITY. Finally, an optional relationship (WASINFORMEDBY) can be added between the current ACTIVITY contained in the provenance

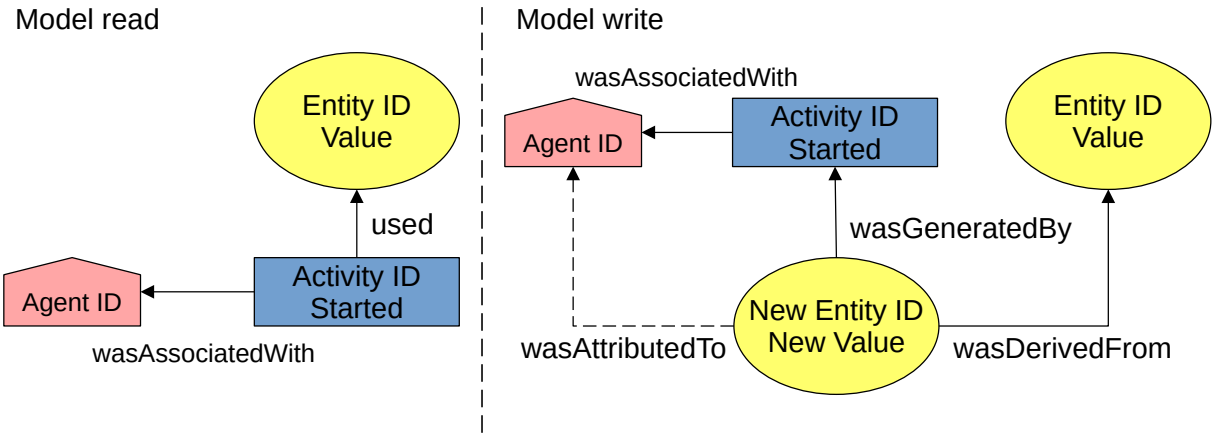

Figure 5.4: Model read/write PROV-DM descriptions (repeated from page 102)

message and the ACTIVITY that produced the *"used"* ENTITY.

Model write The Curator ensures that nodes for the AGENT and ACTIVITY in the message are on the graph. Next, the ENTITIES to reflect the new and old model element versions are created. Typically, an ENTITY should exist for the old attribute, and so it would be reused and not re-created unless it was missing in the current context (e.g. existed on a prior time window). The smallest model write for a message only creates one new ENTITY node for the new attribute and its appropriate relationship edges, as seen in Figure 5.4.

Time windows and snapshots

The Curator's memory consumption is related to the size of the provenance graph it builds in memory, and the amount of memory a host system provides limits how big a provenance graph a Curator can build. A provenance graph on a time window will become exceptionally large over time, or if a busy system rapidly produces messages; large provenance graphs cause problems as discussed in Section 4.2.2. Time windows simplify the management of the Curator's memory consumption: each time window represents a block of execution time passing, and as they are not dependent on each other, they can be unloaded/loaded from memory as required.

A system's long-running activities may span multiple windows, which presents problems with how provenance information is distributed over time windows (Section 4.2.4 on page 111). Cronista implements Option B as discussed in that section: provenance messages are recorded to the current time window, which distributes the provenance of a long-running activity over multiple time windows. Option A would introduce a complex challenge with Curator memory management: a long-running

activity would block a time window from unloading and freeing memory. Using Option B for Cronista's Curator creates a design that predictably releases memory consumed by a provenance graph when a time window closes.

The Curator can open new time windows when required, and continue to process any outstanding provenance messages in the queue into the new time window. No provenance information is lost, but there is a slight increase in storage cost as the provenance graph for the new time window will not be able to reuse nodes from the provenance graph for the previous time window. However, this enables different strategies for deciding when a Curator should create a new time window, assuming a snapshot of the runtime model can be taken when a new time window is opened. Triggers for a new time window could be based on the Curator's memory consumption, the number of nodes in a provenance graph, the passage of time, or the number of messages received from the Observer.

In-memory EMF model snapshots (LRQ 1.4) Cronista's Observer sends a message to the Curator to create a new time window. The Observer leads this process due to the limitations of the system model being a native in-memory EMF model, which does not have a snapshot facility. There are tools and facilities for copying EMF models, but some copy processes can result in new URIs and fragments on the model copy, which would not match the provenance graph entities. An easy approach to snapshot an in-memory EMF model is to serialise it to a file. The Observer can provide this facility if a system cannot do this, the system processes should not access or change the model while the Observer writes the model to a file. This approach of saving a model to file preserves the internal EMF model URIs and fragments. The Observer messages the file name and path for the saved model file to the Curator; who stores them in the History Model as the Time Window's snapshot; i.e. the Time Window's snapshot is a reference to the file containing the saved model on a file system.

Implementation of the History Model Store (LRQ 2)

Different technologies could be used to store the history model, and each storage technology has its own ecosystem of tools for exploring data. The Curator is designed to be connected with different storage technologies for storing the history model; this was achieved by using an Adaptor design pattern. The behaviour of the History Model Store is kept consistent for the Curator. As such, the logic for curating the History Model (creating the provenance graph and Time Windows) remains the same across different technologies that could be used to create a History Model Store. Each History

Model Store implementation interacts with the specific technology's API on behalf of the Curator (e.g. adding nodes, or building a relationship between two given nodes).

The initial History Model Store was built using the open-source Eclipse CDO [28] model repository. The CDO project is under active development, and has all the features required by Cronista. It also has native support for EMF models, enabling the system model and a History Model (including an implementation of PROV-DM) to be conveniently integrated as they are both implemented in EMF. CDO is likely a familiar technology for developers who use EMF.

5.1.5 Demonstration

To demonstrate the above answers to the LRQs, this section will present some Java code examples for the instrumentation components that have been implemented. The section ends with an example of provenance collection from a system that calculates the Fibonacci sequence using a runtime model.

Code examples

Code examples are provided for the instrumentation applied to a system to enable provenance awareness. The full code listings are available in the GitLab repository [94].

Agent Observer (LRQ 1.1) There needs to be one instance of an Observer (Object) in a Java thread. The Singleton design pattern can be used to ensure only one instance is created of an object. The approach in Listing 5.2 creates one *ThreadObserver* instance within the scope of a Java thread. Within the *ThreadObserver*, an *AgentDescription* object is used to store the details of the agent. A minimal implementation requires providing an external AgentID for provenance.

Activity scopes (LRQ 1.2) Listing 5.3 shows the *ActivityScope* class that is instantiated within a Java *try-with-resource* block to mark out activities in code (Listing 5.1, page 128). The *ActivityScope* class implements the standard Java *AutoClosable* interface required to use it within try-with-resource initialisation lists, and has been designed to make the close method idempotent. When an instance of *ActivityScope* is created, it informs the thread's *ThreadObserver* about the start of an activity. When execution leaves the scope of the try-with-resource, the *ActivityScope* close method is invoked, which informs the *ThreadObserver* about the end of the activity: the Observer will remove the activity from

```
1  public class AgentDescription {
2      private final String id;
3      public AgentDescription(String agentID) { ... }
4      public String getID() { ... }
5  }
6
7  public class ThreadObserver {
8      private final AgentDescription agentDescription;
9
10     public class ThreadObserver {
11         public static final ThreadLocal<ThreadObserver> INSTANCE = new ThreadLocal<>() {
12             @Override
13             protected ThreadObserver initialValue() {
14                 return new ThreadObserver(Thread.currentThread().getName());
15             }
16         };
17     }
18
19     private ThreadObserver(String name) {
20         this.agentDescription = new AgentDescription(name);
21         System.out.println("An Observer is watching " + name);
22     }
23     ...
24  }
```

Listing 5.2: Thread-local Observer and Agent identification

its stack. The *ThreadObserver* sends messages to the Curator for the start and end of the activity, while it manages the elements of the stack.

Model element access (LRQ 1.3) Listing 5.4 shows the *LoggingMinimalEObjectImpl* code that extends *MinimalEObjectImpl*. Both the *eDynamicGet*/*eDynamicSet* access methods have been overridden to detect model accesses. The detection can distinguish between system and inspection model accesses using a simple boolean flag. This boolean flag is stored in the *ThreadObserver*; the Observer should only inspect one model access at a time. Wrapping the location of the inspection code with an *if* statement prevents recursion if the inspection code triggers subsequent model accesses. After the inspection code has executed, a *super* call invokes the original methods for *MinimalEObjectImpl*: this preserves the existing EMF model's functionality and ensures the process of collecting provenance is transparent. The model extensions applied must behave like the *wrappers* described in the provenance system architecture [52].

Model element correlation (LRQ 1.3) Listing 5.5 shows the *eDynamicGet* instrumentation with code added for model object correlation, enabling Cronista to identify unique model elements. The

```
1  public class ActivityDescription {
2      private final String id;
3      private final String name;
4      private final long startTime;
5      private long endTime;
6      public ActivityDescription(String id, String name, long currentTimeMillis) { ... }
7      // ... Accessor methods ....
8  }
9
10 public class ThreadObserver {
11     private final Deque<ActivityDescription> activityStack = new ArrayDeque<>();
12     public synchronized void startActivity(String activityName) {
13         ActivityDescription newActivity = new ActivityDescription( ... );
14         activityStack.push(newActivity);
15         sendMessage(new ActivityStartedMessage( ... ));
16     }
17     public synchronized void endActivity() {
18         if (!activityStack.isEmpty()) {
19             ActivityDescription myCurrentActivity = activityStack.pop();
20             myCurrentActivity.setEndTime(System.currentTimeMillis());
21             sendMessage(new ActivityEndedMessage( ... )); }
22     }
23     ...
24  }
25
26 public class ActivityScope implements AutoCloseable {
27     // In order to make close() idempotent
28     private boolean open = true;
29     public ActivityScope(String activityName) {
30         if (ProvLayerControl.getProvLayerControl() != null) {
31             final ThreadObserver obs = ThreadObserver.INSTANCE.get();
32             obs.startActivity(activityName); }
33     }
34     @Override
35     public void close() // throws Exception
36     {
37         if (ProvLayerControl.getProvLayerControl() != null) {
38             if (open) {
39                 final ThreadObserver obs = ThreadObserver.INSTANCE.get();
40                 obs.endActivity();
41                 open = false; } }
42     }
43 }
```

Listing 5.3: Observer Activity scope components

```
1  public class LoggingMinimalEObjectImpl extends MinimalEObjectImpl {
2
3      @Override
4      public Object dynamicGet(int dynamicFeatureID) {
5          if (theThreadObserver.getSystemCallFlag()) {
6              // MODEL INTERACTION LOGGING OFF
7              theThreadObserver.setSystemCallFlag(false);
8
9              // ... INSPECTION CODE GOES HERE ...
10
11             // MODEL INTERACTION LOGGING ON
12             theThreadObserver.setSystemCallFlag(true);
13         }
14         return super.dynamicGet(dynamicFeatureID);
15     }
16
17     @Override
18     public void dynamicSet(int dynamicFeatureID, Object newValue) {
19         if (theThreadObserver.getSystemCallFlag()) {
20             // MODEL INTERACTION LOGGING OFF
21             theThreadObserver.setSystemCallFlag(false);
22
23             // ... INSPECTION CODE GOES HERE ...
24
25             // MODEL INTERACTION LOGGING ON
26             theThreadObserver.setSystemCallFlag(true);
27         }
28         return super.dynamicSet(dynamicFeatureID, newValue);
29     }
30 }
```

Listing 5.4: Customised EObjectMinimalImpl to enable detection of model interactions

```
1  public class LoggingMinimalEObjectImpl extends MinimalEObjectImpl {
2
3      @Override
4      public Object dynamicGet(int dynamicFeatureID) {
5          if (theThreadObserver.getSystemCallFlag()) {
6              // MODEL INTERACTION LOGGING OFF
7              theThreadObserver.setSystemCallFlag(false);
8
9              // Get a unique identity for the model object
10             String EObjectID = EcoreUtil.getIdentification(this);
11
12             // Get the identity for this model object feature
13             EStructuralFeature dynamicFeatureIDObject =
14                 this.eClass().getEStructuralFeature(dynamicFeatureID);
15
16             // MODEL INTERACTION LOGGING ON
17             theThreadObserver.setSystemCallFlag(true);
18         }
19         return super.dynamicGet(dynamicFeatureID);
20     }
21
22     @Override
23     public void dynamicSet(int dynamicFeatureID, Object newValue) { ... }
24 }
```

Listing 5.5: Customised EObjectMinimalImpl model object correlation

EcoreUtil class from EMF provides a method to obtain the identification for an EObject. In this example, the *EStructuralFeature* being accessed is assigned to a variable *dynamicFeatureIDObject*: this can be used to retrieve more detailed information for provenance collection, such as the name of the feature.

Model element versioning (LRQ 1.3) When the technology used to implement a runtime model does not include a versioning system, there is a need for the instrumentation to provide one. Native EMF models lack versioning, so a minimal fallback implementation was created. The approach taken extends the model's EObjects with a mapping from their features to version numbers (Listing 5.6).

Versioning and value extraction can now be added to the model instrumentation code in the *LoggingMinimalEObjectImpl*. Listing 5.7 shows an example of the modifications required for *eDynamicGet*. Values for the model features are handled as generic *Object*s, as their actual type will depend on the system model design. Later stages of provenance management or querying will need to examine the value to discover the data types. A call to an Observer method to create a provenance message is needed on line 27 of Listing 5.7 before returning the model feature to the system agent.

```
1  public class VersionController {
2
3      // HashMap <Key:modelFeature Value:memoryVersion>
4      private HashMap<String, AtomicInteger> versionMemoryMap = new HashMap<>();
5
6      public AtomicInteger getVersion(ModelFeature modelFeature) {
7          // ... Add model feature to table if not present ...
8          // ... Get model feature version ...
9          return version;
10     }
11
12     public AtomicInteger incrementVersion(ModelFeature modelFeature) {
13         // ... Add model feature to table if not present ...
14         // ... Increment model feature version ...
15         return version;
16     }
17 }
```

Listing 5.6: Simple in-memory EObject model feature version controller

The Observer's method code for creating provenance messages (containing the agent, activity and entities detailing new/old model element, version, and value) has not been listed here, but can be found in Cronista's Git repository [94]. A provenance message is a simple data structure containing a set of *EObject*s that represent the PROV-DM concepts the Curator needs to store.

When a call to *eDynamicSet* occurs (Listing 5.8), the instrumentation code increments the version number of the accessed model element. If the table does not contain an entry for an element, then one is added with an initial version number of 1.

Fibonacci example (LRQ 2.2)

The development of the instrumentation to enable provenance awareness of agents, activities, and entities involved short development cycles, using manual testing to ensure expected behaviours. Once the components proved reliable and could obtain consistent data for provenance collection, their integration was tested with a larger experiment on a small system that calculated the Fibonacci sequence. The experiment was designed to demonstrate that Cronista could track the provenance of a known process, and produce a provenance graph that represented its workflow. Calculating the Fibonacci sequence is a recursive process, where results from a calculation are reused in the next. These values are stored in a runtime model implemented using EMF and Cronista's *LoggingMinimalEObjectImpl*. The Fibonacci calculating sequence executes in a single thread, with its own *ThreadObserver*.

```
1  public class LoggingMinimalEObjectImpl extends MinimalEObjectImpl {
2
3      private VersionController versionController;
4
5      @Override
6      public Object dynamicGet(int dynamicFeatureID) {
7          if (theThreadObserver.getSystemCallFlag()) {
8              // MODEL INTERACTION LOGGING OFF
9              theThreadObserver.setSystemCallFlag(false);
10
11             // Get a unique identity for the model object
12             String EObjectID = EcoreUtil.getIdentification(this);
13
14             // Get the identity for this model object feature
15             EStructuralFeature dynamicFeatureIDObject =
16                 this.eClass().getEStructuralFeature(dynamicFeatureID);
17
18             // Get feature version
19             AtomicInteger version = versionController.getVersion(dynamicFeatureIDObject);
20
21             // Get value as an Object (unknown type)
22             Object value = super.dynamicGet(dynamicFeatureID);
23
24             // MODEL INTERACTION LOGGING ON
25             theThreadObserver.setSystemCallFlag(true);
26         }
27         // REPORT THE COLLECTED PROVENANCE INFORMATION
28         return super.dynamicGet(dynamicFeatureID);
29     }
30
31     @Override
32     public void dynamicSet(int dynamicFeatureID, Object newValue) { ... }
33
34 }
```

Listing 5.7: Customised EObjectMinimalImpl model object versioning for get

```
1  public class LoggingMinimalEObjectImpl extends MinimalEObjectImpl {
2
3      private VersionController versionController;
4
5      @Override
6      public Object dynamicGet(int dynamicFeatureID) { ... }
7
8      @Override
9      public void dynamicSet(int dynamicFeatureID, Object newValue) {
10         if (theThreadObserver.getSystemCallFlag()) {
11             // MODEL INTERACTION LOGGING OFF
12             theThreadObserver.setSystemCallFlag(false);
13
14             // Get a unique identity for the model object
15             String EObjectID = EcoreUtil.getIdentification(this);
16
17             // Get the identity for this model object feature
18             EStructuralFeature dynamicFeatureIDObject =
19                 this.eClass().getEStructuralFeature(dynamicFeatureID);
20
21             // Get feature version before set
22             AtomicInteger oldVersion = versionController.getVersion(dynamicFeatureIDObject);
23
24             // Get value as an Object (unknown type) before set
25             Object oldValue = super.dynamicGet(dynamicFeatureID);
26
27             // Set the feature value to new value
28             super.dynamicSet(dynamicFeatureID, newValue);
29
30             // Update the feature version (assuming set sucessful)
31             AtomicInteger newVersion = versionController.setVersion(dynamicFeatureIDObject)
32
33             // MODEL INTERACTION LOGGING ON
34             theThreadObserver.setSystemCallFlag(true);
35         }
36         // REPORT THE COLLECTED PROVENANCE INFORMATION
37         return
38     }
39
40 }
```

Listing 5.8: Customised EObjectMinimalImpl model object versioning for set

```
1  public static void fib() {
2      Stringy Console; Inty A, B, C;
3      try(var sc0 = new ActivityScope("init")) {
4          Console = GPLDataFactory.createStringy();
5          A = GPLDataFactory.createInty();
6          B = GPLDataFactory.createInty();
7          C = GPLDataFactory.createInty();
8          try(var sc1 = new ActivityScope("init-names")) {
9              A.setName("A"); B.setName("B"); C.setName("C");
10             Console.setName("Console");
11         }
12         try(var sc1 = new ActivityScope("init-values")) {
13             A.setValue(0); B.setValue(1); C.setValue(0);
14             Console.setValue(null);
15         }
16     }
17
18     try(var sc0 = new ActivityScope("Loop condition")) {
19         do {
20             try(var sc1 = new ActivityScope("A + B")) {
21                 C.setValue(A.getValue()+B.getValue()); }
22             try(var sc1 = new ActivityScope("B to A")) {
23                 A.setValue(B.getValue()); }
24             try(var sc1 = new ActivityScope("C to B")) {
25                 B.setValue(C.getValue()); }
26             try(var sc1 = new ActivityScope("Update console string")){
27                 Console.setValue(C.getName() + " is " + C.getValue() + "\n"); }
28             try(var sc1 = new ActivityScope("Display console output")) {
29                 System.out.println(Console.getValue()); }
30             // ... continued ...
31         } while (A.getValue() < 255);
32     }
33 }
```

Listing 5.9: Activity scopes applied to a Fibonacci code example

Listing 5.9 shows an example of the activity scopes defined in the Fibonacci program. Lines highlighted in blue (Java try-with-resources blocks) have been added to make the activity scopes. When the code executes, each try-with-resources block creates an appropriately named `ActivityScope` that the Observer tracks. The Observer handles timestamp collection and messaging when scopes are added or removed from the Observer's stack. This shows how the developer only needs to add these try blocks with a descriptive name. The activity scope covers all code within the body of the *try* statement.

As the developer wraps lines of code, they need to consider the level of provenance detail required from a loop in their code. In this example, the outer loop (line 17) defines an activity for evaluating the loop condition. Each step within the loop is a different nested activity, producing fine-grained provenance with an ACTIVITY node per iteration of the loop. If the inner processes of a loop need hiding, then the nested activities could be omitted. In that case, the activity scope for the loop would collect all model interactions as ENTITIES and associate them with a single ACTIVITY node. In other cases, provenance collection would not be practical as it would produce too much data: the loop could be omitted from all activity scopes.

The instrumented Fibonacci program was executed, and provenance was collected to a CDO-based History Model Store. An attempt was made to explore the history model from the Eclipse IDE using the existing CDO Explorer view. However, the CDO Explorer did not provide a suitable big-picture view of the provenance graph. Its interface produces a form-based view of a model, i.e. each provenance graph node can be viewed individually. A tool was developed to enable visualisation of the provenance in a notation closer to the W3C PROV approach, which is described below.

Graphviz-based provenance visualisation This visualisation is needed to provide a general view of the workflow of the Fibonacci system's execution. A small Java program was created to perform a model-to-text transformation from the history model in CDO to a Graphviz [37] DOT file. The transformation program reads in a provenance graph from a time window to write an equivalent graph in DOT file notation. The resulting file was then visualised as a graph, using Graphviz.

The initial graphs produced were difficult to read, and it was necessary to add a "then" relationship to arrange the activity nodes in order of execution, to improve readability. Small changes like adding a "then" relationship can be introduced during a model transformation, without altering the original model. The transformation process can automate some of the effort required to lay out a provenance

graph for easier reading. Finally, the resulting Graphviz files can be exported to a vector graphics application (e.g. Inkscape [108]), and cleaned up further by hand. An example of a graph produced by this process is shown in Figure 5.5.

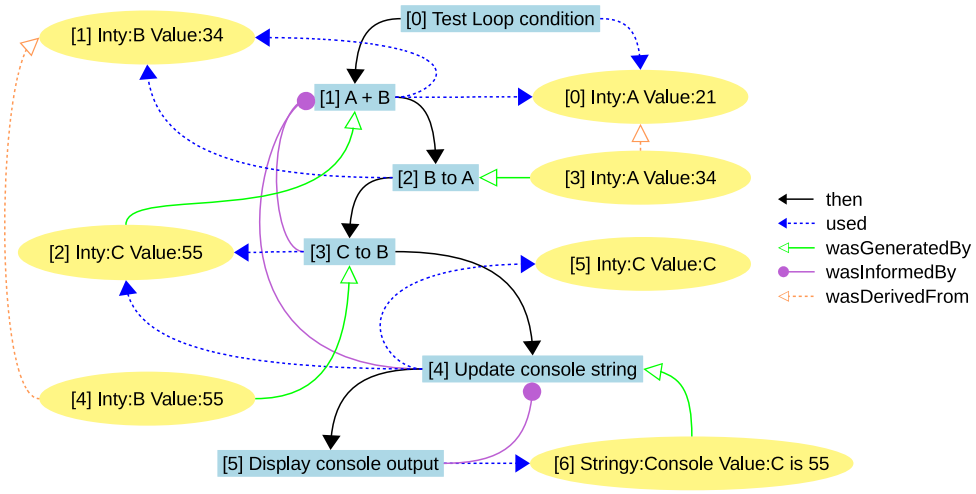

Figure 5.5: Graphviz render of a provenance graph from an execution of the Fibonacci system

Figure 5.5 shows the activity and entity interactions for one pass of the loop from lines 17 to 30 in Listing 5.9. Agent nodes are not shown, as the system contains a single agent (one thread). The activity labels are visible in the blue rectangle nodes. An interaction with the runtime model creates entity nodes as yellow ovals labelled with the object, attribute and value accessed. While traversing the activity nodes on the graph shown in Figure 5.5, the following statements can be deduced:

[0] A loop condition test used the value of A.

[1] intyA and intyB are used in an “add” activity to generate a value stored in intyC.

[2] intyB is used to generate a value stored in intyA.

[3] intyC is used to generate a value stored in intyB. This activity was informed by activity [1].

[4] The value and name in intyC is used to generate StringyConsole as an update. This activity was informed by activity [1].

[5] The value in StringyConsole is displayed. This activity was informed by activity [4].

Alternative model transformations of the provenance graph were explored to produce different graph visualisations using Graphviz. These model transformations left out one of the node types to produce a significant simplification of the graphs: Figure 5.6 excludes the entity nodes, and Figure 5.7

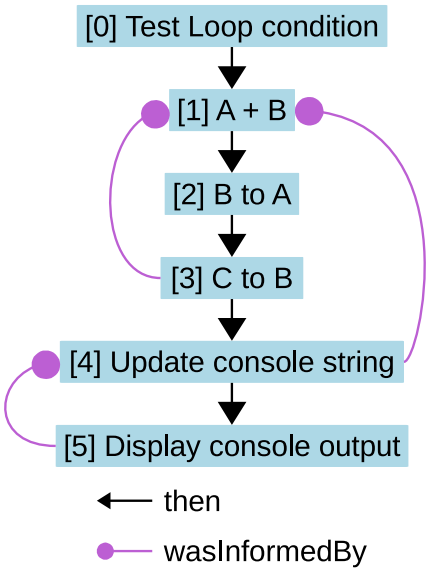

Figure 5.6: Graphviz render of only Activities on a provenance graph, after running the Fibonacci system

excludes the activity nodes. Figure 5.6 produces a series of activity nodes for the flow of the code execution, which could be useful for tracing execution. Figure 5.7 shows the sequence of changes to the runtime model that occurred: this is useful to track the evolution of the model states.

5.1.6 Evaluation

In this section the research questions raised in the objective (subsection 5.1.3) are answered. Following on from the answers to the research questions is a discussion of the lessons learned.

LRQ1 and sub-question answers

LRQ1 and sub-questions raised questions about how to implement the actor-side libraries for enabling provenance awareness, as mentioned in the provenance system architecture (Section 2.5.1, page 52).

LRQ1

How can a system be instrumented by extending the framework used for its shared-knowledge base, to enable a reusable approach to provenance awareness?

LRQ1 answer Model-driven engineering frameworks can be extended to include provenance awareness. These frameworks include several features such as reflection, object identification and

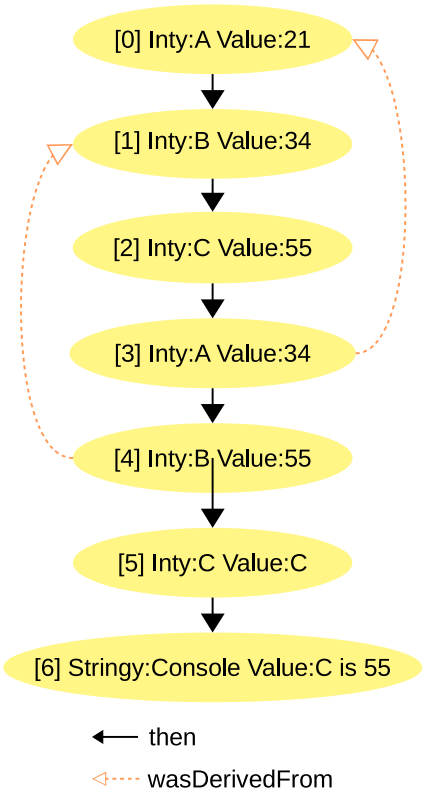

Figure 5.7: Graphviz render of only Entities on a provenance graph, after running the Fibonacci system

object persistence, which are required to enable a system with provenance awareness. Modelling frameworks may also include code generators and other code structures, that can be adapted to automate some provenance instrumentation processes, saving developer efforts.

Applying provenance awareness to a system through framework components enables other systems built using the same framework to be made provenance-aware for less developer effort, since a developer does not need to re-implement provenance awareness components. However, when there are variations in framework components (e.g. different model code implementation), some one-off customisation of the instrumentation may be required.

LRQ1.1

How can a developer instrument system execution threads (Agents), for provenance awareness?

LRQ1.1 answer The Java language and standard libraries include facilities for creating multi-threaded systems: Cronista's agent instrumentation targets these facilities. Cronista's agent instrumentation should work with other MDE frameworks and systems that use the same Java facilities.

LRQ1.2

How can a developer instrument a system to define where Activities start and end, for provenance awareness?

LRQ1.2 answer Activities in the system can be tracked through a mechanism implemented as an extension of Java's try-with-resource blocks. This Java feature provides a robust way to track execution entering/exiting sections of the system's code. The instrumentation for activities should be reusable with Java-based systems and MDE frameworks, because standard Java features have been used.

The activity scope approach enables one or more lines of code, including method calls, to be marked out as an activity for provenance collection. However, there is a need to refactor a system's code if two or more activities exist in a single line of code (which provenance should reflect). The line would require refactoring to several lines of code to enable separate scopes to be applied for each activity. However, refactoring a few lines of code for this edge case is probably preferable to alternative approaches, such as refactoring a system's code to align activity scopes with method boundaries.

LRQ1.3

How can a developer instrument a system's runtime model to record accesses to model elements (Entities), for provenance awareness?

LRQ1.3 answer Model instrumentation is achieved through an extension of the class used by the EMF code generator. Implementing these extensions is a one-time effort for a model's base class. These model instruments can be reused across multiple systems, where the same base class is used to implement the system's model.

Manipulation of the code generator automates the instrumentation of an entire model with full coverage for provenance awareness. A developer does not need to select which parts to cover, or manually add code to model accessor methods.

LRQ1.4

How can snapshots be taken that preserve the relationship between runtime model elements and entities?

Snapshots for a model are required to preserve the model IDs used for entity (Model Element) correlation and versioning. As such, MDE tools that enable the copying or cloning of models can be used, provided they do not change IDs. The initial implementation used existing EMF features to save a model to a file. A large model would likely take some time to commit to a file and could generate duplicate information, increasing storage costs (e.g. if multiple copies of an unchanged model were stored). Future development work should investigate MDE tools that could make a more efficient model snapshot system.

LRQ2 and sub-question answers

LRQ2 and its sub-questions sought answers for how the provenance system storage components should be implemented. These components are most likely to be easily reusable between systems, but the technologies used to implement them could ease some difficulties a developer may experience when adopting them.

LRQ2

How can a provenance store for a history model be implemented?

LRQ2 answer A history model store can be implemented using MDE, and the same representations created by the Curator can be persisted in a model repository. The model repository is used to store EMF objects that represent the features of the history model. CDO provides an EMF-compatible model repository, which commits the models to disk to persist them.

LRQ2.1

How can the concerns between the Curator and storage be separated, enabling the use of different technologies for history model storage without requiring changes to the Curator?

LRQ2.1 answer Curator messages contain EMF objects for provenance, and an adaptor separates the Curator from the CDO storage. The Curator process interprets the provenance messages from an Observer, which results in a provenance graph fragment. This fragment is created by rules in the Curator, which signal how the data will be stored via an adaptor. EMF objects from the messages are passed via the adaptor to the CDO database to create the provenance graph.

LRQ2.2

Could an existing model-driven engineering (MDE) technology be used to implement the storage for a history model to reduce development costs?

LRQ2.2 answer Using CDO to implement the storage of the history model meets the provenance store requirements in the provenance system architecture [52]. A provenance store requires the implementation of some interfaces (for querying, managing, recording) and components, and a model repository like CDO provides generalised implementations of these for models. These generalised model implementations enable querying, management and persistence (recording) for models; these can be leveraged as the provenance data is treated as a model.

LRQ2.3

How can the history model inside the storage component be accessed using existing technologies?

LRQ2.3 answer The provenance data is stored as an EMF model inside CDO. As such, CDO Explorer (a plug-in for the Eclipse IDE) can be used to view the content of a CDO repository. This provides a form-based user interface for working with model objects in a CDO repository. However, the model in CDO can be accessed and transformed with a program like any other EMF model (i.e. perform model-to-model or model-to-text transformations). A short program was written to transform the provenance graph model into a (DOT Language) text file for visualisation with Graphviz.

Discussion

In this section, the initial implementation of a software artefact was created using the conceptual designs presented in Sections 4.1 and 4.2. The software artefacts created share some similarities to

the provenance system architecture components, with the separation of concerns between system components being comparable. The discussion will review Cronista's agent-side libraries (Observer components) and provenance systems (Curator components).

Observer components To enable a system with Cronista's provenance awareness, some components need to be added to the system to observe its execution. These components record the processes occurring during system execution using the W3C PROV-DM notation. The data collected is intended to answer provenance questions about how a system interacts with its runtime model. In contrast, an approach to provenance awareness could be taken that does not add components to a system, instead relying on logs or similar data that a system natively exports; this data would be used to try and reconstruct the system's execution process and create the provenance recording.

Applying provenance awareness using Cronista's Observer and Curator provides a structured and controlled approach for collecting provenance data that describes a system's execution. A developer adds Cronista's components to their system and then proceeds to apply provenance awareness by identifying the following in a system:

1. Threads of executions as agents performing activities.
2. Activity scopes in the system code that relate to the system processes performed by its agents.
3. The system's code implements the shared-knowledge (runtime model).

Cronista's Observer instrumentation detects the interaction between these identified system agents, activities and runtime models. The interaction is then automatically described in a provenance message to a Curator for recording. A developer does not need to consider how they should define a system's execution in a log description. An instance of an Observer is created per thread that requires provenance awareness, and several observers can report to one Curator via the same message queue.

A system may need to be enabled for provenance awareness to address some specific needs or explanation requirements, as described in PLEAD [60] (Section 2.5). Cronista could be used to implement provenance awareness that collects the provenance data for explanation requirements, assuming that the provenance of agents, activities and runtime model interactions addresses a facet of provenance questions in those requirements. To do this, a developer does not need to create an agent view of the system. Instead, Cronista's provenance awareness is based on observing and

reflecting on the abstractions within the system. The explanation requirements may drive changes to a system's design - for example, if a provenance requirement identifies an explanation that needs information, but the system's processes and shared-knowledge do not yet contain a representation of that information. Addressing the missing representations or modifying a system's processes to meet explanation requirements could be described as improving a system's explainability.

Agents and activities The identification and instrumentation of these components are based on standard Java libraries. Agents are based on Java's multi-threading libraries and therefore should be transferable to other Java-based systems using the same libraries. In the current design, there is a need for a separate instance of the Observer component per thread: this makes a simple 1-to-1 Observer-agent pairing. The Observer can only detect activities and model interactions occurring in the same thread, thus, an Observer reports using a single agent ID.

For Cronista to track activities in a system, there was a need to implement a mechanism for developers to be able to mark the start/end of activities in code. An approach was created using Java's try-with-resources blocks, which enables a developer to apply activity scopes like code annotations. A system's execution thread of execution passes through activity scopes which create tokens on a stack within the Observer. The Observer can examine the top of the stack when creating a provenance message to determine the agent's current activity.

During the implementation of the activity scopes and stack, there was a concern that execution could unexpectedly exit an activity scope; for example, unhanded exception errors. The automated resource acquisition and release from try-with-resource blocks helped with this problem, as their resource is always released once acquired regardless of whether the body of the block succeeds or fails. In this case, the resource being released is interpreted as the activity having ended. Overall, the approach seems sufficiently robust to address minimal requirements for relating agent activities with sections of code for provenance awareness.

Model instrumentation The approach to model instrumentation to observe model interactions with Cronista requires a runtime model to be built using a modelling framework and model code generator. This dependency on a modelling framework limits the use of Cronista to developers who build systems using model-driven software engineering practices and tools. However, within the scope of MDE, Cronista's components could be reused or adapted. It is possible that Cronista could

be used with other modelling frameworks and code generators, which may also extend to different programming languages other than Java.

An increasing set of components for different modelling frameworks and languages could be implemented once for Cronista. These components could then be re-used across multiple systems built with the same modelling framework. There is a requirement that a modelling framework permits the extension or customisation of the code generator.

Instrumenting a systems model through a model code generator keeps provenance awareness features out of a system model's design. A system's model design does not need to be changed to include provenance-specific features; for example, IDs for model element correlation. The provenance collection modifications are applied via the code generator and only exist in the code implementation of the model. Furthermore, the changes made to the model code are transparent from a system's code perspective; the same model get/set methods are used and maintain their original behaviour.

Curator and history model storage Using MDE to implement PROV-DM and history model data structures enabled the use of existing MDE technologies for Cronista. The available ecosystem of EMF-based MDE tools reduced the development efforts for implementing a provenance system.

CDO was selected to implement the first history model store, as it can be used with EMF-based models. With the Observer and system models using EMF models, extending the use of EMF to the Curator and history model implementations seemed logical. The messages containing provenance data from the Observer are EMF objects, which the Curator can commit to the history model in a CDO repository with minimal changes. CDO handles the movement of EMF objects between memory and disk seamlessly; the complexity of implementing the Curator was reduced to the provenance message processing logic (Section 5.1.4).

Model snapshots Cronista can be used with an in-memory EMF system model, which does not have a native snapshotting facility: it is only possible to save the model to a file, which is limited in functionality and scalability. Large in-memory models might be slow to save to disk: this could cause a slowdown or stuttering of the system's rate of execution.

Saving an in-memory EMF runtime model to a file is an option for creating snapshots. Each file created for a snapshot would be a full copy of the model from memory. However, if a model does not change between snapshots, two or more snapshot files could contain identical data. This duplication

is an inefficient use of disk storage space that increases the runtime costs of a system.

The research objectives of this thesis do not include improving model snapshot techniques. However, the limitations of the current approach to taking snapshots of an in-memory EMF model need to be addressed. A solution to the snapshot problem could be found within existing MDE tools that enable model versioning. During the development of a model, it is desirable to keep several copies of a model at different stages of development. Each version of the model is likely very similar, as seen with the model snapshots. A versioning model repository could save disk storage space by only storing the changes between versions of a model.

Accessing the history model The initial development of Cronista has only briefly examined how a developer might access the history model. Plugins for the Eclipse IDE, like CDO Explorer, give a developer a user interface to access a model stored in a CDO model repository. These plugins are intended for developing models using CDO as a collaborative version-controlled repository. It is possible to use CDO Explorer to access/view the history model, but it offers no provenance-specific functionality like performing a provenance query. CDO Explorer might be more suitable for viewing model snapshots, which is in keeping with its intended uses. The next development cycle should explore CDO-compatible model query languages, which could enable provenance queries to be expressed.

Provenance graph visualisation CDO Explorer is limited to presenting models using on-screen forms, which makes it difficult to understand the provenance graph structures created by the Curator. Developing the Curator process involved writing Java code to access and manipulate a model in CDO. The same skills quickly transferred to creating a Java program that produced a Graphviz representation of provenance. Ideally, a model transformation tool would have been used, to create a more reusable solution for visualising provenance graphs. However, existing graph visualisation tools could potentially be used to explore a provenance graph. Enabling the interoperability of the history model store with existing graph database (and visualisation) technologies would also be an option.

The Graphviz approach to visualisation is limited to systems with a small model and a short execution: these systems produce small provenance graphs. In the early development stages, the Graphviz approach helped to identify issues with the provenance graph structures being created. There were instances of the versioning and cycling issues as discussed in Section 4.1.4, which were

easy to see on a small provenance graph drawn with Graphviz.

Graphviz drawings of provenance structures created by the Curator enabled discussion with peers. For example, a discussion with peers was needed regarding the *"wasDerivedFrom"* relationship. The Curator could apply this relationship between entities that were *"used"* and *"wasGeneratedBy"* an activity. However, this produced many relationships, with a very dense graph that peers found difficult to understand. There is also an element of ambiguity that can occur when an activity uses and generates multiple entities; there is an assumption all used entities contribute to all generated entities, which may not be the case. It was then that *"wasDerivedFrom"* was changed so it linked each entity to its previous version. Provenance of a model element can be traced through the *"wasDerivedFrom"* relationship, without inspecting the model element version details.

Manipulating the Java code that rendered the provenance graph with Graphviz enabled quick experimentation with alternative visualisations. Examples of simplified provenance graphs were shown in the evaluation section (Section 5.1.6 on page 147, and Figures 5.6 and 5.7). The two examples of simplification are achieved by filtering out graph nodes based on type (e.g. show only activity or entity). These alternative visualisations show the usefulness of an interactive visualisation tool that can selectively hide or show parts of a provenance graph on demand.

5.2 Applying Cronista to an MDE SAS (Traffic Manager)

5.2.1 Motivation

The demonstration in the previous section showed a provenance graph of the Fibonacci system being used to create statements about the system's execution. These statements about the execution would answer some simple provenance questions, whose answers are derived from one node or two related nodes. The Fibonacci graph was too simple to explore more complex provenance questions and answers, which require information that is spread across multiple nodes and relationships in a larger, more complex provenance graph.

Applying Cronista's provenance collection to a multi-agent system could create a sufficiently complex provenance graph in a history model to explore more complex questions. This multi-agent system would need to meet Cronista's requirements at this time, such as being built with EMF and having a shared-knowledge runtime model.

Cronista produces a history model that reflects the system's execution as interactions with its

shared-knowledge. Therefore, the design of the system's processes and shared-knowledge model affect the kinds of provenance questions that can be answered. To create a provenance graph with meaningful representations for experimenting with answering questions, it would be advantageous to have a good working knowledge of the target system.

5.2.2 Problem

A more complex system is required to test the implementation than the previous Fibonacci example, since the system used for testing must create a large provenance graph that involves multiple agents interacting via a shared-knowledge model. However, to evaluate Cronista's provenance representations for the system's execution, a good understanding of the system design is also required. Creating a system explicitly for this experiment would enable a good working knowledge of the system to be established, which would assist in validating that the provenance representation is correct.

The experiment's requirements imply developing a multithreaded (multi-agent) system with a shared-knowledge model based on EMF (Cronista's current tooling). However, EMF models are not natively thread-safe, and therefore an approach to creating a thread-safe system with EMF models is needed. Ideally, the approach to instrumentation for *MinimalEObjects* which has been developed and demonstrated could be reproduced if a similar base class is used for the multithreaded system.

The initial implementation simply snapshots an in-memory EMF model by saving a copy to disk. For a system with a large runtime model, this approach could introduce a slowdown of the system's execution. Furthermore, a large amount of disk space would be required to store the snapshots over a long execution. A large amount of disk space could be filled with duplicate data for unchanged sections of a runtime model. There is a need for more space- and time-efficient runtime model snapshots. To achieve this, EMF-compatible model repository tools might be used in place of the EMF in-memory model.

In addition to creating a more complex provenance graph, there will be a need to investigate more powerful approaches for exploring the provenance graph. To do this, there is a need to evaluate the use of an existing model query language for writing provenance queries. A developer would use the model query language to extract data from a provenance graph, in order to answer provenance questions about the system's execution.

5.2.3 Objective

Experimenting further with Cronista requires a multi-agent system that is well understood by a developer and that fits within Cronista's technical requirements. For these reasons, a 'Smart' Traffic Manager system will be built. The system will have a MAPE-K architecture, where multiple agents interact with a shared-knowledge model. Such a system could be developed using EMF and CDO, using the knowledge and skills developed in previous design and development cycles; the system would likely be compatible with Cronista's existing components.

To achieve thread-safety for agents accessing the shared-knowledge model, CDO could be used to detect conflicting changes and react accordingly. The CDO model repository is intended for collaborative modelling between humans. However, judging by the experience gained with creating the history model in the previous section, it should be possible to have system agents (non-human modellers) use CDO like a group of human modellers. Using CDO for a multi-agent system's runtime model also enables an investigation into the use of CDO's versioning system for creating model snapshots.

Using Cronista with a system model in CDO required an extension of the model instrumentation. Cronista's current model instrumentation is designed for use with EMF *MinimalEObject*s. CDO provides an extension to *MinimalEObject* called *CDOObjects*. Model Element access/correlation/versioning for *CDOObjects* will need to be investigated to create an appropriate set of reusable model instruments.

In the prior section (Section 5.1), CDO Explorer was considered for accessing snapshots: this was chosen to be evaluated in this experiment. Furthermore, MDE tools for querying models stored in a CDO model repository should also be evaluated with an experiment.

The objective motivates a design and development cycle that should answer the following local research questions.

LRQ1 *How can Cronista's model instrumentation be adapted for use with runtime models stored in an existing model repository?*

- **LRQ1.1** *How can an existing model repository model versioning be leveraged for Model Element versioning?*
- **LRQ1.2** *How can an existing model repository model version control system be used to create*

snapshots of a runtime model?

LRQ2 *How can we estimate the costs of using Cronista with a system?*

- **LRQ2.1** *How can we estimate the design-time developer effort for applying Cronista to a system?*
- **LRQ2.2** *What are the runtime resource overheads for using Cronista on a system?*

LRQ3 *How can a developer explore the history model to improve their understanding of a system?*

- **LRQ3.1** *How can a developer explore a provenance graph from a known point of interest, e.g. a failed state?*
- **LRQ3.2** *How can a developer write a provenance query to find answers to questions using the history model?*

The local research questions should contribute to answering RQ1.2 and RQ1.3:

RQ 1.2

To what extent can the reuse of the existing infrastructure for the shared-knowledge system also reduce the costs of providing provenance awareness?

RQ 1.3

For which common facets of provenance questions can the collected provenance data provide some information to support an answer?

5.2.4 Design and development

This design and development phase is divided into two sections. The first section covers the Traffic Manager system design, which will be used in an experiment to demonstrate provenance collection from a multi-agent SAS. The second section presents the design of new instrumentation for Cronista.

Self-adapting Traffic Manager

To experiment with Cronista's new components, an appropriate system that could be made provenance-aware was required. A self-adapting Traffic Manager was created using EMF to enable experimentation with Cronista. Figure 5.8 shows the Traffic Manager system, which controls traffic lights at different junctions in a traffic simulation program called SUMO [35, 73]. The Traffic Manager runs several 'smart' controllers that connect to the junction traffic lights in SUMO via an API called TraCI [36]. The Traffic Manager system has an explicit EMF runtime model for the system's shared-knowledge, which is suitable for applying provenance awareness.

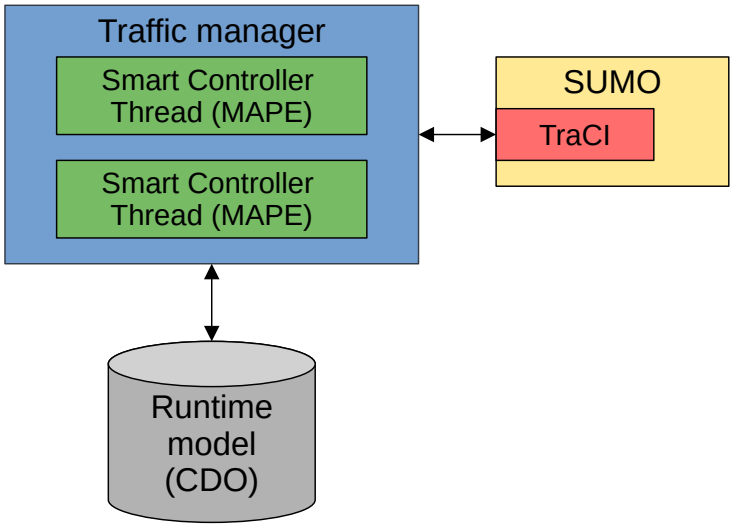

Figure 5.8: Overview of the Traffic Manager

A Traffic Manager runs several smart controllers in separate threads. Each smart controller executes a MAPE control loop (Figure 5.9), and stores its knowledge in a single shared-knowledge runtime model. A smart controller can adapt its behaviour using the runtime model in response to the traffic conditions it detects. The runtime model is stored in CDO, enabling thread-safe access to the model from multiple smart controllers; CDO provides concurrency control and versioning facilities.

The following sub-subsections provide more in-depth information on SUMO and the implementation of Traffic Manager.

Simulation of Urban Mobility (SUMO) A simulation of a pair of traffic junctions was implemented using SUMO (Simulation of Urban Mobility [35, 73]). SUMO is a simulator for road networks and traffic flows, which includes features to provide uncertainty, such as the Σ traffic flow parameters, which determine how erratically the cars are driven.

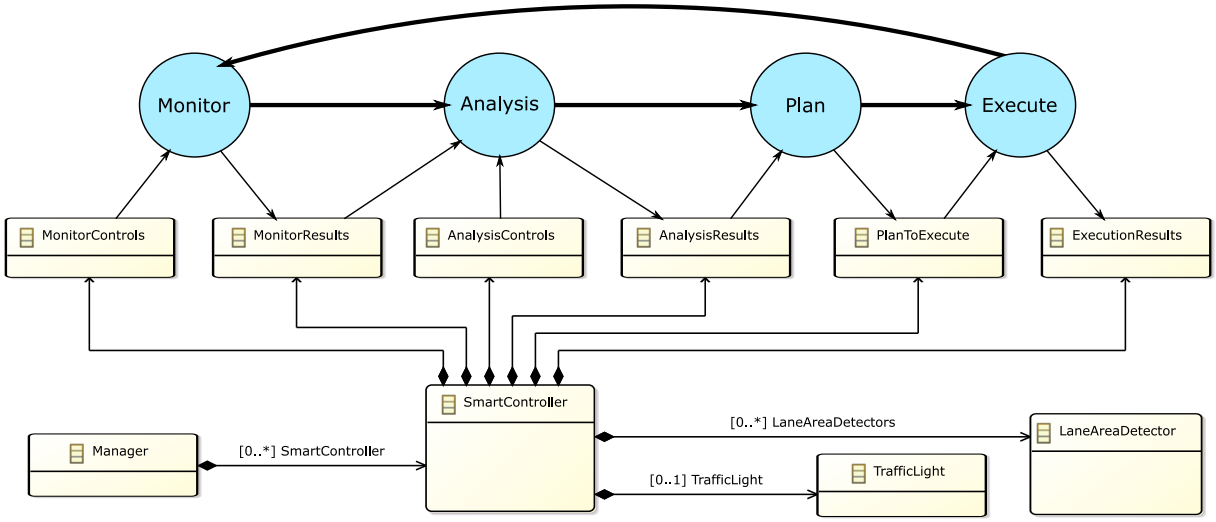

Figure 5.9: MAPE over Traffic Controller Model

Figure 5.10 shows a screenshot of the SUMO world simulation, which contains two traffic lights connected by a road. The Traffic Manager will be required to control the traffic lights on both junctions. The Traffic Manager accesses the SUMO simulation via the Traffic Control Interface (TraCI) [36]. TraCI enables objects in the simulation to be manipulated while a simulation is running.

Traffic Manager The Traffic Manager system extends the traffic simulation running in SUMO shown in Figure 5.10. Several traffic light smart controllers run concurrently, each in a separate Java thread, sharing a single TraCI connection to SUMO. The traffic lights in each junction follow a predefined cycle of timed *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:

- **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 need to copy that plan, to pass the traffic coming from the other junction and so prevent a queue). 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). Check 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

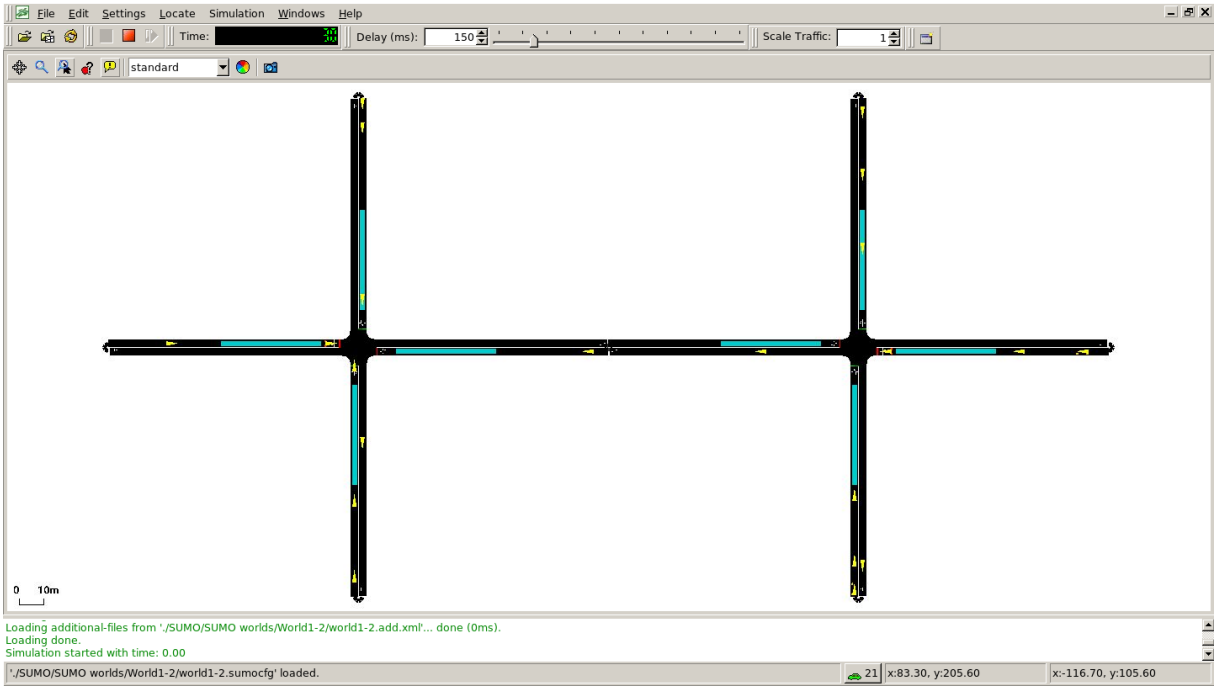

Figure 5.10: Screenshot of the SUMO-based traffic simulation

more than 2 are jammed), or decrementing J by one (if fewer 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 predefined duration.

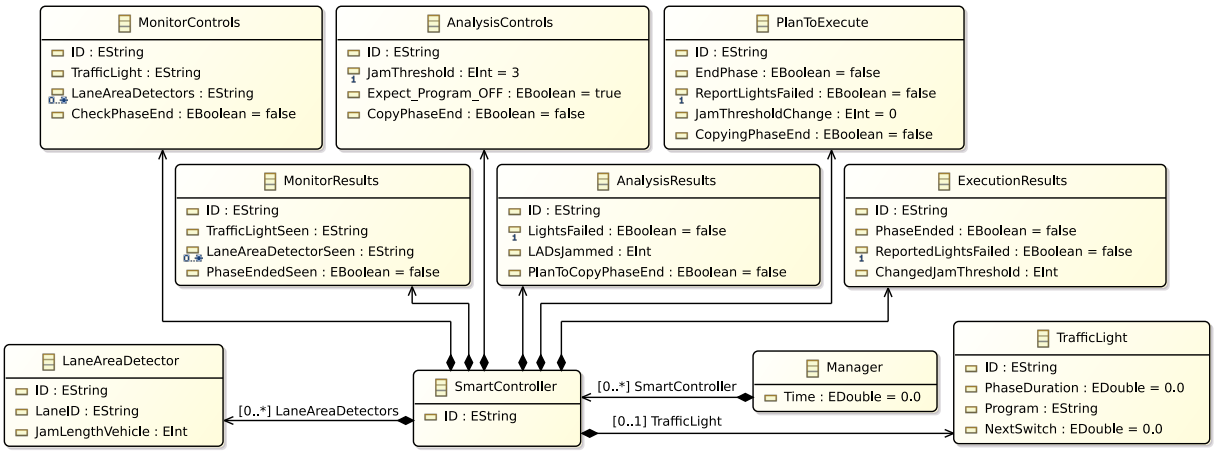

Figure 5.11: Class diagram for the metamodel of the system model

Figure 5.11 shows a class diagram for the system metamodel. The `Manager` program runs several concurrent `SmartController`s. 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 representing the SUMO objects managed by each controller; namely, the `TrafficLight`s and the `LaneAreaDetector`s.

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. As a target system for provenance awareness experiments with Cronista, it provides an example of 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.

Cronista's CDO model instrumentation

The Traffic Manager system is a multi-threaded Java application. Each thread is a Smart Controller (agent) that executes a MAPE loop (activities) to manage traffic signals. The knowledge (entities) component of MAPE is shared between all the Smart Controllers using a CDO model repository; the Smart Controllers perform "commits" to write changes to the shared-knowledge model.

A number of Cronista's original implementation components (Section 5.1) can be reused without modification. The Curator and CDO History Model Store are independent of the system being made provenance-aware and can be directly reused. Some of the Observer components remain the same: for example, the provenance message creation and queuing for the Curator. The agent and activity instrumentation do not require adaptation, because they are based on standard Java libraries.

The use of CDO for storing the runtime model in the Traffic Manager system requires new model instrumentation in the Observer. These model instruments will collect the same entity (model) information as the original EMF model instruments. However, these new instruments will be adapted to take advantage of CDO's *EObject* implementation, which includes model versioning.

In the design of the new model instrumentation for runtime models in CDO, a limitation in Cronista's versioning scheme was discovered. Models in CDO are handled differently from EMF in-memory models. These differences are with how Model Objects are handled; Model Objects are copied from

storage to memory, and several copies of the same Model Object can exist in memory. As a result of this discovery, there was a need to revise the implementation of the versioning components to accommodate the extra information. The extension to the versioning scheme is explained in the discussion of Model Element versioning later in this section.

As in the previous section, the model instrumentation design will be divided into three functions: Model Element access, correlation, and versioning.

Model element access The code generator for CDO model implementations can be set to create the same *eDynamicGet* and *eDynamicSet* methods, which can be extended similarly to the previous *LoggingMininalEObjectImpl*. However, the inspection code for extracting access information must be designed to specifically extract data from a *CDOObjectImpl* object.

Model element correlation Correlation of the Model Elements for a model based on *CDOObjectImpl* objects remained similar to that of *MinimalEObjectImpl*. All agents in the system retrieve Model Objects from a CDO repository using a CDO object identifier. This CDO object identifier is unique for every object in the model. Thus, two agents can access the same part of a model simply by using the same CDO object identifier.

Model Element versioning CDO provides versioning for the models it stores; when a CDO commit includes a change for a Model Object, that Model Object's version number is incremented in the CDO repository (if the commit is successful). However, this versioning system is applied at a Model Object level and not to Model Attributes. Therefore, CDO's versioning system is not of sufficient granularity for Cronista to represent versions of Model Elements as unique entities.

There is another problem with agents copying Model Objects from the CDO repository; this problem overlaps with Model Object correlation. Multiple agents in the system can access the same Model Object in CDO by using the same CDO ID in the access request. When agents access the same CDO Model Object concurrently, multiple instances (copies) of the Model Object are created in memory (one for each agent). When Cronista's Observers inspect each agent's accesses to Model Elements, the different Model Object instances all report the same CDO ID but their Model Attributes may report different states or values.

Cronista's initial implementation versioning system only considered Model Element versions for a

single instance of a Model Element. This was based on an assumption that a multi-threaded system would provide a single instance of a thread-safe wrapped model. Thus, multiple agents would access a single instance of each Model Element, which could have multiple versions over time as its state changed. For this experiment, however, Cronista needed to provide another versioning system that could track multiple copies of Model Elements and their state changes.

A new versioning component was created for systems that store their runtime models in CDO. This new versioning component provides a thread-local versioning system for Model Elements. Cronista maintains a (v_s, v_m) pair for each Model Element within a thread's memory space: the storage version v_s is controlled by CDO, and the in-memory version v_m is set to 0 upon a CDO commit and incremented each time the entity is modified between CDO commits.

The Observers must create or reuse a unique entity ID when reporting an agent's Model Element access (Section 4.1.4). Part of the entity ID is derived using the Model Element version: this would now be a combination of the storage and in-memory versions from the thread-local versioning system. However, there is the potential for conflicting entity IDs to be created by Observers: several agents could access the same Model Element and produce identical sequences of storage and in-memory versions. To prevent entity ID collisions, an Observer should qualify the Model Element versions with a triple: thread (agent), storage version, and in-memory version.

Figure 5.12 shows a provenance graph of two agent threads copying a Model Element from storage into their memory space. The yellow ovals representing ENTITIES are labelled with the entity ID triple the Observer would produce, however, the Model Element version information is encoded in the provenance graph with the *"wasDerivedFrom"* relationship between two entities. Each thread creates multiple in-memory versions, with the agent's Observer tracking and reporting the accesses. Finally, each thread returns its in-memory copies to storage (here, Agent2 has ignored the conflict warning) and creates new storage versions.

Model snapshots The versioning in CDO can be exploited by Cronista for creating model snapshots. A CDO commit operation involves pushing a collection of changed Model Objects to the CDO repository in a transaction. A commit transaction is accepted by the CDO repository (if there is no conflict), and the changes are applied to the stored model. A commit reference is then sent from the CDO repository to the committer, which can be used at any time to revisit that version of the model. Since Cronista's instrumentation for tracking Model Element versions already required detecting CDO

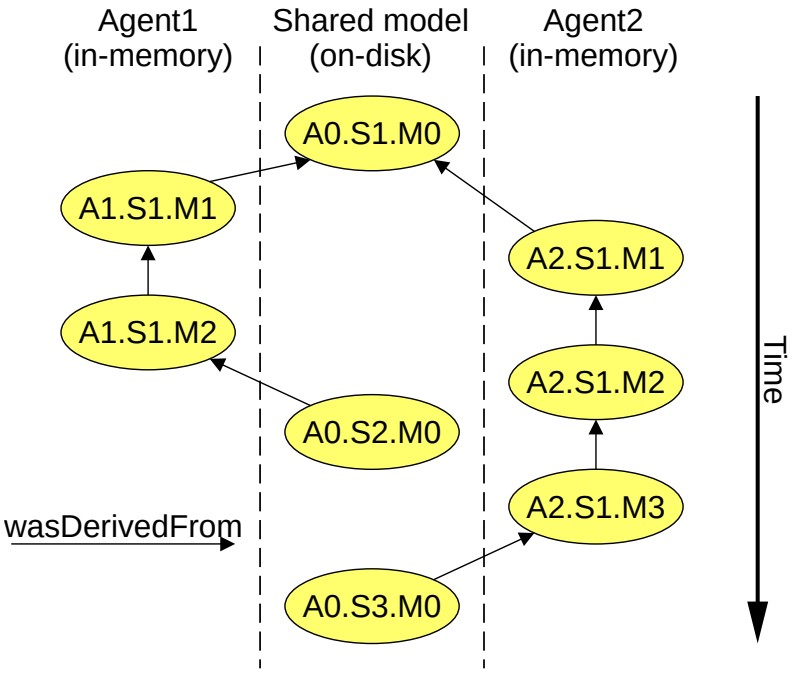

Figure 5.12: Provenance of a single Model Element being accessed by two agents in parallel

commits, this was extended further to capture the references produced by the commits.

For simplicity, Cronista's Observers report the commit references to the Curator, who stores the reference as the time window's snapshot. The time window no longer needs to contain a copy of the model: the model snapshot remains in the system's CDO repository as a model version. This approach avoids the delays in creating a full model copy, and takes advantage of CDO's efficient change-based model storage.

5.2.5 Demonstration

This section will present code and implementation examples for Cronista's CDO provenance instrumentation. Following these examples, an experiment with the Traffic Manager will be presented.

Code and implementation examples

Cronista provides model instrumentation for CDO in *LoggingCDOObjectImpl*, which replaces the *CDOObjectImpl* base class normally used in CDO models. While creating this second set of model instruments, some functionality in the original *MinimalEObject* was duplicated in the *LoggingCDOObjectImpl*; e.g. functionality for processing model accesses after details (like IDs) are extracted from the model implementation.

The functionality for processing model accesses was separated into a *ModelAccessBuilder* component, which takes in *Model Objects and Attributes* (that conform to an interface) from the model the model-specific instrumentation. Developing these components separates Cronista's model access processing from the model (instrument) implementation details: the interfaces act as generalised abstract representations of *Model Objects and Attributes*.

Model Element access The provenance recording process of a system agent performing model accesses needed to be generalised beyond the original implementation. Listing 5.10 shows two new interfaces: *ModelObject* (line 1) and *ModelAttribute* (line 12). These interfaces specify the Model Object and Model Attribute information a developer creating model instrumentation must provide from their specific model implementation.

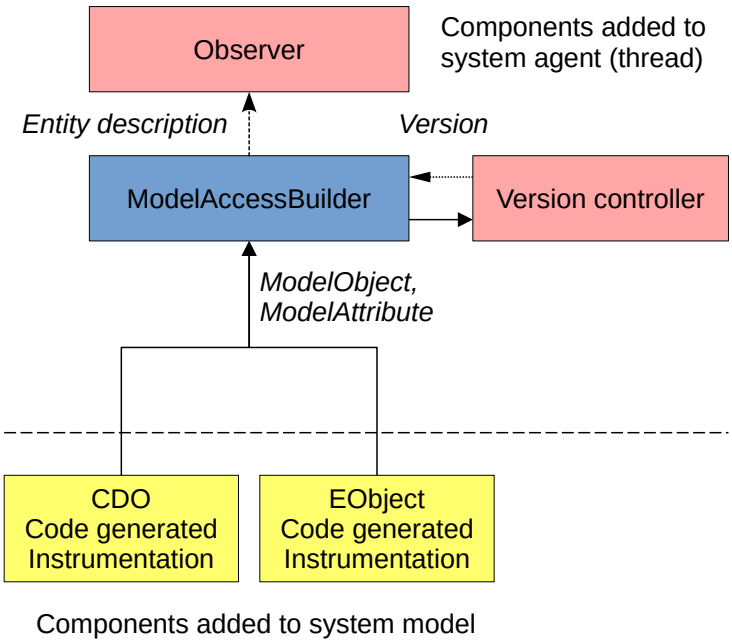

Figure 5.13: A ModelAccessBuilder receives generalised (Model Object, Model Attribute) pairs detected by the CDO/EObject code-generated instrumentation.

The developer's implementations of *ModelObject* and *ModelAttribute* can be passed to Cronista's *ModelAccessBuilder* (Listing 5.10, line 20). The *ModelAccessBuilder* creates the relevant provenance entity representations for sending provenance messages to the Curator via the Observer (Figure 5.13).

The instrumentation for CDO is applied to the eDynamicGet and eDynamicSet methods (Listing 5.10, lines 31 and 37). In these methods, instances of *ModelObject* and *ModelAttribute* are created and passed to the *ModelAccessBuilder*, together with an enumeration value indicating the type of access being performed.

```
1  public interface ModelObject {
2      String getIdentifier();
3      String getName();
4      String getType();
5      Object getValue();
6      String getValueAsString();
7      Object getRaw();
8      boolean hasStorageVersioning();
9      int getStorageVersion();
10 }
11
12 public interface ModelAttribute {
13     String getIdentifier();
14     String getType();
15     Object getRaw();
16     boolean hasStorageVersioning();
17     int getStorageVersion();
18 }
19
20 public class ModelAccessBuilder {
21     public void reportModelAccess(ModelObject modelObject,
22         ModelAttribute modelAttribute, ObjectAccessMethod objectAccessMethod)
23
24         switch (objectAccessMethod) {
25             case GET: /* Entity */; break;
26             case BEFORE_SET: /* Old Entity */ ; break;
27             case AFTER_SET: /* New Entity */ ; break;
28             case COMMIT: [ ... ]; break;
29 } } }
30
31 protected Object eDynamicGet(int dynamicFeatureID, EStructuralFeature eFeature,
32             boolean resolve, boolean coreType) {
33     // ... Single inspection lock ...
34     ModelAccessBuilder.reportModelAccess(modelObject,modelAttribute, GET);
35 }
36
37 protected void eDynamicSet(int dynamicFeatureID, EStructuralFeature eFeature,
38             Object newValue) {
39     // ... Single inspection lock ...
40     ModelAccessBuilder.reportModelAccess(modelObject,modelAttribute, BEFORE_SET);
41     // ... Perform super set() ...
42     ModelAccessBuilder.reportModelAccess(modelObject,modelAttribute, AFTER_SET);
43 }
```

Listing 5.10: Model element access instrumentation for CDO Objects (*LoggingCDOObjectImpl)*

Model Element correlation The Observer now concatenates the CDO Object ID, Feature ID, Agent ID (from the thread), storage version and memory version to produce a unique entity ID. The concatenation of the IDs ensures a unique ID is produced, as the IDs of the source are unique in their scopes:

- **CDO Object ID** makes the object unique from all others in the model.
- **Feature ID** makes an attribute unique from all others on the same object.
- **Agent ID** is unique to the Java thread (memory space) containing an Observer and version controller.
- **Storage version** is unique to the state of a Model Element in a shared storage system.
- **Memory version** is unique to the state of a Model Element in memory.

Model Element versioning Cronista's initial implementation (Section 5.1.5) included a simple version controller, which could be used to version in-memory EMF models that lacked native versioning. CDO provides Model Object versioning that increments a counter when a commit changes a Model Object. However, this version counter is at an object level and does not track the versions of attributes, which may or may not be changed when a Model Object is committed. Thus, Cronista needs to provide a version controller implementation that is aware of the distinction between the versions committed to CDO and those only available in memory within a particular thread.

With the introduction of the interfaces for *ModelObject* and *ModelAttribute*, Cronista's version controller can be adapted to track and version models using the generalised model representation. The model instruments report the model access to the Observer (in the generalised form); the Observer passes the *ModelObject/Attribute* to its own version controller instance for versioning operations: depending on the type of model access the versioning operation either recalls (read access) or increments (write access) a version number.

The version controller creates a hashmap of keys, with each key formed as a combination of *ModelObject/Attribute* IDs (see Model element correlation in Section 5.1.4). With each key the version controller keeps an object representing the Model Element's version: either an in-memory version number or an in-memory and storage version number pair. The generalised Model Object/Attribute

representation includes a flag to identify if storage versioning is available (and a storage version number).

The version controller can determine which instance (i.e. copy) of a Model Object it is versioning based on the presence of some or no version information in the Model Object (Table 5.1). For example, if an agent accesses a Model Object for the first time, the object is loaded from storage (CDO) into memory, and the version controller has no entry for the Model Object. Upon loading the Model Object to memory, the version controller creates an entry with the storage version set to the Model Object's CDO version number and the memory version to 0. As the Model Attributes on the Model Object are updated, their memory versions are incremented. Finally, the agent commits the object back to CDO: when successful, CDO increments the revision number of the object, and Cronista's version controller discards all entries relating to Model Objects in a commit (described below in the Model snapshots section).

		Storage version (S)	
		S=0	S>1
Memory version (M)	M=0	Created in memory	Loaded from storage
	M>1	In memory only	In memory and storage

Table 5.1: Version conditions for types of CDO model accesses

The version controller is coupled to the Observer's *ModelAccessBuilder*, which triggers the necessary versioning processes based on detected model accesses. Currently, the version controller assumes there is always a need to track in-memory versions; storage versions are used if model instrumentation indicates it is present, or else storage versions are set to 0 to signify no storage versions exist. Further improvements could be made by having the version controller adapt to the presence of in-memory versioning information in the *ModelObject* and *ModelAttribute*. If a system or framework has an in-memory version controller, the developer would map version information from a system's internal runtime model and the version controller would relay this information to the *ModelAccessBuilder*.

Model snapshots Model snapshots for a runtime model stored in CDO can be achieved by tracking commits. A system can transparently load CDO objects and manipulate a copy of the CDO object in memory. However, for the new state to be persisted in CDO, a commit is required. During the commit process, the CDO objects are checked for conflicts[3], which need resolving before CDO can replace

[3]A conflict is when contradictory changes are being made to a model, i.e. the current commit involves part of the model that changed since the commits base-copy of the model was taken.

the CDO object in storage. Successful commits return a reference that can be used to access a view of the model at that moment; this reference can be stored in the history model as the snapshot.

Detecting a system performing a commit is similar to detecting a model access: a commit method is extended with instrumentation for collecting provenance and notifying the version controller. For the initial implementation of this method, a "commit" method has been added to the Observer, which wraps a CDO commit method. A developer needs to manually replace the CDO commit method calls in the system with the Observer's commit method.

The Observer performs the CDO commit on behalf of the system, and if successful a commit reference is passed back to the system. For each Model Element included in the commit transaction, the *ModelAccessBuilder* is used to send a series of commit provenance messages, containing model elements (new and old), an activity description of "commit", and the agent performing the commit. The 'commit' activity node is created with related old and new entities: these connect an agent's in-memory version of the model elements to the new versions in storage.

A Curator does not need to record a snapshot and start a new time window for every commit. The system may frequently perform commits, which would result in very short provenance graphs and snapshots with very few changes, potentially increasing the storage costs for provenance. The Curator only needs to align its process for creating a snapshot and new time windows with the arrival of the commit message for a commit transaction. For this experiment, a new time window is periodically started, based on a duration set by the developer.

Traffic Manager experiment

Cronista was applied to the Traffic Manager system discussed in Section 5.2.4. SUMO traffic simulations were run with both a Traffic Manager monitored by Cronista, and an unmodified Traffic Manager. The resource usage metrics were collected and compared. The Cronista-monitored simulations were run with and without an injected fault, to produce different provenance traces and see if the fault could be identified from the collected data. Finally, Cronista used model query languages to evaluate the completeness and accuracy of the collected provenance information, by finding the injected fault.

Design-time cost observations In terms of developer effort, the integration of Cronista required adding 16 activity scopes in two levels. First, an activity scope defines each of the MAPE phases.

Next, each 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 the junction controller code in 16 try-with-resources blocks, as in Listing 5.11. Commits to CDO were modified so they would go through the `ThreadObserver` for the current agent: this was 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.

The traffic management system’s code base grew after these changes from 823 lines of Java code to 846, as measured by `sloccount` 2.26 [113] while ignoring generated code. This represents less than a 3% increase in code. The reusable Observer and Curator components are independent of this codebase, measuring 943 lines (ignoring as well code automatically generated by EMF).

Runtime cost observations To compare time and memory demands, the simulation uses a background thread to record total heap memory usage with the memory pool management beans present in Java (`MemoryPoolMXBean`), once per second and once at the end of the execution. The simulation was run 10 times over 200 ticks with and without the reusable provenance layer. These executions were done on an Ubuntu 20.04 system with a Linux 5.4.0-42-generic kernel, using Oracle JDK 11.0.6, SUMO 1.4.0, and CDO 4.10 (as included in Eclipse 2020-06). 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. Java was run with an initial heap size of 256MiB, and a maximum heap size of 512MiB.

To measure changes in disk usage, the instrumented version of the simulation was run, and the sizes of the Derby databases used by CDO to store models were measured. The system model database took 864KiB and the history model database took 16MB. This shows that a history model can be much larger than its system model, and therefore, it justifies the need for pruning old history to manage storage demands.

The experiment collected the following metrics (shown in Table 5.2) for the runtime costs of applying Cronista to the Traffic Manager system. Standard deviations (SD) for time and memory show the degree of consistency in the results collected over the 10 runs, which gives some confidence in the mean values. Applying Cronista did not increase the time taken by the simulation: this indicates

```
1  public void run() {
2
3  while (!done) {
4      startTime = System.currentTimeMillis();
5      try (ActivityScope scope0 = new ActivityScope("Monitor")) {
6          try (ActivityScope scope1 = new ActivityScope("monitorTrafficLight")) {
7              // Read traffic light data from SUMO }
8          try (ActivityScope scope1 = new ActivityScope("monitorLaneAreaDetector")) {
9              // Read Land Area Detectors data from SUMO }
10         try (ActivityScope scope1 = new ActivityScope("monitorOtherControllerPhaseEnd")) {
11             // Read shared−knowledge model for other junction controller phase end signals }
12         commit(transaction); // Update shared−knowledge model (contain activity scope commit)
13     }
14     try (ActivityScope scope0 = new ActivityScope("Analysis")) {
15         try (ActivityScope scope1 = new ActivityScope("analysisTrafficLightFailureOUTER")) {
16             // Test traffic lights }
17         try (ActivityScope scope1 = new ActivityScope("analysisLADJammed")) {
18             // Check LADs for "Jams" }
19         try (ActivityScope scope1 = new ActivityScope("analysisPhaseEndEvent")) {
20             // Did another junction end phase early? }
21         commit(transaction);
22     }
23     try (ActivityScope scope0 = new ActivityScope("Plan")) {
24         try (ActivityScope scope1 = new ActivityScope("planReportLightsFailed")) {
25             // ReportLightsFailed }
26         try (ActivityScope scope1 = new ActivityScope("planJamThresholdChange")) {
27             // Change the Jam threshold? }
28         try (ActivityScope scope1 = new ActivityScope("planEndPhase")) {
29             // Attempt to clear a Jam? (EndPhase) }
30         commit(transaction);
31     }
32     try (ActivityScope scope0 = new ActivityScope("Execute")) {
33         try (ActivityScope scope1 = new ActivityScope("executeReportLightsFailed")) {
34             // Report lights failed }
35         try (ActivityScope scope1 = new ActivityScope("executeJamThresholdChange")) {
36             // Change the Jam Threshold }
37         try (ActivityScope scope1 = new ActivityScope("executeEndPhase")) {
38             // Clear a Jam by ending a phase }
39         commit(transaction);
40     }
41     endTime = System.currentTimeMillis();
42     snooze(DECISION_FREQUENCY_INTERVAL);
43 }
44 // EXIT
45 }
```

Listing 5.11: Junction controller run method code with activity scopes applied

Metric	Provenance?	Mean	SD
Time	No	32.34s	0.26s
	Yes	32.34s	0.30s
Memory	No	8.95MiB	0.04MiB
	Yes	18.50MiB	0.21MiB

Table 5.2: Means and standard deviations of execution times and maximum memory usage, over 10 simulations across 200 ticks without/with the provenance layer.

the increased CPU load has not caused system execution to slow down.

The peak memory consumption has increased by approximately 10MB, just over double the system's original memory footprint. Given the original system has a small footprint, the additional components Cronista added could account for a significant portion of the increase. During execution, Cronista's Observer and version controller maintain data structures whose size varies depending on the volume of provenance information being handled. A system performing many activities quickly, with many model interactions, would create a significant number of provenance records that increased the memory consumed by Cronista.

Provenance query experiment In the previous Fibonacci demonstration (Section 5.1.5), graph visualisations were used to explore the collected provenance. In this demonstration, the use of a query language to express provenance queries was explored. With the history model stored in CDO with its default Derby backend, SQL could have been used to query the Derby database. However, these queries would not translate to other CDO backends. Instead, the query was written with an existing model query language that supports CDO: the Epsilon Object Language (EOL) [67].

To motivate the need for writing a provenance query, a hypothetical situation involving the Traffic Manager system was created. While the situation was imaginary, it served as an example of how the provenance graph might be used to identify the root cause of a defect. For this demonstration, it was also necessary to deliberately seed a fault in the junction controller code. The root cause of this fault needed to be found via a query. The fault could then be repaired, and further checks made to confirm the fault had been fixed. Correcting the fault should change the system's execution, which would be reflected in the provenance as changes to the graph in the area where the query exposed the fault.

The situation was as follows. Suppose a new developer starts at a company 'Smart Traffic Inc.'. On the developer's first day, they are asked to investigate an issue with a recent update to a smart junction controller. The report says that the traffic lights are changing too often, and enclosed with the

```
1   // 1. When was PhaseEnded set to true?
2   var ePhaseEnded = Entity.all.select(ed |
3     ed.attr() == 'PhaseEnded' and ed.bool() and ed.isWrite());
4   for (entJT in ePhaseEnded) {
5     entJT.printLayer(0);
6     // 2. Where did the EndPhase this activity used come from?
7     for (entUsed1 in entJT.wasGeneratedBy()?.Used
8         ?.selectOne(e | e.attr() == 'EndPhase')) {
9       entUsed1.printLayer(1);
10      // 3. What information was used when EndPhase was set?
11      for (entUsed2 in entUsed1.wasGeneratedBy()?.Used) {
12        entUsed2?.printLayer(2);
13      }
14      '\n=end='.println();
15  }}
```

Listing 5.12: Excerpt of query Q1: find information used when a phase was ended

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

Listing 5.13: Example of output of Q1 with faulty system

report is Figure 5.11 (the metamodel of the system's shared-knowledge model), a brief description of the junctions' MAPE feedback loop, and a history model from the faulty system including the execution provenance graph.

The developer would start by writing a query (e.g. in EOL) to find the activity that caused the phase change, by referencing the metamodel in Figure 5.11. Such a query would look like the one in Listing 5.12, although it would typically be built up incrementally (to save space, we only include this final version): i) look for the cases when `PhaseEnded` was set, ii) look for what activity informed those cases, and finally iii) focus on `EndPhase` and then look for what informed its value at the time.

To determine the reason for the ACTIVITY generating a '`PhaseEnded` set' ENTITY, 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 5.12: 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 5.13. 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

```
1     // activity scope
2     try (var s = new ActivityScope("analysisLADJammed")) {
3       analysisLADs(sc.getLaneAreaDetectors(),
4       sc.getMonitorResults(), sc.getAnalysisControls(),
5       sc.getAnalysisResults());
6     }
7     // method being called
8     public void analysisLADs(...) {
9       //sysAR.setLADsJammed(0); // ← FAULT
10      for (LaneAreaDetector sysLAD : sysLADs) {
11        if (sysMR.getLADSeen().contains(sysLAD.getSumoID())) {
12          if (sysLAD.getJamLength() > sysAC.getJamThreshold()) {
13            sysAR.setLADsJammed(sysAR.getLADsJammed() + 1);
14    }}}}
```

Listing 5.14: Excerpts of seeded fault located by Q1

```
1  L0: PhaseEnded true > GenBy > executeEndPhase
2    L1: EndPhase true > GenBy > planEndPhase
3      ...
4      L2: LADsJammed 3 > GenBy > analysisLADJammed
5  =end=
6  L0: PhaseEnded true > GenBy > executeEndPhase
7    L1: EndPhase true > GenBy > planEndPhase
8      ...
9      L2: LADsJammed 2 > GenBy > analysisLADJammed
10     L2: PlanToCopyPhaseEnd true > GenBy > analysisPhaseEndEvent
11 =end=
```

Listing 5.15: Examples of output of Q1 with fixed system

last part would raise concerns, as `LADsJammed` should never be larger than 4, the number of LADs at each junction.

The developer would then know that the problem was in the `analysisLADJammed` activity. At this point, the developer would look at the code (see Listing 5.14) and inspect the code in the activity scope. Within the activity scope, the developer would find that someone inadvertently commented out an important line which resets the `LADsJammed` counter before recalculation.

After fixing the fault, the developer could let the system run again, before re-running the same query and checking that the phases are ending for the correct reasons. A working system would produce an output such as that shown in Listing 5.15, showing 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.

Provenance visualisations Section 5.1.5 mentioned the use of CDO Explorer to visualise models stored in CDO. Figure 5.14 is a screenshot of CDO Explorer being used in the Eclipse IDE to view a history model's provenance graph. CDO Explorer provides a tree structure for navigating the model (left-hand panel in Figure 5.14); in this panel, the provenance graph nodes can be seen as the children of the time window's provenance graph node. Model node properties can be seen in the bottom panel of Figure 5.14; the selected ACTIVITY properties reveal the *"used"* ENTITIES. The view that CDO Explorer provides is limited to nodes that are connected directly: the user must traverse an edge to see nodes that are connected by more than one edge. The limited view of connected nodes is one reason why CDO Explorer is more appropriate for viewing snapshots than a provenance graph.

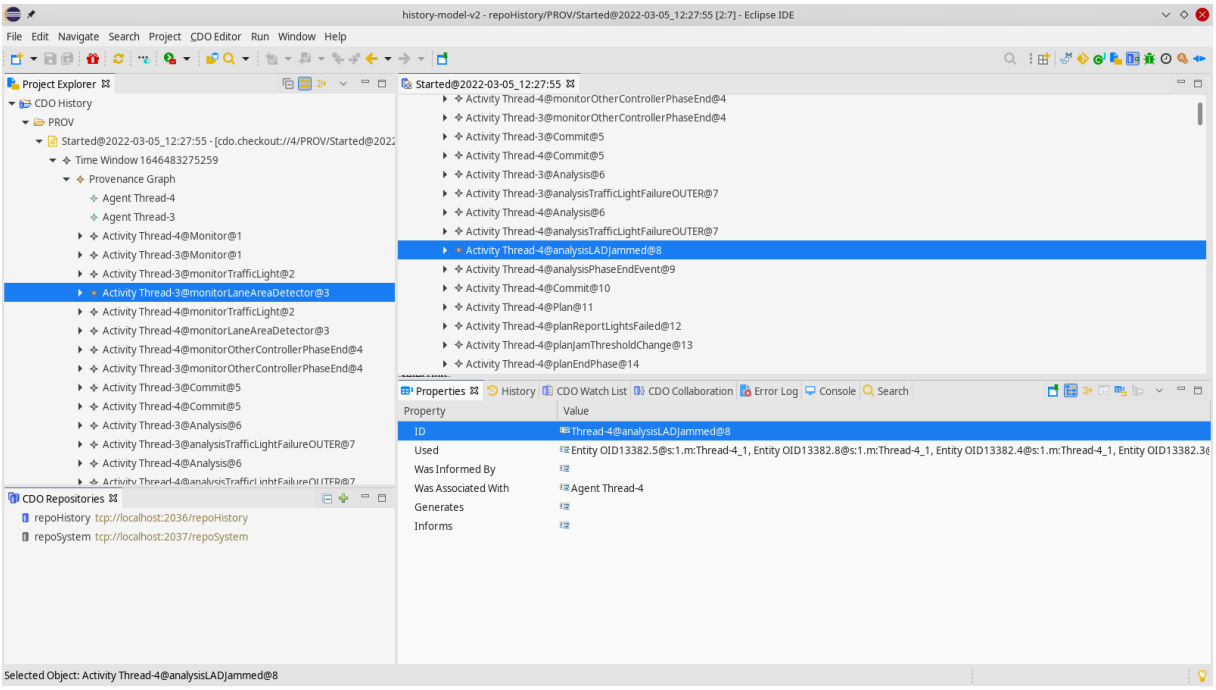

Figure 5.14: Screenshot of CDO Explorer being used to explore a history model

The previous Graphviz-based methods for provenance visualisation were used to produce some provenance graphs from the history model captured during the Traffic Manager experiment. Once again, the produced graphs required some editing by hand to make them easier to read.

Using an ENTITY-only view of the provenance graph, it is possible to search for the expected system behaviour for changing the *JamThresholdChange*. In the first few time windows, the provenance graphs show the relaxing of the jam threshold, which is the expected behaviour for an empty junction. From the entity-only graph in Figure 5.15, this is seen as a *"wasDerivedFrom"* connection between

an earlier and later entity for the jam threshold.

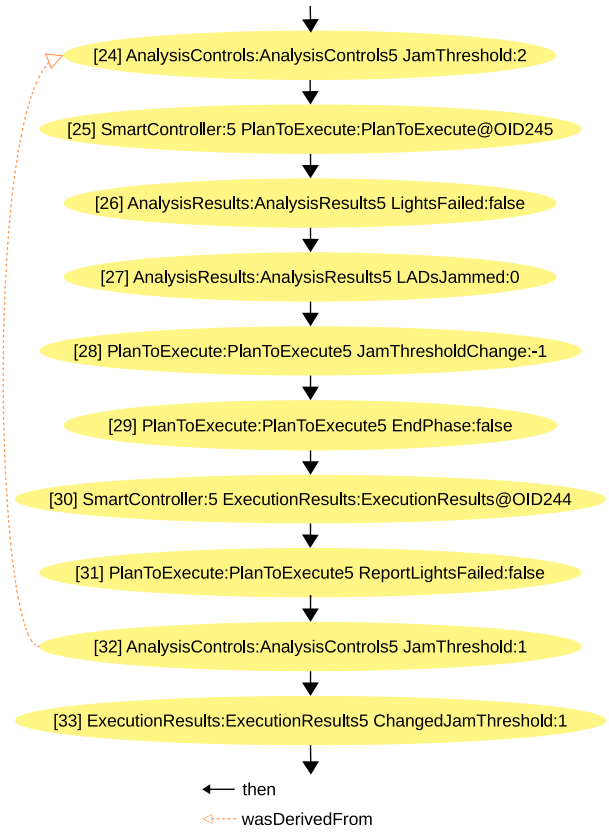

Figure 5.15: Entity-only provenance for Jam Threshold being lowered

In a later time window, the provenance graph pattern shown in Figure 5.16 can be found. The ENTITY only view shows a single entity representing the jam threshold as the value is unchanged, having reached the minimum value (1). However, *[28] PlanToExecute* (Figure 5.16) is showing a further attempt to reduce the jam threshold. In this instance, the condition is guarded against so as not to cause a problem.

The provenance graphs in each time window of the Traffic Manager's history model are too complex to present in their entirety. A part of the provenance graph is shown in Figure 5.17, which was found by performing a search against the graphs for the entity representing the `LADsJammed` counter. It was possible to capture the moment the counter passed the threshold of 4 in a visual representation (Figure 5.17, ENTITY nodes [25], [26] and ACTIVITY [8]).

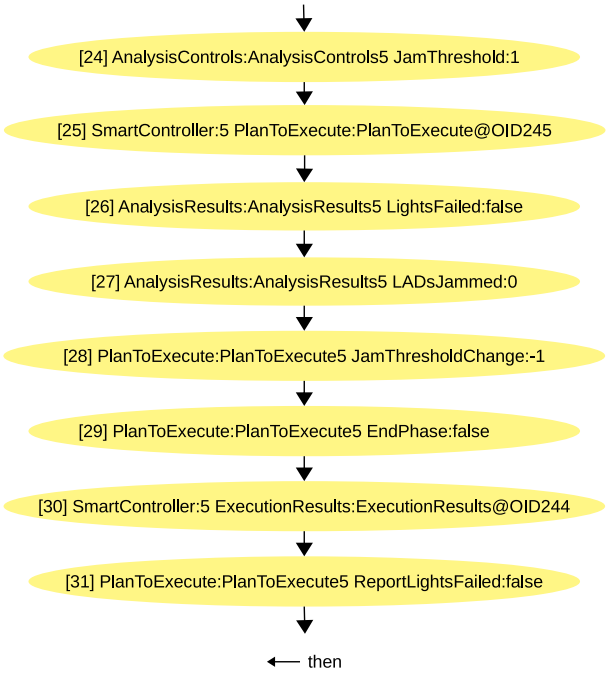

Figure 5.16: As Figure 5.15, but Jam Threshold now at minimum level

5.2.6 Evaluation

In this evaluation section, the research questions raised for this development cycle will be answered. Followed by an examination of the threats to validity for Cronista and the demonstration. The section closes with a discussion that leads to the next development cycle.

RQ1 and sub-question answers

The answers to RQ1 and sub-questions are largely around identifying the separation of concerns and leveraging a common abstraction layer that developers extend or adapt to a specific system. The components a developer would need to implement for Cronista to be used with a different technology should be minimal (e.g. writing adapters to provide the required information about Model Objects and entities), with Cronista providing the more complicated functionality required to record provenance representations.

LRQ1

How can Cronista's model instrumentation be adapted for use with runtime models stored in an existing model repository?

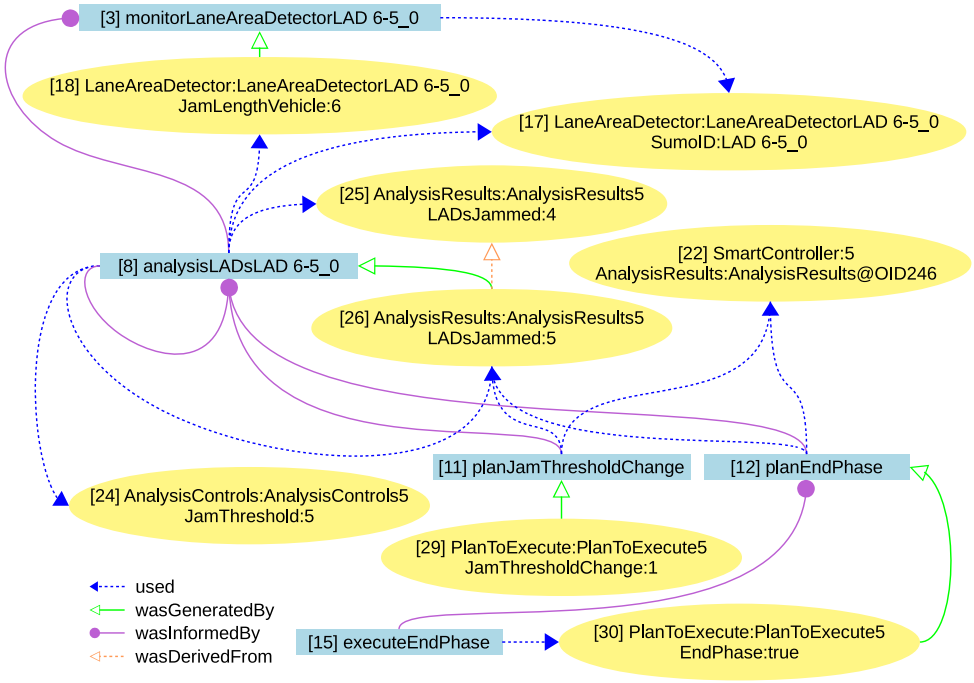

Figure 5.17: Cropped graph for runaway condition

LRQ1 answer Two Java interfaces have been introduced to guide developers in creating new model instrumentations for Cronista. These interfaces specify the requirements on Model Object and Attribute information from Cronista. This approach enables different MDE technologies to be mapped by writing adapters to a common Model Element representation that Cronista can interpret to create ENTITY representations.

LRQ1.1

How can an existing model repository model versioning be leveraged for Model Element versioning?

LRQ1.1 answer The intermediate Model Object and Attribute representations decouple the Cronista version controller from the specifics of the technology used to store the models. The new version of Cronista includes implementations of these intermediate representations for in-memory models, and models stored in a CDO model repository. Future work could extend the set of implementations to accommodate other versioning systems.

LRQ1.2

How can an existing model repository model version control system be used to create snapshots of a runtime model?

LRQ1.2 answer Cronista has been made aware of CDO's commit operation, which is used to push changes to the model store. During this operation, the CDO repository returns a reference that can be used to recall a view of the model produced by the commit. Cronista's Curator sends messages to update the provenance graph, representing the changes made by the commit. Given the approach to distributing provenance across time windows, the Curator can freely create a new time window with a snapshot when a commit occurs.

Since the snapshot can be retrieved from the system's CDO repository, only a reference to the commit needs to be stored in the history model, saving time and space since a full model copy is no longer needed.

RQ2 and sub-question answers

The second set of research questions sought knowledge of the cost implications of the use of Cronista. Both design and run-time costs would vary between systems and development teams. Experiments have been performed to collect some metric information, which gives a measure of the overheads for the case study system.

LRQ2

How can we estimate the costs of using Cronista with a system?

LRQ2 answer Applying provenance awareness to a system involves adding code or components to a system, and in some cases may also require design changes to the system itself. The scale and scope of the changes to a system to apply provenance awareness will increase the cost of the system at design- and run-time. There is a need to understand how these costs could be estimated. The following two sub-research questions contribute answers to this broader question of cost estimation.

LRQ2.1

How can we estimate the design-time developer effort for applying Cronista to a system?

LRQ2.1 answer Applying Cronista's model instrumentation to a system's model is mostly automated. A developer configures the EMF code generator with the appropriate instrumentation class and generates the model code. The size and complexity of the model do not appear to affect the manual developer effort involved.

The most significant developer effort is seen in the application of the activity scopes. Inserting the activity scopes requires a developer to wrap code regions into blocks, to denote the start/end of activities. The effort seen in the example does not demonstrate the effort a developer may also need to apply to understand the system's code to be able to define these activities.

Automation of common or repetitive tasks can reduce the cost of applying provenance awareness. Cronista demonstrates that some of the design-time activities can be automated using existing technologies: this removes some of the developer time and effort overhead for applying provenance collection. However, there are some manual activities a developer needs to perform during the design-time processes. Some of these activities are a one-off effort from a developer: for example, creating a model instrument for a new model implementation. This type of effort results in a component that could be reused across multiple systems; the more this component is reused, the greater the return on investment from the initial effort.

Some developer efforts are harder to predict, as they will vary depending on the system to which provenance-awareness is being applied. An example of these costs is seen in the need to identify and apply activity scopes in the system. This process could become iterative, where the developer needs to refine the scope of activities to improve the provenance descriptions of the system's execution. The SLOC counts in the experiment results indicate the number of lines of code needed for the Traffic Manager system. Of the added lines of code, some are required to attach Cronista to the system, and then 1 line of code is needed for each activity that should be recorded in provenance.

Cronista offers some reusable components, which can handle some of the general implementation requirements for enabling provenance awareness. For example, the CDO history model implementation is reusable between Java-based systems, as it has almost no dependencies on the technologies used by the system: the Observer and provenance message separate the concerns. Reusable

components for common functionality reduce the design time costs for implementing provenance awareness.

LRQ2.2

What are the runtime resource overheads for using Cronista on a system?

LRQ2.2 answer The implementation of Cronista has not been fully optimised, but the implementation was created to demonstrate that the addition of provenance-awareness was possible with negligible overhead.

The runtime resource costs for provenance collection can be divided into separate groups. Some resources are external to a system, e.g. the Curator and History Model Store: these external resources could use separate hardware from the system. Then there are resources internal to the system, e.g. the Observer and instrumentation: these resources are seen as the added overhead on the system, and probably cannot be delegated to external resources for processing.

The amount of external resources for the Curator and history model will depend on the provenance requirements: for example, the provenance retention period required. Systems that process a large amount of data or perform many activities for provenance recording will produce more data.

Internal resources will increase with the amount of provenance data being collected, and proportionately to the number of Observers and instruments needing to be applied. These costs will increase or decrease with the granularity of the provenance recording. For example, more fine-grained provenance of activities would describe smaller steps in a system execution, and more provenance messages being sent to the Curator. Thus, more instrumentation and Observer processing occurs around the system's code, slowing the speed of execution and consuming memory.

Aside from optimisation of the Observer and instrumentation to reduce runtime costs, a system designer needs to strike a balance for the granularity of provenance recording. Using a provenance requirements methodology like PLEAD is one way a designer might do this. In cases where provenance collection is needed to provide a broad coverage of the system, the costs can be managed. For example, Cronista's use of multi-threading reduces execution slowdown by not blocking the system's execution with provenance information processing. However, there is a need for Cronista to process the provenance information faster than a system produces it.

RQ3 and sub-question answers

The third set of research questions explores how a developer might use the history model and provenance data a system produces at runtime.

LRQ3

How can a developer explore the history model to improve their understanding of a system?

LRQ3 answer A developer can use query languages and visualisation tools for the native storage technology that supports the history model. With the current CDO implementation of the history model, it was possible to use EOL queries and the CDO Explorer to access the data collected from the Traffic Manager experiment. However, developers are not limited to only these tools: it is feasible that visualisation or query tools could be created. In the demonstration, a small Java application was used to produce Graphviz-based visualisations.

LRQ3.1

How can a developer explore a provenance graph from a known point of interest, e.g. a failed state?

LRQ3.1 answer The demonstration presented an example of a developer finding a defect in a system by using EOL to query the history model. The developer followed an approach described by Herschel [54], starting with an element of search to find an approximate point of interest, and then step-wise navigation to the specific point of interest. EOL appears to be expressive enough to cover most scenarios for exploring provenance as described by Herschel.

In the example in the demonstration section, the developer used the metamodel for the shared-knowledge model to find the initial point of interest in the provenance graph; e.g. an entity representing the end-phase signal. However, a developer could also perform broader searches, as seen with the Graphviz examples where a provenance graph can be simplified to a type of provenance node. A developer could look for a point of interest by viewing chains of ACTIVITIES, or the evolution of part of the model represented as an ENTITY.

LRQ3.2

How can a developer write a provenance query to find answers to questions using the history model?

LRQ3.2 answer A developer could initially create a query to answer a question using the approach described in LRQ3. The demonstrated process incrementally builds a structured query that can be rerun to find a point of interest, without the need to perform searches or navigation. Thus, a query can be built once to answer a question and then reused to answer the same question on similar history models, e.g. different executions of the same system.

EOL can extend types with context operations such as `printLater` or `attr`, which are not part of the history metamodel. These features would make it feasible to create reusable libraries of functions to cover various scenarios or questions. It may be possible to package queries in a library form into a UI for domain experts that covers the most common cases or facets of questions.

Threats to validity of the demonstration experiment

The threats to the validity of the experiment with the Traffic Manager in the demonstration are considered against the classification provided by Feldt et al. [38]. Specifically, this section will discuss the internal validity threats, which examine if the treatment is the cause of the outcome. These are followed by a discussion of external validity threats, or the extent the results might generalise beyond this experiment.

Internal validity The simulations in the demonstration were only executed 10 times for each condition (fault and no-fault) to obtain the averages and standard deviations. Longer simulation executions and repetitions could expose unseen effects on resources. For example, a very small memory leak may not be visible in a short execution, in a long execution the same leak could cause a failure.

Having created the Traffic Manager system for experimentation with Cronista's instruments, the system design was influenced with the intention of collecting provenance for explaining execution. Thus, the experiment with a detectable seeded fault that could be explained with Cronista is slightly biased towards succeeding. In a real-world situation, an unexpected fault could be considerably more difficult to trace with Cronista; the system model and activity scopes would need to collect

suitable data to describe the system execution where the fault occurs. A similar success bias could be introduced if the system is designed using PLEAD or a similar methodology based on designing a system to meet some explanation requirements; which would create a system with model and activity scopes that intentionally explain specific parts of the system's execution for a requirement.

External validity The experiments with Cronista so far have produced history models for a Fibonacci toy experiment and the Traffic Manager. While the execution of these systems has produced a provenance graph that can successfully be mapped to steps/events in the system execution, the effect may not generalise to other systems or domains. Cronista's design has identified where some provenance collection concerns can be separated, and that the internal features of a system can be reflected in a provenance graph representation of a system's execution.

Discussion

Visualising the provenance graphs from the Traffic Manager using Graphviz was more difficult than the provenance graphs created with the Fibonacci example (Section 5.1.5) The increasing complexity of the provenance graph and limited control of node/edge layout in Graphviz required more manual effort to create the graph seen in Figure 5.17. Improvements to this visualisation approach could be made but this activity is outside of the scope of the research, and could also be considered an increased design time cost of Cronista; a developer effort for creating history model visualisation tools. Future work could consider implementing the history model with another technology, and exploiting existing tools for graph visualisation would save developer effort in creating visualisations. There are likely existing graph technologies that could address the complex problems for graph visualisation.

CDO was selected as the initial technology for the History Model Store, because it is a technology with which developers in the MDE community might be familiar. This in turn constrained the choice of compatible tools for working with the history model. The existing tools for CDO enable querying and visualisation, but these tools were developed for working with models rather than graph structures. Given the separation of concerns between a system and Cronista's Curator and History Model Store, the need for familiar modelling technology may not be as important as originally assumed. The provenance analysis and querying processes could be easier with a different technology that is more suitable for graph queries; a developer could be shielded from the complexities of the History Model Store in other ways, instead of relying on familiar MDE technologies.

Model query languages like EOL can be used to query provenance. However, EOL may not be the most concise language for certain provenance queries: for instance, a user may just want 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. Evaluating these languages should be part of the next development cycle in this thesis.

5.3 Extending history model store technologies

5.3.1 Motivation

Provenance data can be stored and searched using general data handling technologies. CDO has been used in the previous design and development cycles as Cronista's History Model Store permitting the use of existing tools for experiments: CDO Explorer for visualisation, and EOL for provenance queries. However, visualising provenance at scale required a custom Java program to transform the provenance in CDO into a GraphViz file. The resulting file output could then be viewed with GraphViz, which is not a CDO-specific tool.

Using customised tools to access Cronista's history model could enable an improved user experience. Improvements would come from tailoring the tools for a specific user audience and system; these tools could sacrifice their ability to generalise to a wider audience and instead target particular user requirements. Creating custom tools for viewing and querying the history model and provenance graph is not in the scope of this research thesis. However, there may be a set of existing tools that are a better fit for the History Model Store and analysis.

5.3.2 Problem

CDO was selected for the initial implementation primarily because it was an MDE tool for storing models. Both provenance and shared-knowledge can be implemented as a model and stored using CDO, using one model technology for two different purposes. This duality of use enables a developer to work on shared-knowledge and provenance collection under a common technology. However, since the initial implementation of the History Model Storage, its scope and purpose have become more clearly defined. Creating a self-contained approach to a History Model Store, and delivering it as a packaged component, could reduce the design-time costs relating to developer effort.

A developer is currently restricted to a set of MDE tools for CDO, or must implement their own for data analysis in the History Model Store. Provenance analysis using existing CDO tools was possible, but may not be optimally effective. Models and graphs are similar, but they can be used very differently and these different uses affect the kinds of tools developers create as there are specific problems for each use case. Given Cronista's emerging modular design, it should be possible to implement another History Model Store using a different data storage technology, which would be packaged as a modular component for developers, allowing the toolset for analysis of the provenance data to be changed.

Only one History Model Store has been implemented using CDO, which could cause additional design-time costs if it fails to scale sufficiently for a different use case. The current scalability of Cronista's History Model Store relies on the Curator using a single instance of CDO to store the history model. To overcome the limitations of CDO, a developer might need to alter Cronista's Curator to use multiple CDO repositories, or a developer might be forced into creating another History Model Store implementation. It would be desirable for Cronista to have a History Model Store technology that could be scaled up externally to Cronista's components; e.g. the technology can be configured for horizontal scaling or to use different databases, without changing the Curator component.

5.3.3 Objective

Cronista's initial implementation included a history model based on CDO, which imposes some limitations that would require additional design time effort for a developer to solve. The modular design of Cronista's architecture presents an opportunity to reduce these costs. The History Model Store component could be conveniently packaged for a developer, who would choose one and configure it for use with their system. Several packaged history models stores could then be created using different technologies, and these could then be re-used depending on their suitability.

During the experimentation with EOL, it became apparent that a more declarative approach to query the provenance graph would be convenient. This led to a consideration of graph database technologies (which provide pattern-based graph query languages) for the provenance graphs. Previously, the proposal that the history model should store model snapshots in addition to provenance graphs complicated the use of a graph database, since system model copies would need to be converted into a graph-compatible form. However, the initial implementation in Section 5.1.4 in-

memory EMF model snapshots, did not store a snapshot copy of the model in the History Model Store; a path and file name is used to reference where the snapshot could be found. Thus, a graph database History Model Store could use a similar referenced snapshot approach; these snapshots could taken and held in the system's shared-knowledge model repository (e.g. CDO)

The objectives of this design cycle should provide answers to the following local research questions.

LRQ1 *To what extent is it possible to create a drop-in replacement for the CDO-based history model store based on graph technologies, without affecting the system being monitored?*

LRQ2 *How does the developer experience change when using graph technologies for analysis of the history model?*

LRQ3 *How do the runtime resources for a graph technology-based history model compare to the previous CDO-based implementation?*

Answers to these research questions should expand on the answers to RQ3 and sub-questions.

RQ3 and sub-questions

RQ3 *What new knowledge of provenance awareness costs was discovered by the design and development of an artefact to answer RQ1?*

- **RQ3.1** *What are the design-time activities for using the new approach, compared with existing provenance awareness methodologies?*
- **RQ3.2** *What runtime overheads were seen on the systems used to evaluate the implementation, and how do they compare to the results published for provenance awareness?*

5.3.4 Design and development

Apache TinkerPop [3] was chosen to provide the alternative History Model Storage based on graph databases. As a vendor-agnostic platform, it provides flexibility within its technology stack. The TinkerPop API enables support for different graph database technologies and graph analysis tools with

a common API. Some graph databases specialise in a horizontal or scale-out distributed architecture, while others seek to offer low-latency graph accesses through vertical scaling. There is also a growing ecosystem of graph analysis tools enabling the visualisation and querying of large graphs. In summary, TinkerPop presents many opportunities for configuring different History Model Store implementations and provenance analysis techniques, requiring only a single (TinkerPop) History Model Store adapter to be implemented.

Some of the graph databases that support TinkerPop also offer configuration options that are relevant for Cronista's History Model Storage. For example, we can consider JanusGraph [109], a scale-out TinkerPop-compatible graph database, which is optimised for storing and querying large graphs, and is a free software project under The Linux Foundation. JanusGraph can use several databases for storage, such as Apache Cassandra, Apache HBase, Google Cloud Bigtable, Oracle BerkeleyDB, or ScyllaDB. Each database adds further options for tuning the performance of the History Model Storage. Furthermore, the feature set offered by JanusGraph enables elastic and linear scalability, with data protection through replication and backups.

Using Tinkerpop gives access to the Gremlin Query Language [4]. The Gremlin language is used to form queries as a series of edge traversals between nodes (vertices). A developer or expert user can write and execute Gremlin queries with a console application (the "Gremlin console"). Furthermore, several language drivers are provided for other programming languages besides Java. These drivers enable Gremlin queries to be integrated into the code of another application. Gremlin queries can be used to search for provenance nodes by type or property, and are well suited to navigating provenance [54] by iterating on or extending a query to traverse edges between nodes.

The Curator design required adaptation and improvements to the separation of concerns with the storage interfacing: the initial design in Figure 5.1 (on page 126), was extended to the design in Figure 5.18. Provenance messages are processed with the same Curator logic as originally implemented in Section 5.1. A new History Model Storage API layer has been introduced, which provides the Curator with a common API for different History Model Storage technologies.

A graph-based history model was also conceived since a graph database cannot create identical data structures to a model repository. Each time window is represented as a node; edges connect the time window node to the prior and following time windows. A time window has edges to two child nodes: one containing the snapshot reference, and the other acting as a root node for the provenance graph. Within the provenance graph in CDO, an ENTITY could store an attribute's value

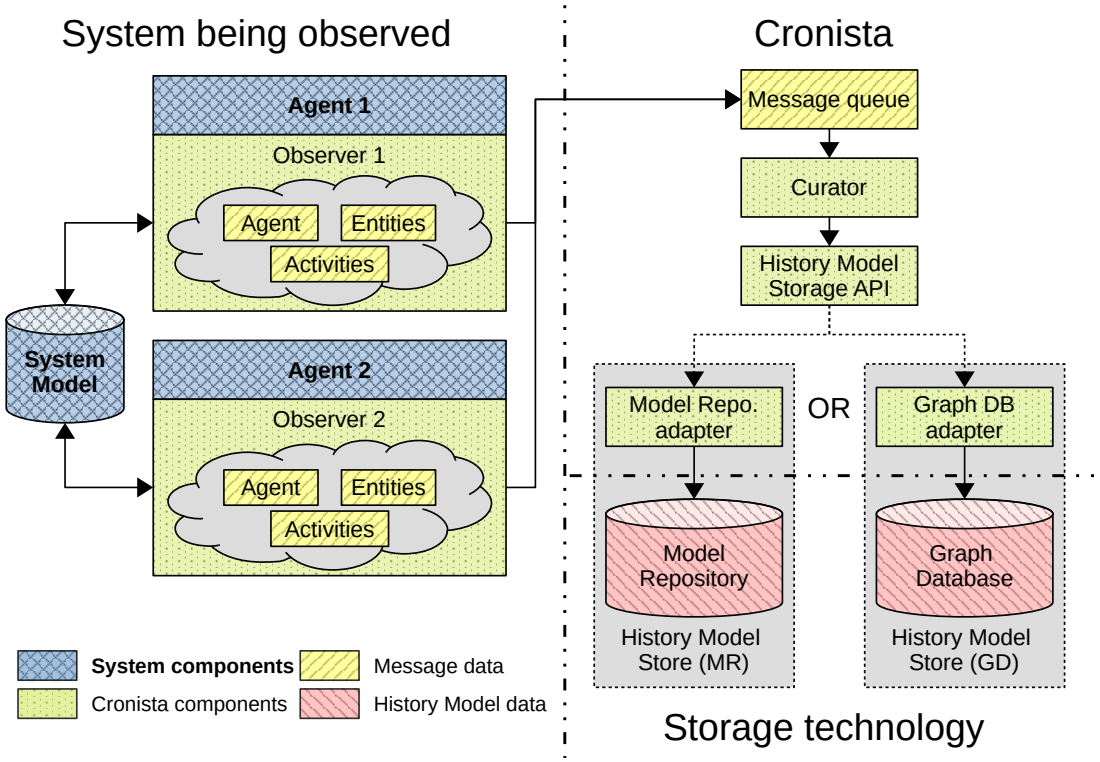

Figure 5.18: Overview of Cronista's extended architecture: a new History Model Storage API and adapters have been added to enable History Model Stores to be implemented with different data storage technologies

using a (raw) object; in this way, entities could express values of many different types. In the CDO ENTITY the attribute's value and type were stored as strings for convenience along with the raw object. A Tinkerpop graph node cannot store an attribute value as a raw object: thus, some attributes may only be stored using a string representation.

When the Curator uses a CDO History Model Store, the provenance graph model objects are instantiated in the Curator's memory with references for the provenance relationships. The model repository adapter component in Figure 5.18 is now used to create and hold the CDO provenance model. The model repository adapter commits the small in-memory model to the larger provenance graph model stored in CDO when the Curator starts a new time window via the History Model Storage API. Similarly, the graph database adapter handles a storage-specific implementation of the history model and provenance graph.

In the graph database approach, the graph database adapter uses graph queries to instruct the database to add provenance nodes/edges to the graph. This leaves the slow IO operations to the graph database to handle. However, the graph database adapter can reduce the graph database's workload by giving more precise instructions to the graph database (i.e. supplying graph node IDs,

instead of instructions to perform a search or traversal to locate a node on the graph). The use of more precise queries to add nodes and edges should enable the graph database to quickly build a provenance graph and process higher volumes of provenance messages.

In order for the Curator's graph database adapter to be able to give these more precise instructions to the graph database, it needs to maintain an index of graph node IDs whose creation it requested. This is facilitated by the graph database adapter component, which contains the implementation of the TinkerPop query building and indexing of nodes (Figure 5.18). New node IDs are returned from the graph database to the adapter when a new node is created by a query. A small index of (node ID, and provenance ID) pairs enables the adapter to send precise queries to the graph database. For example, when adding a relationship (edges) between two graph nodes the node IDs can be provided. If the node IDs are not indexed against their provenance IDs, then the Curator would need to issue queries to find two graph nodes matching some provenance IDs before requesting the creation of an edge.

A graph database's index tracks all nodes on the graph (sum of all time windows and all provenance graphs nodes); thus, the history model is one large graph. The Curator could have the graph database perform searches for provenance nodes, to discover existing provenance nodes that do not need to be created to record a provenance message. This search is an overhead that could be avoided by having the Curator maintain a small index of graph nodes relating to the provenance graph on the current time window. These indexes will be smaller than the graph database's entire index, for a small cost of some additional memory usage in the Curator. The adapter's index of the graph nodes is populated as nodes are created, and the index is cleared after closing a time window: the adapter only indexes a small subset of relevant nodes.

It is safe to clear the adapter's graph node index with the end of a time window, as we do not want to create edges across time windows. As discussed in Section 4.2, each time window should be independent so that time windows can be "forgotten" or removed from storage. If a provenance node is repeated from a prior window (i.e. has the same provenance ID), then a new graph node (ID) is needed as a child of the current provenance graph for the current time window. Clearing the adapters' index enforces a clean separation between provenance graphs, and prevents the index from growing uncontrollably over a long execution that creates many time windows.

Figure 5.18 shows two History Model Stores: one based on a model repository (CDO), and one based on a graph database (TinkerPop/JanusGraph). These History Model Stores sit below

the adapters that perform the storage technology-specific interactions, which could occur over a network connection. CDO and TinkerPop provide a network API for remote access: this enables an instance of CDO or TinkerPop to be launched independently of Cronista. Docker [62] can be used to package a pre-configured instance of a storage technology for use as an independent History Model Store. These Docker container images are readily available for reuse from public repositories (e.g. Docker Hub), and maintained independently of Cronista and of the system which is being made provenance-aware. Similarly, the Docker build and configuration can be tuned without requiring changes to Cronista. A developer's design-time efforts for setting up a storage technology would then be: i) to choose which history model they want to use, ii) to set the Curator to use the chosen store type, and iii) to deploy a Docker container.

5.3.5 Demonstration

The code used for the TinkerPop-based History Model Store follows similar patterns and techniques as the initial implementation (Section 5.1.5). For this reason, and to save space, this demonstration section will not give code examples: the code is available for inspection on Cronista's Gitlab repository [94] [4].

The storage adapter components track recently created provenance nodes using indexes, and the indexes are cleared with each passing time window; a behaviour previously implemented in the Curator's code. Both storage adapters should produce equivalent (storage-specific) provenance representations for the same instruction from a Curator; the adapters should not modify the provenance sub-graph representations a Curator produce for each type of provenance message (e.g. the provenance message and sub-graph defined in Sections 4.1.4.

To demonstrate Cronista's history model extensions, the experiment performed in the previous design cycle demonstration (Section 5.2.5) can be repeated. A repeat of the experiment can test that the new TinkerPop-based History Model Store works correctly: it should produce a history model that is structurally equivalent to that produced by CDO. Using the same traffic manager fault-finding exercise, a graph query can be written to find the same root cause for the fault: previously, this was found using EOL model repository queries. The new graph queries can then be compared with the model queries from the previous experiment. To compare runtime resource metrics, the underlying

[4] Java package `https://gitlab.com/sea-aston/cronista/-/tree/master/Cronista/uk.ac.aston.provlayer.storage.tinkerpop?ref_type=heads`

storage for both options will be provided via Docker containers: this will allow for measuring their use of memory, CPU and I/O at the operating system level.

Graph database selection and configuration

To perform the experiment and compare a model repository with a graph database technology for use as a History Model Store, it is necessary to pick a specific implementation of a TinkerPop-compatible graph database. To assist with the reproducibility of the results, it was decided to focus on generally available open-source technologies with broad user adoption. Having the source code available was considered beneficial for investigating unexpected behaviours and for the transparency of the evaluation process. This section describes how the specific storage technologies were chosen.

As discussed in Section 5.3.4, there are many different graph databases and graph model variations. Several surveys and benchmarks have been published over the evolving state of the art in graph databases [27, 22, 64, 68]. The database configuration used for different graph workloads can affect the performance of graph processes. The above studies documented the time required to load a graph into different graph databases, and reported significant differences between them in various performance metrics. This is important for Cronista, as a graph database will have to be fast enough to allow the Curator to keep up with the incoming messages from the Observer while the system runs. A slow graph database would cause a bottleneck, requiring either dropping some messages, or reducing the rate at which the runtime model changes (e.g. by running the feedback loop less often, or slowing system execution via back-pressure from the message queue).

In short, the first step was to find an open-source TinkerPop-enabled graph database that was mature, easy to deploy, and supported horizontal scaling out-of-the-box for future needs.

As a key player in the graph database market, Neo4j was considered for its open-source license, but horizontal scaling was only available in its proprietary enterprise edition. On the other hand, JanusGraph (formerly named TitanDB) provided horizontal scaling to all users, and implemented a variety of backends of its own that could be selected depending on the situation. OrientDB was also considered due to its open-source license and built-in support for multi-master replication, but the broader range of possible configurations for JanusGraph and its adoption by key players such as RedHat and Netflix motivated the selection of JanusGraph as the first choice of study.

Some experimentation was required to find a configuration that would ingest data at a quick rate. To provide an initial approximation of the ingestion rate, a small Java program was written to measure

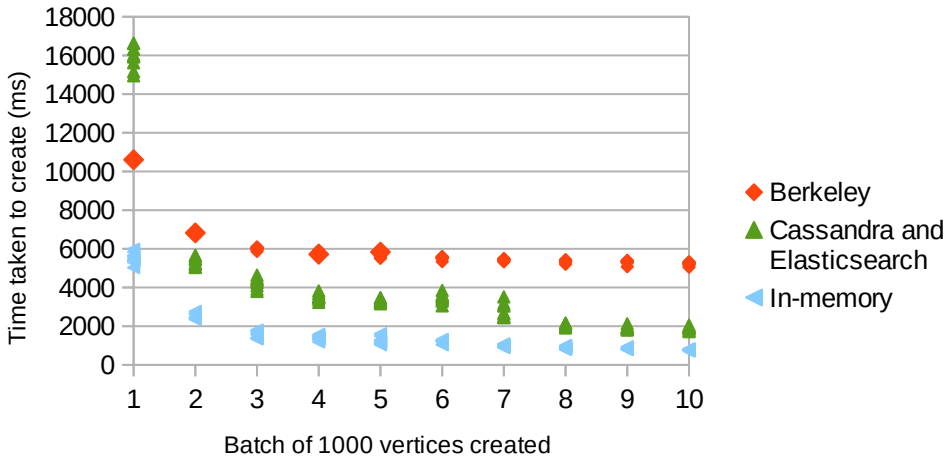

Figure 5.19: Benchmarks for creating 10,000 vertices in batches of 1,000 with different JanusGraph backends.

the time JanusGraph needed to create 10,000 vertices (without edges). Times were measured on top of three different JanusGraph backends: one based on the Berkeley database, one that combined Cassandra and Elasticsearch, and an ephemeral in-memory representation. The default configuration parameters (e.g. buffer sizes) for each backend and for JanusGraph in general were chosen, on the assumption that these had been set as a safe default for most developers. The 10,000 vertices were created in 10 batches of 1000, recording the time taken: this was repeated 10 times. These initial benchmarks were run on a 4-core AMD Phenom II X4 970 Processor with 16GB of DDR3 RAM and a SATA3 SSD, and the results are shown in Figure 5.19.

In these results, it is noticeable how JanusGraph speeds up after the first batch of nodes, and how it speeds up even further after a few more batches. It appears that the initial creation of its internal data structures (e.g. the allocation of an initial block of node IDs) slows down the insertion of the first nodes, and then its ingest rate improves considerably.

The results of this initial experiment align with similar published results such as those of Barquero et al. [6] who also found that disk I/O can be restrictive for real-time processes that use a graph database. Based on these findings, it was decided to initially evaluate JanusGraph with its in-memory backend. It must be noted that in-memory graphs are constrained by the available amount of system memory and are not persisted to disk: we discuss this threat to validity in Section 5.3.6. For the present case study, this constraint on size was not significant.

Traffic manager experiment setup

The experiment setup is broadly the same as the previous demonstration in Section 5.2.5. The notable differences can be seen in Figure 5.20: the experiment now uses Docker containers for some system components. Docker containers were introduced as a means to uniformly package different technologies as a History Model Store. However, containers are also useful for performing repeatable (scripted) computer simulation experiments, and metrics for Docker containers can be measured during the execution of an experiment. The containers can be destroyed and recreated between the executions to reduce the effects that caching or execution optimisation may have on runtime metrics. Finally, it is possible to switch Cronista back and forth between different History Model Store Docker containers without making substantial code changes.

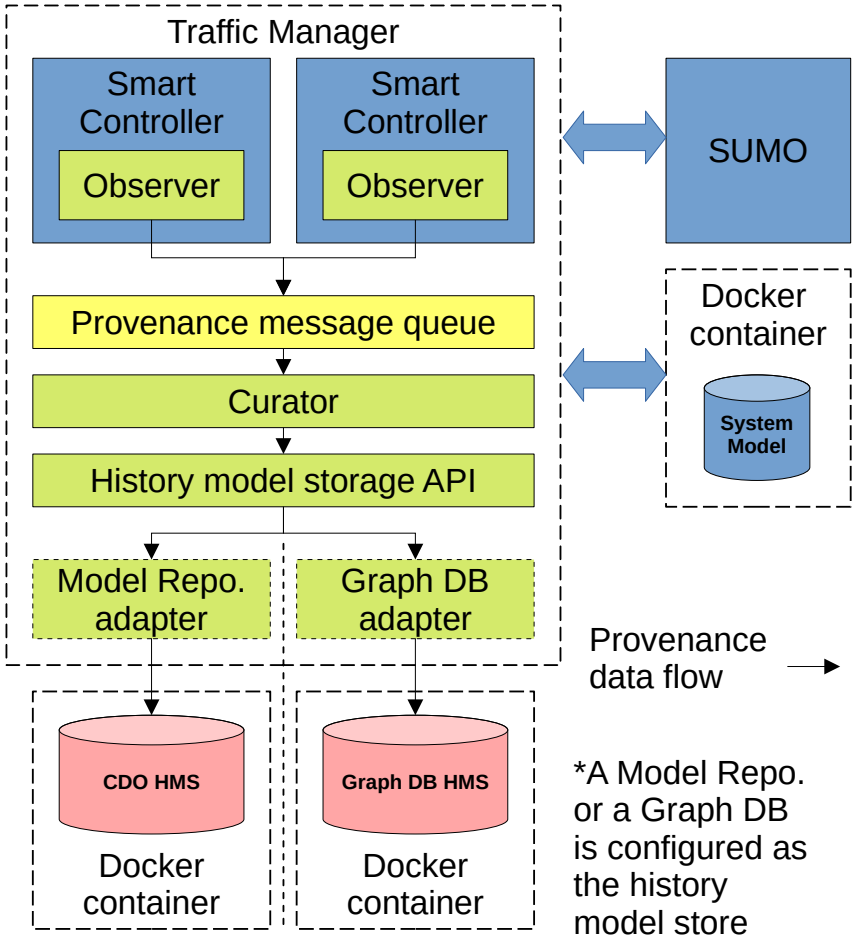

Figure 5.20: Component setup for the experiments to compare the use of a model repository and graph database History Model Store

Experiment conditions During each experiment run, four processes were running on the same machine. The traffic manager's Instrumented Controller (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[5]. The HMS was managed by a second Docker container, using either the same CDO Docker image, or the JanusGraph 0.5.3 Docker image[6].

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 been 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.

Results

Design-time cost observations The design-time cost for applying Cronista's provenance instrumentation code to the traffic manager system for agents/activities/entities is the same for this experiment. Changing the Curator to use a different History Model Store had no impact on the agent Observer, activity scopes, and model instruments previously applied to the traffic manager system.

Cronista's Curator components underwent some changes in their instantiation and setup code. However, some of these changes were also required to enable the traffic manager to conveniently switch between a model repository and a graph database for the experiment. As discussed in Section 5.2.5, this type of developer effort is a one-off design-time activity, which would not vary greatly in cost between systems.

Managing some of Cronista's components with Docker containers helped to reduce developer effort: for example, when setting up the History Model Stores for the experiments. Once a Docker

[5] `https://gitlab.com/sea-aston/cdo-docker`
[6] `https://github.com/JanusGraph/janusgraph-docker.git`

image has been built, a Docker Compose configuration can be created to enable the rapid deployment of a container in a Docker environment.

Runtime metrics Table 5.3 provides statistics on the execution times for the IC, the memory usage for the IC and History Model Stores, and the network I/O of the History Model Stores. The mean IC time for JanusGraph on 200 ticks is nearly 10 seconds (25%) longer than the time without Cronista. This is mostly because 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 additional time after the simulation has been completed. On the other hand, the CDO History Model Store 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: for 1000 ticks, the IC with JanusGraph only takes 3s longer to complete than the 152.90s of the IC without provenance collection. As Figure 5.19 showed, JanusGraph speeds up after the first few thousand vertices have been added, and becomes comparable to CDO in insertion speed. In general, it can be seen that as the simulation runs for longer and longer, the impact of the provenance collection process on execution times is reduced.

		200 ticks		1000 ticks		5000 ticks	
Metric	HMS	Mean	SD	Mean	SD	Mean	SD
IC time	None	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
	Janus	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
	Janus	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
	Janus	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
	Janus	10.12MB	0.10MB	53.20MB	0.96MB	262.40MB	6.52MB

Table 5.3: 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.

On the other hand, the memory usage of the IC does grow over time with the current implementation. With 200 ticks, collecting provenance information increases the original 11MiB of memory usage to around 18MiB. For 1000 ticks it goes from 11.51MiB with no provenance, to 55MiB with the

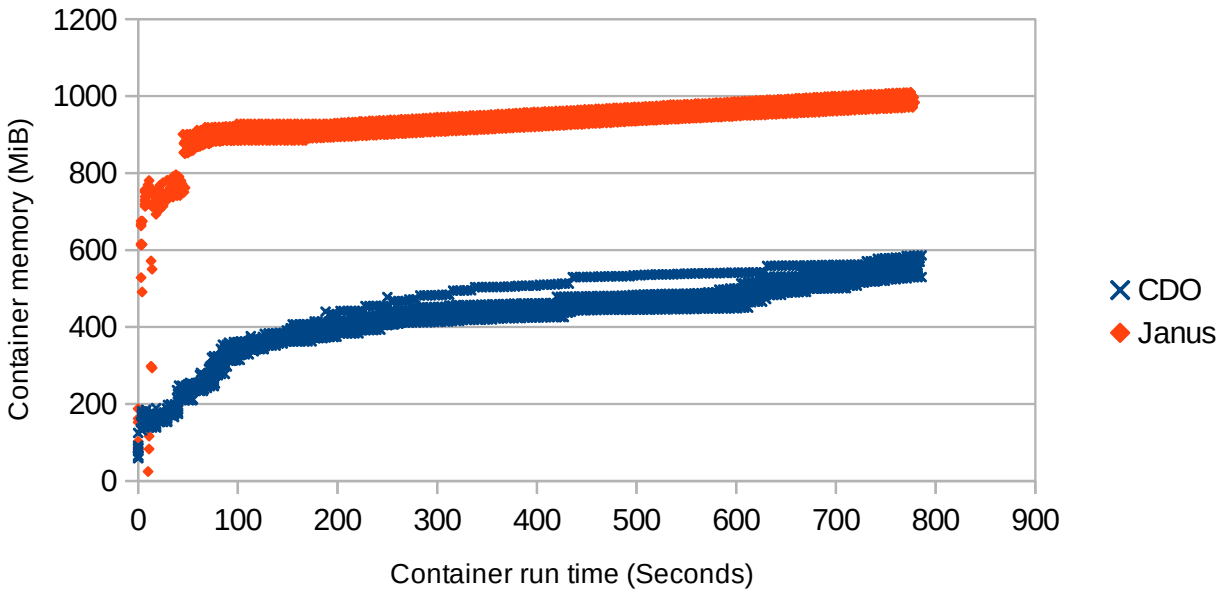

Figure 5.21: Evolution of memory usage by the Docker containers over 10 runs of 5000 ticks: times include server startup, a 20s wait to allow the server to settle down, and the execution of the simulation. Data points for all 10 runs are included.

CDO HMS and 36.33MiB with the JanusGraph HMS. Finally, for 5000 ticks it goes from 14.74MiB to 227.65MiB with CDO and 127.44MiB with JanusGraph. We anticipate that this increase over time is due to the 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 TinkerPop + JanusGraph History Model Stores 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. Figure 5.21 shows the evolution of the memory usage from the CDO and JanusGraph Docker containers across all 10 runs with 5000 ticks, showing how JanusGraph has higher baseline memory requirements but has a slower growth in memory usage as the simulation proceeds.

Table 5.4 shows how much disk space the embedded H2 database in CDO used on average across 200, 1000, and 5000 ticks. The disk space increased linearly, and disk usage figures stabilised across runs for longer histories. On average, each tick took about 35.88KB of disk space in the HMS

Simulation Ticks	Mean	SD	Mean/Ticks
200	6.16MB	1.09MB	31.55KB
1000	37.83MB	0.00MB	38.74KB
5000	175.20MB	0.00MB	35.88KB

Table 5.4: CDO HMS Container volume sizes after simulation

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 to limit disk usage.

These differences between JanusGraph and CDO must be weighed against other factors, such as the tool ecosystems available for each. The rates of memory consumption are comparable and show that over longer runtimes, there is a need to use disk storage. As identified in the prior section which discussed graph database selection and configuration, further investigation is needed to find a JanusGraph configuration with enough throughput to persist the graph to a disk while a system is running.

Graph query (fault finding) A query was developed to trace the root cause of junction signals changing frequently, as seen in the previous experiment with the Traffic manager in Section 5.2.5. Both query languages are used in this demonstration to check the cause of a junction signal change; this tests whether the instruments collect equivalent provenance. Furthermore, the style of EOL and Gremlin query languages can be shown in the context of being used to query a provenance graph.

EOL query The EOL query listing is presented in Listing 5.16, this is the same as seen in Section 5.2.5. With EOL, a developer needs to use code loops to iterate through connected nodes to find matching nodes. In Figure 5.22 the provenance layers revealed by each section of the EOL query have been mapped. This mapping of the EOL query to the provenance layers can be compared with the Gremlin query below.

Gremlin query A Gremlin query (shown in Listing 5.17) was developed to follow the same path as the EOL query. A developer iterated over the query in stages, stepping through the layers of provenance and reviewing query results. The process was similar to the one seen in Section 5.2.5 with the development of the EOL query.

Gremlin has convenient functions such as `valueMap()` that can display the properties on a

```
1  // 1. When was PhaseEnded set to true?
2  var ePhaseEnded = Entity.all.select(ed |
3    ed.attr() == 'PhaseEnded' and ed.bool() and ed.isWrite());
4  for (entJT in ePhaseEnded) {
5    entJT.printLayer(0);
6    // 2. Where did the EndPhase this activity used come from?
7    for (entUsed1 in entJT.wasGeneratedBy()?.Used
8          ?.selectOne(e | e.attr() == 'EndPhase')) {
9      entUsed1.printLayer(1);
10     // 3. What information was used when EndPhase was set?
11     for (entUsed2 in entUsed1.wasGeneratedBy()?.Used) {
12       entUsed2?.printLayer(2);
13     }
14     '\n=end='.println();
15 }}
```

Listing 5.16: Example of EOL query tracing the excessive count of traffic light phase endings back to its cause.

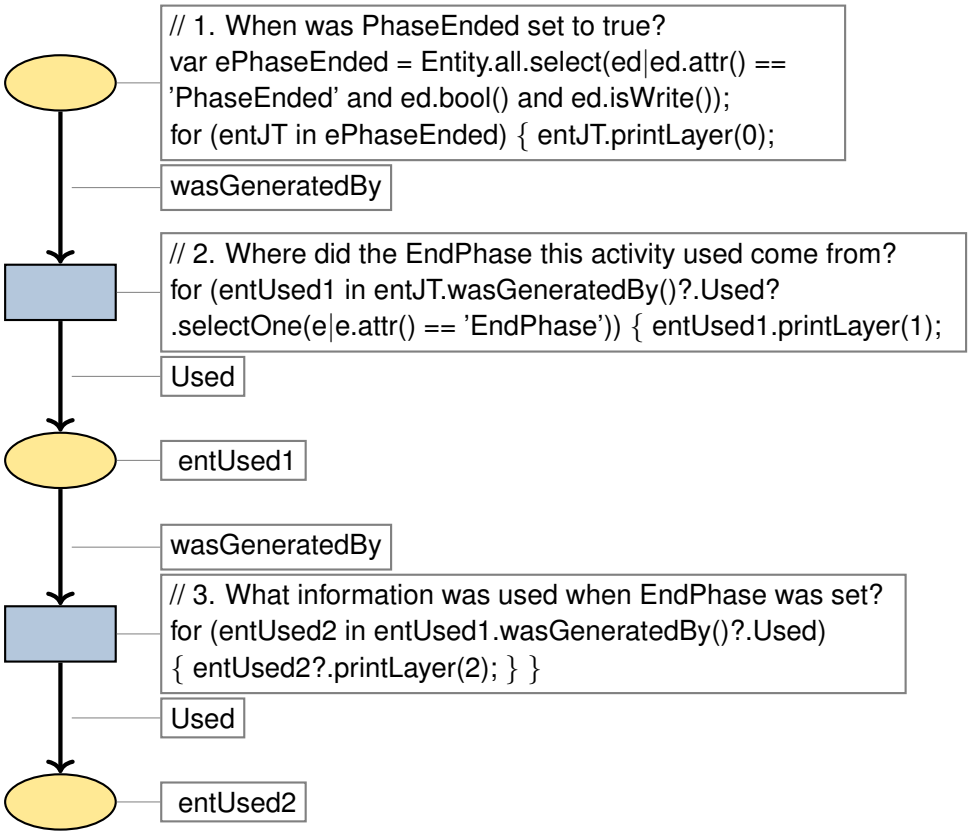

Figure 5.22: EOL graph query used to find the root cause of the defect from the collected provenance graphs, visualised as a graph pattern. Ovals represent entity nodes, and rectangles represent activity nodes.

```
1  g.V().
2    has('Entity','AttributeName','PhaseEnded').
3    has('AttributeValue','true').
4    out('WasGeneratedBy').out('Used').
5    has('AttributeName','EndPhase').
6    out('WasGeneratedBy').out('Used').
7    has('AttributeName','LADsJammed').
8    valueMap()
```

Listing 5.17: Example Gremlin query tracing the excessive count of traffic light phase endings back to its cause.

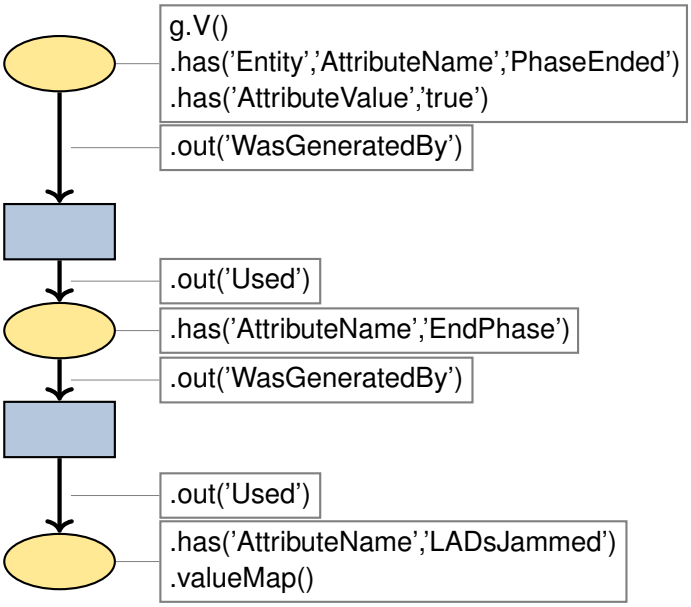

Figure 5.23: Gremlin graph query used to find the root cause of the defect from the collected provenance graphs, visualised as a graph pattern. Ovals represent entity nodes, and rectangles represent activity nodes.

vertex, and these are helpful when exploring a provenance graph. Accessing the next layer of nodes in the provenance graph through a relationship can be done with `out()`. The `valueMap()` in combination with `out()` can be used to explore the graph in steps, starting from a specific point of interest. For example, starting from an ENTITY, a step can be taken to each connected ACTIVITY, which could be via a *"used"* or *"generatedBy"* relationship.

This step-wise approach to exploring provenance information - accessing the ENTITIES *"used"* by an ACTIVITY and then stepping into their related activities - is likely to be a common operation in any investigation. The simple and repeatable pattern of combining these two operations is friendly to new developers, and it is closely related to the navigation approaches described by Herschel [54].

Contrasting observations Comparing the EOL query in Listing 5.16 and the Gremlin query in Listing 5.17, 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 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.

Graph visualisation Graphexp is a lightweight web interface for visualising and exploring TinkerPop graphs. As such, Graphexp has a search facility to help find nodes of interest in a graph. Figure 5.24 shows a screenshot of Graphexp being used to explore a provenance graph on a time window [7]. The graph shown is only a small section of a larger graph. Graphexp limits the number of nodes shown on the screen when performing a search. Navigation is possible by clicking on a node shown on the graph. Selecting a node exposes all edges and nodes directly connected to it. A user navigates the graph by clicking across a sequence of neighbouring nodes. As the distance (number of edges) between an old node and the current node increases, the node begins to fade before it is removed.

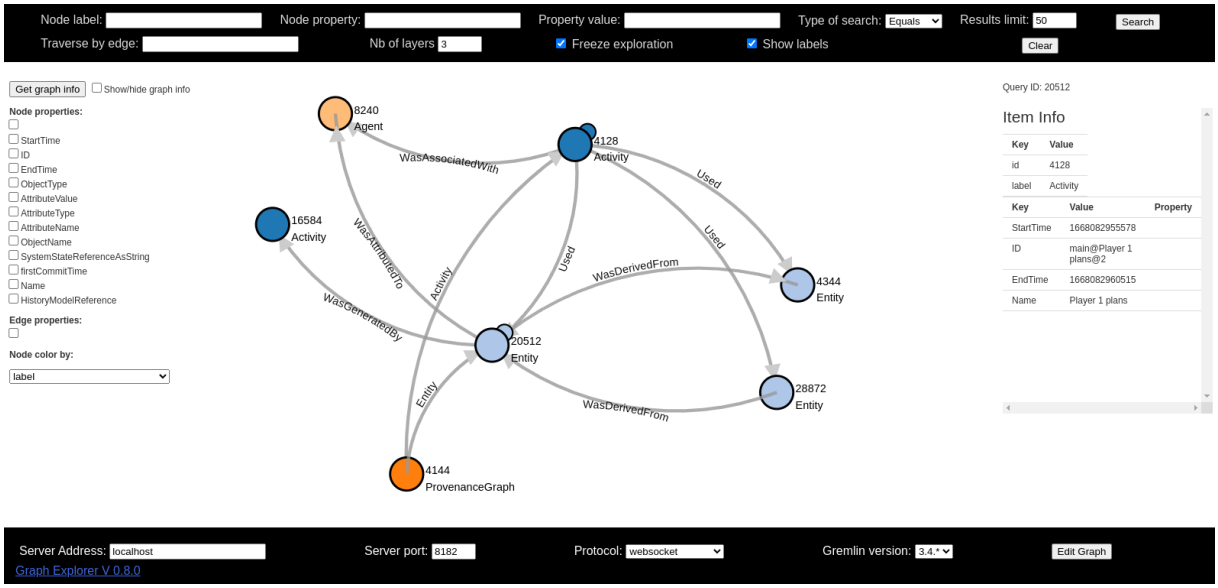

Figure 5.24: Example screenshot of Graphexp rendering a provenance graph

Graphexp is the most suitable candidate for visualising provenance from the small selection of tools reviewed (CDO Explorer, Graphexp, and Graphviz). However, Graphexp has its limitations, which prevent it from being the ideal tool. The biggest problem is that Graphexp can be quickly overwhelmed by a node with many edges. When a node with many edges is clicked, all attached nodes and edges are added to the screen, which results in the application slowing down and, in the worst case hanging. Thus, all the work to search and navigate to the point of interest in the provenance graph is lost. As such, the tool seems more useful for checking local areas and short navigating hops between nodes.

[7] Screenshot is of Graphexp showing a provenance graph created in an experiment in Chapter 6

5.3.6 Evaluation

The cycle of development in this section is evaluated as follows: the research questions and answers are presented. This is followed by a discussion of the threats to validity of these findings and answers, and finally, a discussion of the broad findings.

Research question answers

The local research questions are repeated and answered in the order in which they were presented in the objectives Section 5.3.3.

LRQ1

To what extent is it possible to create a drop-in replacement for the CDO-based history model store based on graph technologies, without affecting the system being monitored?

LRQ1 answer The Curator's history model-building logic is based on the expected behaviour of the History Model Storage API. A layer of abstraction between the Curator and the History Model Store can be used to provide the Curator with a common History Model Storage API. This separates the Curator implementation from the specific API of the storage technology to be used as a History Model Store, allowing adapters for different storage technologies to be developed independently of the Curator. Each adapter creates a bridge connecting the History Model Storage API and the technology-specific API.

This separation of concerns also enables experimentation with different techniques for writing to storage. Cronista's current History Model Storage adapters use a simple provenance information cache that clears when a time window closes. This caching approach avoids the need to perform searches for provenance nodes while writing information to the provenance graph. However, future work could investigate ways to optimise provenance graph writing for different technologies.

LRQ2

How does the developer experience change when using graph technologies for analysis of the history model?

LRQ2 answer As discussed in the demonstration in Section 5.3.5 graph query, EOL and Gremlin query languages have different styles. A developer's preference for imperative or declarative style of query, and/or their experience with a particular style or query language are likely driving factors for the selection. However, Gremlin's intended use for querying graph structures with an imperative style seemed more intuitive to a novice developer new to both query languages. Overall, the Tinkerpop ecosystem for graph databases probably presents a better set of tools for accessing provenance graphs than existing MDE tools.

LRQ3

How do the runtime resources for a graph technology-based history model compare to the previous CDO-based implementation?

LRQ3 answer In the demonstration, the metric results show that baseline resources required by CDO and JanusGraph are different. There is a notable limitation with JanusGraph's throughput for consuming provenance messages while persisting a graph to disk. A naive in-memory database configuration was used to overcome the problem; optimising a graph database configuration is not the intended focus of the research.

CDO did not have a problem with consuming provenance messages and recording the data to disk. CDO did not require elaborate configuration changes to enable provenance messages to be consumed at the rate required, whereas JanusGraph had a more complex set of configuration options. The rate of provenance message consumption is an important metric to consider for runtime performance. In this regard, due to CDO's simpler configuration and higher consumption rate, which makes it capable of writing to storage, CDO is a better option than a JanusGraph for recording provenance at a high rate.

Threats to validity of the demonstration experiment

The Traffic Manager experiment in the demonstration section could have some threats to validity. These threats to validity are considered using the classification provided by Feldt et al. [38]. Specifically, it will discuss the internal validity threats of the treatment causing the outcome, followed by external validity threats, which cover how well the results could be generalised.

Internal validity The treatment of the resource measurements at the Docker container level may be overestimating resource usage when comparing the different storage technologies, since components other than the storage technology in the container may affect the measurements. Given that both model repositories and graph databases consist of several software layers that introduce unpredictable overheads, the data growth rate of the history model may be obfuscated. Therefore, a more accurate measurement of the history model growth rate is needed to produce a model that could predict the history model resource usage for a given duration that a system runs.

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. There is a reasonable argument that it is necessary to refine the JanusGraph configuration to allow longer periods to be captured and stored, with a back-end that persists history to disk and still has the throughput needed to keep up with the simulation.

The JanusGraph database can be configured in a variety of different ways, including the choice of the backend, its integration with the operating system, and the internal performance tuning parameters of the tool. It may have been possible to find a more optimal configuration of JanusGraph that would provide better performance and resource metrics for the graph database version of the History Model Storage. Likewise, the TinkerPop History Model Store implementation could be improved to reduce the processing demands on the graph database.

External validity 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. However, other model repositories and graph databases could be explored, as well as alternative configurations and optimisations in future work.

Cronista has still 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 History Model Store 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 message queue may fill up, and it may be necessary to drop some messages or introduce backpressure and ask the Observer to focus updates on a specific area.

The current version of Cronista depends on the Eclipse Modelling Framework and its `EObject`s for instrumentation, limiting the scope of which systems can be integrated with it. Therefore, other methods for instrumentation will need to be explored, such as aspect-oriented programming (this is discussed in Chapter 6).

The system models were designed by following the well-known MAPE architecture. MAPE is commonly used in the self-adaptive systems community. However, there is still the risk that since the Traffic Manager in the case study was created from scratch for the experiment, the design of the system models may be biased towards supporting automated provenance collection. Further studies will be needed to validate that the approach can be reused with system models "in the wild", which have not been designed with the idea of automated provenance collection biasing their internal structures. In the next design and development cycle (Section 6.2), an existing system without this bias is investigated.

Discussion

In this development cycle, a separation of concerns between the Curator and the History Model Store was introduced. This enables a single Curator module with history model building logic to be implemented and re-used against different storage technologies that could be used as the History Model Store. A developer only needs to create a suitable adapter for a specific storage technology API, the Curator will record provenance to a graph using the same graph fragment patterns.

From the initial investigations of MDE and graph database technologies, both seem to be suitable for use as a History Model Store. However, they both have some limitations or undesirable characteristics that make neither one a 'perfect' solution which generalises to all use cases. CDO offers a higher provenance recording rate, but it is limited to vertical scaling (at the time of writing). JanusGraph could not record provenance fast enough to store drastically reducing its scalability to an in-memory database. However, if a slow rate of provenance recording can be tolerated, JanusGraph could scale horizontally beyond CDO's limits.

Some of the existing tools for MDE and graph databases have been used to access the provenance

graphs within the History Model Store.

Summary: Existing tools tested for accessing the History Model

- **Text-based querying approaches**
  - EOL queries (CDO)
  - Gremlin queries (TinkerPop)
- **Visualisation approaches**
  - CDO Explorer, form-based (CDO)
  - Graphviz, via Model-to-text transformation (CDO)
  - Graphexp, interactive graph visualisation (TinkerPop)

The most intuitive ecosystem of tools for accessing provenance graphs is likely to be found in the Graph database technology space. Both Gremlin and Graphexp have provided the most intuitive and functional experience for a novice developer. However, a developer's experience and preferences could cause different opinions to arise that challenge this opinion. Given the limitations on the rate of provenance consumption, the option of CDO or JanusGraph may be forced by the rate a target system produces provenance. If a JanusGraph database configuration can not be optimised sufficiently to record provenance to disk storage, CDO is needed; otherwise, another storage technology and History Model Storage API adapter are needed.

Chapter 6

Design and development: non-MDE shared-knowledge bases

Cronista demonstrates an implementation of a provenance collection approach based on a shared-knowledge model. However, Cronista is very dependent on leveraging MDE technologies to reduce the design time costs for applying provenance awareness, by using MDE tools to automate the coding effort that is needed for a developer to instrument a system's shared-knowledge model. Without using MDE to automate this process, the design-time cost for implementing provenance awareness increases.

Cronista's dependency on MDE tools also presents a challenge with testing the approach against a wider array of systems. MDE approaches to system design and implementation are not as popular as general programming languages. To further the adoption of provenance awareness it makes sense to take the tooling towards general programming languages that developers use more widely. In this chapter, Cronista's design is extended again to enable automated approaches to instrument shared-knowledge model systems, where MDE code generation tools are not available or can not be extended to apply instrumentation.

6.1 Reducing Cronista's dependency on an MDE code generator for automating the model instrumentation

6.1.1 Motivation

In order to apply provenance awareness to a system, PRIME requires an SOA or actor-and-message design view of the system. The SOA design view is used to identify where provenance instrumentation should be applied to a system. Provenance instrumentation is carefully and selectively placed in a system, in order to collect enough of the right provenance data to satisfy a set of explanation requirements. Cronista's approach to provenance awareness is based on a shared-knowledge model, which reduces the design-time cost for applying provenance awareness to this type of system.

The provenance awareness approach demonstrated by Cronista is based on recording the provenance of a system's interactions with its shared-knowledge model. Ideally, this approach leverages a system's existing shared-knowledge model design to reduce the design-time cost for applying provenance awareness; using PRIME would require the SOA design view to be created, whereas Cronista would not.

A system where Cronista has been applied produces a history model, which contains a provenance graph documenting the evolution of the system's shared model. The provenance of the model is presented as a description of how the system's agents (threads of execution) performed a series of activities that used/created the states of the shared-knowledge model. The collected provenance can answer a facet of questions about a system's execution: for example, answering questions about how the shared-knowledge model evolved or the sequence of activities that occurred. In turn, these answers could then contribute to an understanding or explanation of a system's behaviour.

Managing the design-time cost for applying provenance awareness to a system has been an important aspect of this thesis. In part, the design-time cost can be reduced by enabling provenance awareness through using existing design artefacts; this cost saving depends on the availability of suitable artefacts, e.g. a shared-knowledge model. The more practical costs of implementing provenance awareness can be managed by taking advantage of Cronista's reusable components and its modular design, which only requires a small adapter to be created to fit other model implementations. Instrumentation of a model's codebase is automated with an MDE code generator, which saves a significant amount of manual developer effort by inserting provenance collection code into each

model object's accessor methods.

Cronista's implementation at this point requires MDE software development tools: specifically, the code generator to automate the model instrumentation for provenance awareness. When creating a new system with a known requirement for provenance awareness, this requirement for an MDE code generator can be considered when selecting the framework to build the system. However, this dependency on an extendable code generator for deploying the model instrumentation is a limitation for Cronista when provenance awareness needs to be retrofitted to an existing system. For example, two obvious situations would make Cronista difficult or impossible to use for retrofitting a system with provenance awareness. Firstly, the MDE framework used to create the system may have a code generator, but the code generator may not be extendable. Secondly, a system may already use the code generator in a custom way which conflicts with Cronista's instrumentation extensions. When an MDE framework has been used, but a code generator cannot be used to instrument the model's codebase, Cronista's design-time costs would increase; an alternative approach to automate the model instrumentation is needed.

6.1.2 Problem

Figure 6.1 shows Cronista's workflow for instrumenting a system model using a code generator. A developer takes the model design and passes it through the code generator, which is configured to use a set of *Model Root Classes* extended with Cronista's instrumentation code. The output from the code generator is a set of *Implemented Model Classes* containing the instrumentation code; these classes are added to the system's code base and are available for use by other *System Classes* that implement the agents and activities.

In the absence of a code generator component, a developer would need to add the instrumentation code to the implemented model classes manually. In the worst case, a developer must add a few lines of code to every accessor method in the implemented model classes. Some system model designs may allow a developer to reduce the number of lines that need manual editing. For example, some model subclasses could inherit their accessor methods; the parent methods would be instrumented once, and the code would propagate to the many subclasses. However, regardless of the model design, several lines of instrumentation code need scattering across multiple implemented model classes.

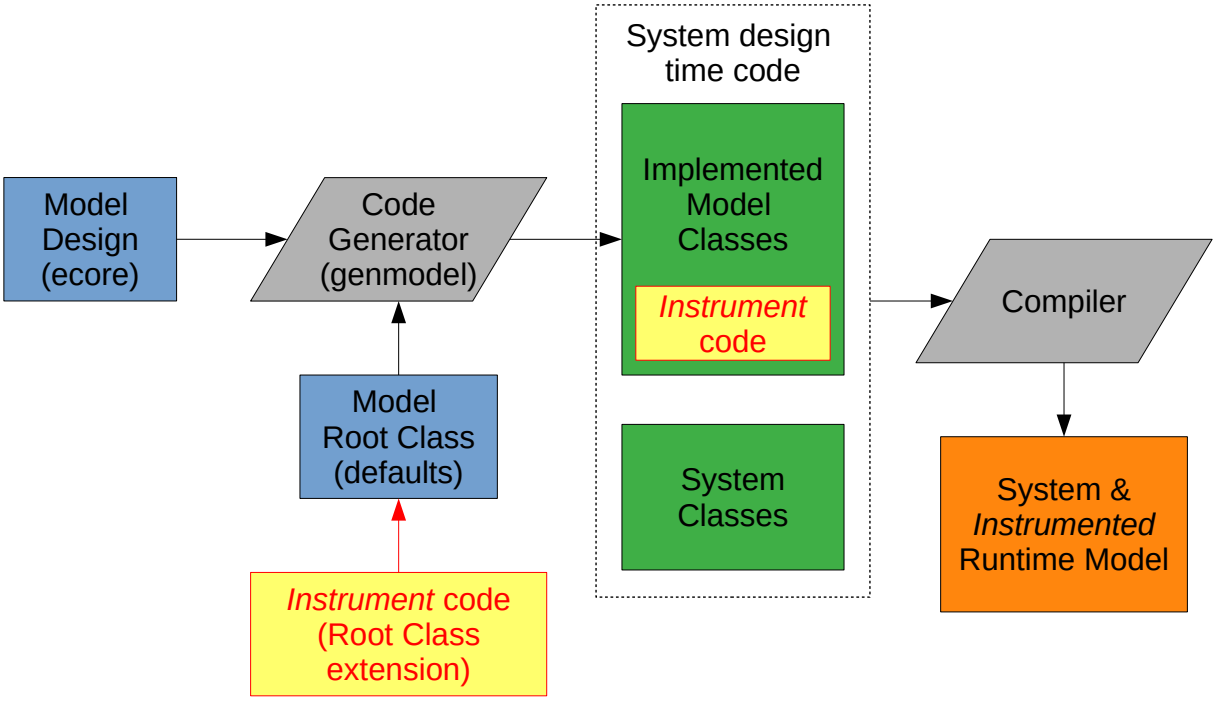

Figure 6.1: Workflow for EMF Code generator instrumentation

A developer who manually adds many lines of code may accidentally not add the code to some parts of the model, which could result in a failure to collect some elements of provenance information. The repetitive manual coding work is also prone to other human errors, potentially causing further problems and design-time costs. Overall, without using a code generator to instrument a system model, the design-time cost for applying Cronista increases, and with a large system model, the cost may not be acceptable.

6.1.3 Objective

The objective is to design an alternative approach to automating the deployment of Cronista's model instrumentation to a system's shared-knowledge model. An alternative approach should produce the same provenance as if a code generation method was used, and, by avoiding manual coding, should offer equivalent design-time costs to that of the MDE code generator.

The new automated model instrumentation approach could affect the runtime costs, and may have different execution overheads than the code generator approach. Cronista's provenance collection design seeks to not burden a system's execution in a way that negatively affects its intended function. Therefore any increase in runtime resources for provenance collection with the new approach will need to be managed.

Developing a new approach to automating the instrumentation of a system's shared-knowledge model should answer the following research questions.

LRQ1 *How can Cronista's model instrumentation be automatically applied to a shared-knowledge model without using an MDE code generator?*

LRQ2 *How do the design-time costs of the new automated approach to instrumenting a model compare to using a code generator?*

LRQ3 *How do the run-time costs of the new approach compare to the code generator approach?*

The new approach will also provide knowledge and answers to the thesis' research questions relating to design- and run-time costs.

RQ3 and sub-questions

RQ3 *What new knowledge of provenance awareness costs was discovered by the design and development of an artefact to answer RQ1?*

- **RQ3.1** *What are the design-time activities for using the new approach, compared with existing provenance awareness methodologies?*
- **RQ3.2** *What runtime overheads were seen on the systems used to evaluate the implementation, and how do they compare to the results published for provenance awareness?*

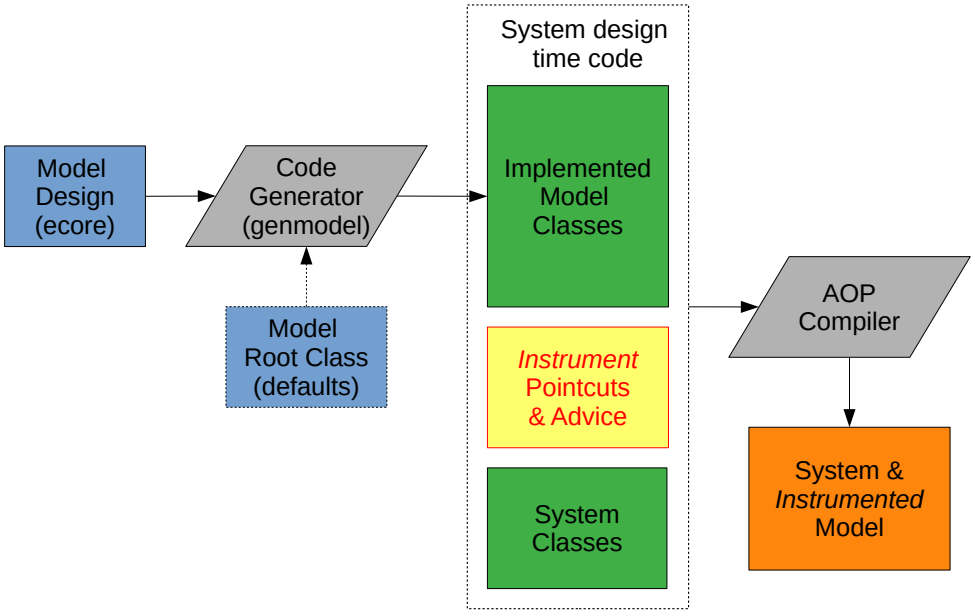

Figure 6.2: Workflow for EMF Aspect instrumentation

6.1.4 Design and development

In the provenance awareness background discussion (Section 2.5), the use of Aspect-Oriented Programming (AOP) had previously been identified as a technique for implementing provenance collection. Cronista's Observer components require code to be added in many places (e.g. across the system's shared-knowledge model), which is a clear case of a *cross-cutting concern* (Section 2.2). There is a reasonable argument that AOP could be used to apply Cronista's instrumentation to a system's shared-knowledge model, after an MDE code generator has produced the model code. AOP would be used to insert similar code for provenance instrumentation into the model code, like the code generator approach. The provenance instrumentation code would reflect the same shared-knowledge model information as the current code generation approach.

The code generator approach inserts instrumentation code during the code generation process, before the system code is compiled (Figure 6.2). However, with an AOP approach, the model code will be generated without instrumentation, and AOP will later modify the code during compilation (Figure 6.1). Pointcuts and advice are used with an AOP compiler to apply the model instruments to the system's shared-knowledge model at compile-time; the resulting system includes an instrumented model.

Cronista's modular architecture has previously separated concerns for handling different model implementations with a common *Model Object/Attribute* representation (Section 5.2.5). The previous

code generator instrumentation for *CDOObject* and *EObject* implementations perform the extraction and translation of their respective model representations in the model accessor methods. In part, Cronista's instrumentation design was an organic extension of the encapsulation patterns (accessor methods) produced by the code generator for each model type. As such, each model implementation type was considered stand-alone; each type requires a one-off reusable set of instruments to be created. For the AOP approach, this design could be improved by adding a separation between the aspects that define where to add instrumentation, and the inspection process that extracts model element information.

In Figure 6.3, the *Aspect Instrumentation Pointcuts & Advice* contains Cronista's model instrumentation. The pointcuts and advice indicate to the AOP compiler the model code modifications needed to apply Cronista's instrumentation. The pointcuts define code patterns to identify where model accesses (accessor methods) occur in the model code. Advice is the instrumentation code the compiler will insert at points matching a pointcut; a pointcut and an advice code work as a pair.

It would be possible to have the advice code extract the required model information and produce the generalised *Model Object/Attribute* representation for the *ModelAccessBuilder*. However, this would bind the advice tightly to a specific model implementation; a developer would have to be comfortable writing aspect advice code to create instrumentation for different model implementations.

To reduce the need for instrumentations being written using aspect code (pointcuts and advice) for different model implementations, an *Inspector* module could be used to separate the concerns between identifying points in code where model access occurs, and the process of *inspection* of the access: the code for inspection would exist outside of the Aspect. The *Inspector* contains all the code required to interpret the system model and produce generalised *Model Object/Attribute* representations for the *ModelAccessBuilder*. Thus, a more generalised aspect advice code could be written for use with pointcuts; the advice code passes the model element accessed at the join point detected by a pointcut to the inspector.

AOP for Provenance Collection from an EMF Model

Defining the pointcuts for an EMF model 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 may instead directly

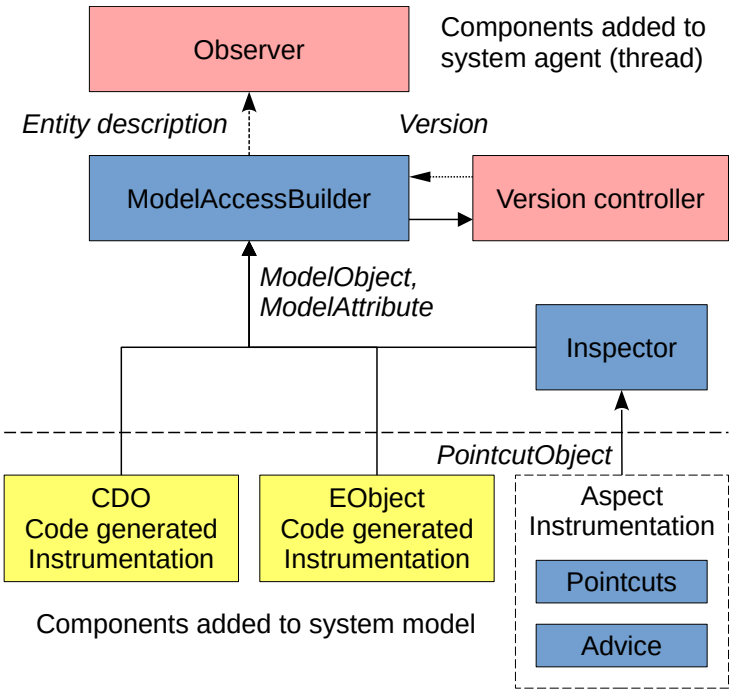

Figure 6.3: Cronista's model instrumentation architecture expanded to include an Inspector for use with Aspect Instrumentation. The Inspector is passed a Pointcut Object representing a detected model access from the Aspect Instrumentation, and the Inspector produces a generalised (ModelObject, ModelAttribute) pair for the ModelAccessBuilder.

manipulate the underlying storage of the class. Instead, to consistently capture EMF model feature accesses it is best to use EMF's *feature delegation* capabilities: 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. These methods from dynamic feature delegation are identifiable for pointcuts.

In a large system, it is quite likely that we will have not only one model but rather several. For example, the Eclipse IDE itself already contains several models. The compiler does not need to instrument every EMF model in the system's code, but rather only the shared-knowledge model. To distinguish the shared-knowledge model from any other models, *code annotations* are also required to assist the pointcut pattern matching. A pointcut pattern can be designed to match only the model accessors where an annotation is present: a developer will be required to add annotations to the shared-knowledge model code to distinguish it from all other models in the system's codebase.

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

Listing 6.1: EMF dynamic feature delegation pointcuts

6.1.5 Demonstration

To demonstrate Cronista's AOP instrumentation of an EMF model, the section presents some code examples and an experiment with the Traffic Manager system seen in previous demonstrations.

Code examples: Cronista's EMF Aspects

Cronista provides the `ObserveModelEMFeDynamic` annotation for a developer to apply to the model classes to be instrumented. A developer needs to add this annotation to the model classes generated by EMF. The annotation can either be applied manually[1] or with EMF dynamic templates. This provides an option to omit parts of a system's shared-knowledge model from the provenance collection by not applying the annotation: this was not previously possible with the code generator approach that applied instrumentation through inheritance. Excluding busy or irrelevant parts of a system's shared-knowledge model would reduce the runtime resource costs incurred for provenance collection.

A set of robust pointcuts was added to Cronista, as shown in Listing 6.1. 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 level of abstraction, and that the advice runs at the same point and produces the same model provenance data.

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 values). Listing 6.2 shows the advice to be inserted at those points in the system. The advice protects from recursive inspections of model access with the `observe` flag. Recursive inspections may occur when either the Inspector calls the same model accessors to extract information, or when an attribute is derived from another model

[1] EMF code generator preserves the annotations between code generations.

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

Listing 6.2: Excerpt of AspectJ advice for provenance collection from a system model created using EMF

object (e.g. via a reference): these could also produce subsequent model accesses on another part of the model, triggering further inspections. Advice should only invoke the `Inspector` once per pointcut. There are 3 separate pointcuts: after a get, before a set, and after a set. Inspector calls are performed using a *try, catch, finally* pattern to ensure graceful handling of errors in provenance collection; a system should function correctly if provenance collection fails for unexpected reasons.

Finally, an `Inspector` class provides methods to collect all necessary provenance information for model accesses. This includes extracting identifiers for the accessed model element (e.g. CDO IDs, feature IDs). These methods are reusable across runtime models which use the same persistence framework, e.g. CDO. The existing model access builder component performs the maintenance of version numbers for the various model elements if required.

Workflow for EMF model instrumentation using Aspects

Applying Cronista's instrumentation to a shared-knowledge model with AOP does change the design time workflow for applying provenance awareness, as expected. Previously with the code generator (Figure 6.1), the implemented model classes are created with instrumentation applied, and the provenance instrumentation code is a visible addition to the model code.

Using Cronista's current AOP approach, a developer manually annotates each model class produced by the code generator to include it in the scope of provenance collection. As stated in the design section above, the annotations could be automatically added by EMF's code generator; however, this option was not explored as this design cycle's objective is to reduce the EMF code generator dependencies.

The EMF code generator can preserve manually-applied model class annotations when the model code is regenerated after a developer changes a model. There are some exceptions to this: for example, a new model class requires annotation when generated for the first time. If a developer

deletes a file containing the model class code and then regenerates it, a new file will be created without the previous annotations. It must be noted that other MDE code generators may not preserve existing annotations as the EMF code generator does.

Traffic Manager experiment: comparison of AOP and code generator approaches

The Traffic Manager experiment seen in chapter 5 is used again in this demonstration. This system can autonomously adapt its runtime behaviour, using an EMF-based shared-knowledge (runtime) model to control the traffic lights on a junction in a simulation. The original version used the code generator approach in Section 5.1.4 to assist with explaining the system's behaviour with provenance graph queries.

In this version of the experiment, provenance is collected from the Traffic Manager using AOP and EMF code generator approaches. The resulting provenance graphs are compared, ensuring that the new AOP approach collects the same provenance information as the EMF code generator version. 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 used in Section 5.1.4 was introduced, and the same queries on the provenance graph were conducted to find the root cause.

The runtime metrics of the Traffic Manager with each approach were collected and compared to evaluate changes in runtime provenance collection overheads. Each Traffic Manager experiment configuration was executed 10 times to provide average and standard deviations for runtime resources. Two configurations of the AOP-based observer were used: one which collected the same information as the original EMF-based observer, and one which collected a partial provenance graph containing only the information needed for the 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 the Traffic Manager was run to set a baseline to compare all the provenance collection approaches.

These different Traffic Manager versions were labelled as follows:

- **NoProv** code and model unchanged, the resource baseline.
- **CodeGen** code and full model instrumented, using a code generator.
- **AspFull** code and full model instrumented, using AOP.

- **AspPart** code and part of the model instrumented, using AOP.

The provenance graphs produced by each Traffic Manager version were verified using a Gremlin graph query. This graph query checks for a known graph pattern (shown later in Figure 6.4) which can identify the injected fault in the system. Thus, each modified Traffic Manager system was run with and without the injected fault, to confirm provenance collection was working.

Experiment conditions The Traffic Manager was run while measuring system and query execution times, memory consumption, and network input/output (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 had been completed, the curator had processed all messages from the observer, and the history model had store shut down.

Results

The experiments with the Traffic Manager system and different Cronista configurations provided the following observations and metric results.

Design-time costs Cronista required some model technology-specific components to be implemented to enable the use of an aspect-oriented approach (blue boxes in Figure 6.3 on page 217). The Inspector and aspect code implementations extract model element information from a model implementation for the model access builder. As seen in the demonstration (Section 5.2.5), a set of common components can be reused between different model implementations; e.g. *model access builder* and *observer* are based on generalised model element descriptions. For a developer seeking to apply provenance awareness with Cronista to a new model implementation, their effort is reduced to a one-off implementation of adapters to couple the model representations with the model element descriptions.

A developer needs to manually apply annotations to the model code after running the model code generator to use the demonstrated aspect approach to instrument a model. Applying the annotations manually per class enables full control of the model instrument deployment; specific parts of the model can be targeted for provenance collection, to reduce the amount of provenance

		200 ticks		1000 ticks		5000 ticks	
Metric	Version	Mean	SD	Mean	SD	Mean	SD
TM time	NoProv	33.96s	0.32s	154.23s	0.25s	754.93s	0.25s
	CodeGen	36.60s	0.27s	156.73s	0.43s	757.59s	1.59s
	AspFull	36.64s	0.42s	156.67s	0.23s	757.20s	0.46s
	AspPart	36.42s	0.23s	156.67s	0.26s	757.00s	0.42s
TM memory	NoProv	11.26MiB	0.03MiB	11.86MiB	0.07MiB	14.98MiB	0.18MiB
	CodeGen	15.47MiB	0.09MiB	20.28MiB	0.27MiB	37.44MiB	0.89MiB
	AspFull	16.27MiB	0.08MiB	21.39MiB	0.32MiB	48.97MiB	2.75MiB
	AspPart	15.92MiB	0.07MiB	20.90MiB	0.24MiB	47.55MiB	2.67MiB
HMS memory	NoProv	754.29MiB	13.31MiB	755.57MiB	8.13MiB	756.29MiB	6.73MiB
	CodeGen	774.53MiB	9.37MiB	893.60MiB	7.84MiB	928.44MiB	11.78MiB
	AspFull	770.57MiB	8.61MiB	895.13MiB	8.49MiB	941.96MiB	10.56MiB
	AspPart	769.71MiB	13.90MiB	896.14MiB	11.66MiB	926.11MiB	12.65MiB
HMS net IO	NoProv	0.01MB	0.00MB	0.01MB	0.00MB	0.01MB	0.00MB
	CodeGen	5.51MB	0.12MB	28.29MB	0.49MB	101.40MB	12.62MB
	AspFull	5.78MB	0.15MB	30.15MB	0.77MB	150.78MB	13.23MB
	AspPart	3.94MB	0.15MB	20.94MB	0.55MB	110.06MB	6.21MB

Table 6.1: Means and standard deviations of execution times, maximum memory usages, and network I/O for TM, over 10 simulations across 200/1000/5000 ticks.

data collected. Annotating the model is a one-off effort, as EMF preserves the annotations when the code is regenerated. However, the previous Traffic Manager modifications for agent and activity provenance collection were unchanged; using either AOP or a code generator is possible with the same agent and activity markup.

Manually applying the annotations to the model classes involves a small increase in developer design time effort, which is probably less than manually instrumenting the model methods in most cases. Nevertheless, even a slight increase in the manual effort to instrument a model could be a limitation with some large system models. If code generation of a large model produces too many classes, it may not be practical for developers to annotate the classes by hand. However, applying the annotation using inheritance to cover all model classes may be possible; aspects applied via inheritance would more closely match the original code generator approach which applies instruments to an entire model.

Runtime costs Table 6.1 shows the metrics collected from the experiments where the Traffic Manager (TM) was run for progressively longer simulations (measured in ticks).

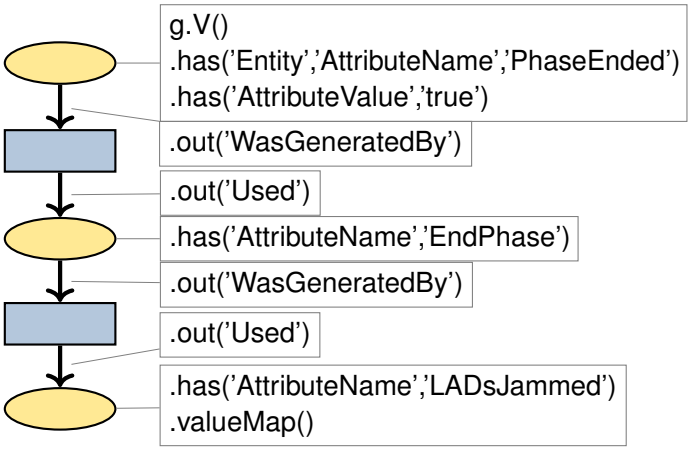

Figure 6.4: Gremlin query, find the cause of the PhaseEnded

The maximum amount of memory consumed increased 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. CodeGen uses less than both AspPart and AspFull, and AspPart is showing a slight reduction over AspFull. Finally, the execution times in Table 6.1 show that all the provenance collection approaches take slightly longer than NoProv (NoProv averaged 754.93s TM time, CodeGen averaged the longest TM time 757.59s): part of this comes from the processing and shutdown of the curation process, which lags behind the traffic control processes slightly.

Query results A Gremlin query can identify the fault and return the expected patterns, providing confidence that the new approach has parity with the original. The query used to verify the provenance collected from CodeGen, AspFull and AspPart all returned the expected subgraph (Figure 6.4), confirming the provenance graphs were valid and contained the required data.

Analysis of the query results for each Traffic Manager with and without the fault correctly returned the `LADsJammed` count for each `PhaseEnded` occurrence. For example, queries on a faulty Traffic Manager presented `LADsJammed` counts above 4, which exceeded the number of LADs connected to the system. Similarly, a working system did not have counts above 4 (the maximum number of LADs that could be reported as jammed), and the system also did not end phases as often.

6.1.6 Evaluation

The objective has been to automate the deployment of Cronista's model instruments to an MDE-developed shared-knowledge model, without using a code generator. This section will present the answers to the local research questions, and then discuss the findings.

Answers to the research questions

The research questions asked how the automation provided by an MDE code generator could be replaced, and what new cost implications an alternative approach introduced. The development of the AOP approach provided answers to the questions.

LRQ1

How can Cronista's model instrumentation be automatically applied to a shared-knowledge model without using an MDE code generator?

LRQ1 answer Aspect-oriented programming (AOP) techniques were investigated as an alternative approach to automating the application of model instruments. AOP is one approach for automating the handling of a cross-cutting concern; in this case, provenance collection from a shared-knowledge model. However, AOP can be used to reproduce some of the automation provided by the EMF code generator; this requires a set of pointcuts that match the specific locations where the code generator added instrumentation code.

When using the code generator approach, a developer controls which models in a system are instrumented by managing the configuration of the code generator. Any model passed into the EMF code generator configured for instrumentation is fully instrumented; currently, the generator's configuration does not have any options to exclude parts of a model.

AOP applies the instrumentation to all the code passed into a compiler; all matched pointcuts are treated with the advice containing the instrumentation. Thus, if the code being compiled contains more than one model, there is a need to differentiate between the models requiring provenance collection, and those that do not. In the demonstrated approach, the models to be instrumented are specified with a code annotation, and the pointcuts match the annotation and code generator patterns in the model. A side effect of the need for annotations is that provenance instrumentation can now be selectively applied to model parts.

LRQ2

How do the design-time costs of the new automated approach to instrumenting a model compare to using a code generator?

LRQ2 answer The developer's workflow for applying instrumentation to a system's shared-knowledge model using AOP has changed compared with the code generator workflow. The change in workflow gives a developer more control over where instrumentation is applied to a system's model. Therefore, an increase in control can be gained if the increase in manual effort is accepted. A developer manually applies annotations to the generated model code to identify the model (classes) to be instrumented; this is a single annotation per model class. The granularity of the pointcuts and annotations could be increased for further precision of the instrumentation control. For example, pointcuts could look for annotations applied to methods instead of classes, in exchange for requiring multiple annotations per class and increasing the effort involved.

Using AOP or an MDE code generator reduces the manual coding effort for model instrumentation through automation. Where possible, a code generator approach presents the lowest design-time cost; a one-off setup of the code generator fully instruments a model. AOP is slightly more expensive when fully instrumenting a model, because of the need for annotations on every model class, and these may need to be re-applied if lost on later code generation runs. Thus, AOP would only be a preferred approach if a code generator cannot be used, or if additional control of the automated instrumentation is desirable for controlling the provenance awareness of a system's shard-knowledge model e.g. omitting part of the model that is frequently accessed that would create a flood of provenance data.

LRQ3

How do the run-time costs of the new approach compare to the code generator approach?

LRQ3 answer Execution time for the AOP and code generator approaches were reasonably similar (within 1 second for a 5000-tick simulation taking approximately 757 seconds to execute). The Traffic Manager's memory footprint slightly increased with instrumentation applied using AOP instead of the code generator. While external system noise probably accounted for some deviations in the metrics,

the results imply that code generation may result in a system which is slightly more resource-efficient at runtime. AOP probably introduces some overheads into the compiled system binary, whereas code generation does not. Another explanation could be that a compiler can better optimise code where instrumentation is applied by code generation; AOP may not produce as many repeating code patterns as the code generator and, thus, cannot be uniformly optimised. The exploration into the finer details of AOP and code generator efficiency is beyond the scope of this research thesis. AOP and code generation in the experiment present reasonably similar runtime metrics; when available, a code generator approach is preferable for low runtime costs for fully instrumented models.

The results for full and partial model instrumentation with AOP indicate that some runtime resource costs could be lowered by excluding parts of the model from provenance collection. This is not an entirely unexpected outcome, but the reduction in runtime resources for Traffic Manager CPU and memory is small (see Table 6.1). The most significant reduction for excluding parts of a model is seen in the network data to the History Model Store, where the provenance data is verbose because of duplicate information. The amount of resources that could be saved is difficult to predict, and likely to be specific to each system since the provenance data production rate is unique to a system. If part of a system's model could be identified as producing a large amount of unnecessary provenance data, the ability to exclude it from provenance collection with the AOP approach would be beneficial.

Discussion

The objective was to find an alternative approach to code generation for automating the process of instrumenting a model, at a comparable cost. Broadly, the presented approach using AOP achieves this objective, with comparable - though not reduced - costs for design or runtime. From the metric result observed in the experiment with the Traffic Manager (Section 6.1.5), no clear argument can be made for using AOP as a cost-saving measure. However, AOP is a suitable alternative when code generators are not available, or when a side effect of the AOP approach is desirable (e.g. partial model instrumentation).

The design-time costs for applying annotations for AOP could be managed or reduced; improvements over the approach demonstrated may be possible. Future work could explore other AOP patterns and techniques for model instrumentation or could seek to use AOP for agent and activity instrumentation. While the code generator in EMF could be used to apply the AOP annotations to the model code during generation, it would only be useful in rare situations. In this edge case, the

EMF code generator cannot apply the instrumentation code directly to accessor methods, but it can annotate model classes. Overall, using the EMF code generator presents a simple and effective approach to instrumenting a model developed using MDE.

The AOP approach has not been tested with an MDE framework other than EMF. Conceptually, any Java-based MDE framework that produces a model implemented as Java classes would be compatible with the AOP approach. MDE code generation results in consistent (predictable) code patterns, which facilitates the use of AOP, since pointcuts and advice depend on code pattern matching to work effectively. However, future work beyond this thesis is needed to demonstrate the use of AOP to deploy Cronista's model instruments against a model created using another MDE framework.

6.2 Enabling Cronista for use with a non-MDE shared-knowledge model (AI-Checkers)

6.2.1 Motivation

In the previous design and development section, the requirement for an MDE code generator to automate model instrumentation was reduced. The MDE code generator served an important role in automating the process of instrumenting a model, enabling reliable and robust provenance collection with less manual developer effort. However, this requirement limited the use of Cronista to the relatively small range and variety of systems created using MDE tools, compared to those produced using more traditional techniques with general-purpose programming languages (GPLs).

It would be desirable to test Cronista's approach to provenance collection against other systems (i.e. systems not developed using MDE). To this end, Cronista's AOP approach could be adapted to enable automation of the provenance instrumentation for non-MDE shared-knowledge model systems. Generalising the automated model instrumentation for provenance collection as a set of reusable AOP-based modules could enable Cronista's approach to be used with non-MDE-developed systems.

At this point in the thesis, Cronista's Observer components have been implemented in Java using MDE and AspectJ to automate the instrumentation of a shared-knowledge model created with EMF model. A further reduction of Cronista's dependency on MDE (EMF) features is needed to enable testing against a non-MDE share-knowledge model system. This design cycle could extend Cronista's

AOP modules to release some of the remaining MDE dependencies, and could potentially generalise Cronista to systems with a shared-knowledge model developed with Java. The experiment alone will not confirm that Cronista generalises to all non-MDE systems. However, it could indicate how well Cronista may generalise to other Java systems with a non-MDE-developed shared-knowledge model. The experiment findings, combined with Cronista's emerging architecture and separation of concerns, may drive future work. This work could seek to re-implement Cronista using other object-oriented languages and explore provenance collection from different systems, domains, and models.

6.2.2 Problem

Cronista's current implementations for *EObjects* and *CDOObjects* both exploit a common encapsulation pattern in EMF. There is an assumption that other modelling frameworks and code generators for models would produce similar encapsulation patterns or model access interception points that could be extended to include Cronista's instrumentation. Thus, through Cronista's modular design, adapters could allow additional MDE frameworks to be used with minimal one-off design-time costs. The adapter could be reused across systems created using the same MDE framework.

The designs of manually-created shared-knowledge models are likely to be less consistent than those created using an MDE framework's code generator. A developer might follow some 'best practices' while creating a system's shared-knowledge model, but there are no guarantees that a consistent design will be enforced. A developer's code is not usually as consistent or predictable as generated code, either within a system's codebase, or across several systems. This lack of consistency between systems would probably result in Cronista having many bespoke model instrumentation modules, each with little re-usability between systems. Design-time cost savings for using Cronista would be reduced, with each system requiring some additional tooling to be created for Cronista.

Cronista has other requirements beyond simply using the code generator or AOP to save costs at design time. Cronista's provenance records for model accesses require model objects to have identifiers: these, and other modelling framework features, were provided by EMF. Another requirement is model versioning. In previous design and development cycles, the lack of model versioning facilities needed to be overcome. A simple versioning module was added to Cronista, which tracked model versions using the ID system. A non-MDE shared-knowledge model may lack or only have limited ID/versioning systems, requiring Cronista to provide additional components to supplement features

previously provided by EMF.

6.2.3 Objective

The main objective of this iteration is to design an approach to creating AOP model instrumentation that can be used with a shared-knowledge model system which has not been developed using MDE. The approach needs to target common code patterns used by developers in the target language, to enable the provenance instrumentation to be applied using automation.

As Cronista has been developed using Java, only Java implementations of shared-knowledge models will be considered. Ideally, Cronista's instrumentation should target a low-level code pattern in Java that could be found in many systems created with Java. This would increase the re-usability of the approach between systems, without requiring them to have some specific architectural features in their shared-knowledge model implementation.

The investigation into an approach to achieve this objective should provide answers to the following local research questions:

LRQ1 *How can Cronista's automated AOP model instrumentation be extended to shared-knowledge model systems not developed using MDE?*

LRQ2 *What are the design-time cost implications for using Cronista with a non-MDE shared-knowledge model system?*

LRQ3 *What are the additional overheads on runtime resources when using Cronista on a non-MDE system?*

Outcomes from answering the local research question will contribute towards answering the following thesis research questions:

RQ3 and sub-questions

RQ3 *What new knowledge of provenance awareness costs was discovered by the design and development of an artefact to answer RQ1?*

- **RQ3.1** *What are the design-time activities for using the new approach, compared with existing provenance awareness methodologies?*
- **RQ3.2** *What runtime overheads were seen on the systems used to evaluate the implementation, and how do they compare to the results published for provenance awareness?*

6.2.4 Design and development

Cronista's instrumentation is placed at the point of model access, between the system and the model element being accessed. The previous Section 6.1 shows that a consistent structure is important for identifying the boundary between system code and model code. Modelling tools such as EMF provide code generators capable of producing model implementations with a consistent structure that separates them 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, such as the JavaBeans API [105].

Such consistent code patterns and separation between the system's logic and model code can help a system developer automate the instrumentation of a model with AOP. However, the boundary between the system logic and model could also be identified by applying code annotations; this is an alternative to pattern-matching the code with an *ad hoc* pointcut.

To create an automated model instrumentation component for Cronista, the model element access, correlation and versioning problems need solving for the specific non-MDE model implementations. The following approach can be used to design suitable pointcuts, advice, and a Cronista Inspector for a non-MDE model implementation:

Model element access The pointcut for EMF models required annotating the model classes to separate them from all other system classes, and non-MDE models are likely to require similar annotation. Pattern matching can be used with EMF models to identify accessor methods, e.g.

*eDynamic** methods in a model object. For non-MDE-developed systems, a similar approach could require annotation of the model classes, whose accessor methods could follow a naming convention for easier matching. However, these accessor methods may not necessarily follow a strict convention: for example, they may not all start with 'get' or 'set'. When accessor method names do not have consistent patterns, annotations can be used to identify the methods interacting with Model Attributes. Pointcuts can be designed to match two annotations: a "model object" class annotation (i.e. "contains model attributes"), and a method annotation (i.e. "interacts with model attributes").

Finally, in both cases, an assumption could be made that all attributes within a model object are model attributes. When some attributes are not part of the model, a further annotation could be applied to exclude them from provenance collection. This approach could be described as a *black-listing* approach, where all are included by default and excluded if listed. The inverse approach of *white-listing* is also possible, requiring annotations on the attributes to include in the provenance collection. Depending on the system model implementation, one of these approaches to include/exclude attributes by default could be more suitable than the other (discussed further in the demonstration in Section 6.2.5).

Model element correlation The advice inserted by the pointcut should pass the model object containing the invoked accessor method to an *Inspector* implementation. The *Inspector* performs the model element correlation processes discussed in Section 4.1, where a model element is a uniquely identifiable (model object, model attribute) pair. The *Inspector* determines which model element is being accessed and creates the appropriate ID to identify future accesses to the same model element.

For non-MDE-developed models, an Inspector will extract an ID from a Model Object and an ID for the Model Attribute, in order to identify the Model Element. The system could contain IDs to identify parts of its shared-knowledge model as part of its intended function. However, a system may not have IDs or the IDs might not be publicly exposed (i.e. inaccessible for provenance collection). A developer may need to look at a system's model code for suitable IDs: these could come from their implementation details. For example, if a framework was used to create a model, it may have a unique ID scheme like EMF, which provided IDs for the *EObjects*.

Model element versioning Cronista currently includes a basic versioning system that can version in-memory model objects and utilise an external source for storage versioning (e.g. CDO versioning). A developer can extend the Inspector to extract versioning information from their non-MDE model if versioning is present. If a model lacks versioning or its versioning is not sufficiently granular, Cronista's versioning module may be used.

6.2.5 Demonstration

To demonstrate the use of Cronista with non-MDE developed systems, this section will present examples of designing model instrumentation for such systems. Since a non-MDE model can be implemented in many different ways, requiring different approaches to the instrumentation, two examples are provided. The first example assumes that the model code follows the JavaBeans conventions. The second example, by contrast, assumes only that the model is a graph of Plain Old Java Objects (POJOs): this approach to model implementation can result in very unstructured code that requires pointcuts with a low-level approach (i.e. a developer creates a model with no consistent or common code patterns for pointcuts to target, so that patterns within the Java Object implementation need to be targeted).

After investigation of these examples, Cronista is experimentally applied to an AI-Checkers (AIC) game. This experiment presents observations of applying Cronista to a non-MDE system which was not designed with provenance collection in mind. The runtime metrics of the AIC system with and without Cronista are collected for comparison. Finally, observations of provenance querying for exploring the played games are presented.

Examples of non-MDE model instrumentation

AOP model instrumentation designs are demonstrated for JavaBeans and Plain Old Java Objects (POJOs). The model element access point for JavaBeans demonstrates pointcuts where annotations and pattern matching are used, whereas the POJO demonstration shows pointcuts using only annotations. The JavaBeans and POJO models share a common correlation and versioning approach, described later in this section.

Model element access: JavaBeans The JavaBeans API specification [105] seeks to create reusable software components by prescribing a common approach to structure objects. As such,

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

Listing 6.3: JavaBeans getter/setter pointcuts

```
1  @ObserveModel
2  public class ModelObject {
3    private String modelAttribute;
4
5    public String getModelAttribute() {
6      return modelAttribute;
7    }
8
9    public void setModelAttribute(String newValue) {
10     this.modelAttribute = newValue;
11   }
12 }
```

Listing 6.4: Java-bean example of an annotated Model Object

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. `getModelAttribute()`) can be used to extract the name of the Model Attribute, 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.

JavaBeans pointcuts can use code pattern matching (for accessor methods) with a class-level (model object) annotation. Listing 6.3 shows example AspectJ pointcuts for intercepting the execution of the accessor and mutator methods of a JavaBeans Model Object annotated with *@ObserveModel*. Listing 6.4 shows an example of a JavaBeans model object implementation that has been annotated for provenance collection with the pointcuts in Listing 6.3.

Model element access: POJO 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

```
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 6.5: Java object field read/write pointcuts

fields, using pointcuts such as those in Listing 6.5.

Listing 6.5 provides an example of annotation-based pointcuts for use with a POJO model implementation. 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 could 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).

In Listing 6.6, a model object class has been annotated for provenance collection using the pointcuts in Listing 6.5. Getting and setting the value of *modelAttribute1* with the accessor methods (Lines 9 and 15) is recorded to the provenance graph when in an activity scope. However, the interactions with *modelAttribute2* are never recorded to provenance (*@HideMe* excludes it from pointcuts). For example, *modelAttribute2* is assigned the value of *modelAttribute1* on line 10. This assignment occurs within an *@ObserverMethod* accessor, on an *@ObserverModel* class; it would be recorded to provenance if *@HideMe* was not applied to *modelAttribute2*. Furthermore, *modelAttribute2* is a public field and could be accessed by other methods with *@ObserverModel* annotations: these accesses would also be recorded to provenance if *@HideMe* was not applied to *modelAttribute2*.

```
1  @ObserveModel
2  public class ModelObject {
3    private String modelAttribute1;
4
5    @HideMe
6    public String modelAttribute2;
7
8    @ObserveMethod
9    public String getModelAttribute1() {
10     modelAttribute2 = modelAttribute1; // Excluded from Provenance
11     return modelAttribute1;
12   }
13
14   @ObserveMethod
15   public void setModelAttribute1(String newValue) {
16     this.modelAttribute1 = newValue;
17   }
18 }
```

Listing 6.6: POJO model example of an annotated Model Object

Model element correlation A system's model may include an ID scheme as part of its design: for example, each JavaBean in a model might have an ID field. In that case, a developer could have the Inspector extract the ID from the system model beans. However, the system's ID scheme would need to conform to the requirements that underpin the creation of entity IDs (discussed in Section 4.1.4, paragraph "Entity Identifiers"). When a system's model has no ID scheme, and only uses POJOs for the model, a developer faces a problem with identifying model objects. Solutions to identity management for model implementation can be found in MDE frameworks and related literature. The scope of this thesis research does not extend to designing a robust approach to managing persistent model object IDs for POJOs. However, a minimal solution for model element correlation in POJO models is required to proceed with the design cycle and experiment.

Cronista's *Inspector* for a POJO model can extract some information from the base implementation of a Java *Object* that is part of the shared-knowledge model. Experimentation and reviewing of the Oracle Java API documentation [87] revealed an existing Java feature that could be used to identify model objects:

Official Javadocs for *java.lang.Object*

hashcode(): "As much as is reasonably practical, the hashCode method defined by the *Object* class does return distinct integers for distinct objects. (The hashCode may or may not be implemented as some function of an object's memory address at some point in time.)" [88]

As an approximation, Cronista uses the value returned from a Java object's default *hashCode()* method to identify a model object. The default behaviour for *hashCode()* is to call *System.identityHashCode()*, which returns an integer related to an object instance, but not calculated from the object's state or value. Within the lifetime of a Java object, the *identityHashCode* of an object is a consistent identifier for an object. The value returned in most practical situations is different for different objects, although it is not guaranteed to be unique.

Official Javadocs for *java.lang.System*

identityHashCode(Object x): "Returns the same hash code for the given object as would be returned by the default method hashCode(), whether or not the given object's class overrides hashCode(). The hash code for the null reference is zero." [89]

Using *identityHashCode* to identify a model object requires maintaining the same object in memory for the same model object. An in-memory *Object* instance and its underlying model object must undergo the same lifecycle (create/read/write/delete) for their provenance to be equivalent. This means that an *Object* instance representing a model object must not be recreated (destroyed and replaced with a new instance) unless the model object is recreated in the context of the model.

Persisting an *Object* to disk and deserialising it will not preserve its *identityHashCode*. Furthermore, different executions of the system would also produce disjoint provenance graphs if relying on *identityHashCode*, which may or may not be acceptable to the user. The fragility and potential duplication of *identityHashCode* values could be a problem for some systems. Therefore, this approach should not be considered a 'general solution' for all POJO-based models. A better solution to these issues requires introducing some form of persistent identifier such as those provided by EMF/CDO.

Model element versioning Since plain Java objects lack versioning, Cronista falls back to a basic implementation of a version controller, which uses the above model element correlation approach. This basic implementation keeps a list of model elements with their storage versions held at 0 (memory-only versioning), and increments the in-memory version on every update. This simple approximation is sufficient to track the provenance of each execution of a single-thread plain Java system. For a more complex or long-running system, versioned model storage with thread-safety features and persistent object IDs would be more appropriate.

The experiment with AI-Checkers

AI-Checkers (AIC) is a Java-based AI that plays the classic Checkers game. The AI uses an alpha-beta tree search to decide its next move. The AI and game code were written by a third-party developer [14], in contrast to the Traffic Manager system created in this thesis to test Cronista's approach to provenance collection (Section 5.2). AIC does not have the design bias towards producing meaningful provenance for self-explanation that the Traffic Manager could have.

The selection process for the third-party system used in this experiment involved a search of GitHub.com [76] for a Java-based AI system with permissive licensing. Key criteria for selection were readability and understandability of project code: large complex code bases can be hard to explain, and shorter and clearly written code was preferable. An AI program was preferred over other types of systems, as it is more in keeping with the research theme of provenance collection for automated decision-making. Finally, board games like Checkers often require the code to maintain a model-like representation of the game state; this equates to a form of shared-knowledge model that could be observed.

AI-Checkers consists of four classes all in the same package, so they can freely interact with each other. The functionality of each of these classes is briefly explained, to give an idea of the system's structure.

The *Main* class contains the main method, which requests game setup information when the program is started: the number of AI players for the game, the time limit for the AI player to search for their next move, and which player moves first. A game is set up, and the main loop of the game is entered. The loop is exited when a game is completed (i.e. when a player cannot make a legal move). An iteration of the game loop has a player take a turn. In the case of an AI player, a turn consists of

planning a move (searching for the 'best move') and then applying the move to the game board. If a human were playing, a turn would consist of collecting the input move and then applying the move to the game board.

The *Game* class has a game board that is represented with a 2D integer array. Additional game information is also held, such as the current active player and the time limit for the turn. Methods are provided to interact with the game board, which performs checks against the game rules, such as validating whether a move (a slide or a jump) is legal.

The *Move* class represents the details of a move. Moves have start and end coordinates, and each piece taken is tracked in a list of coordinates for the taken pieces.

The *Heuristic* class has three settings for 'game difficulty': easy, medium and hard. The case study used only the default setting (hard). The class implements the AI player, whose strategy changes depending on the number of pieces on the board. Each strategy is a variation on moving toward the middle of the board. First, moves are scored, and the highest-scoring moves are selected. Tied moves are collected into an array, and a random element is selected to play.

Changes for repeatability 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 that the same best move would be selected during the multiple executions needed to control for system variability. The time limits for searching for the best moves were also removed, leaving only the search depth limit. Experimentation showed that even minor timing changes in the execution time of AIC could change a game's sequence of moves and outcome. The best move found varies with the amount of time permitted to search and score possible moves. These timing differences could be produced by I/O and other system processes which are hard to control. After running a selection of games with known seeds, some games entered endless searches for the best move. Therefore, only the seeds of the games that could successfully finish were used for the experiment.

Experiment conditions A seeded game was played in each execution of AIC, while measuring system and query execution times, memory consumption, and network input/output (as measured through the JanusGraph Docker container). These experiments were run under Debian 10 with a

Linux 4.19.0-17-amd64 kernel, using OpenJDK version 11.0.12 for the traffic manager with the default Java settings. The system hardware used an AMD Phenom II X4 970 Processor (3.5 GHz), 16GB RAM, and an SSD storage device. A background thread measured execution times and memory usage until the simulation had been completed, the Curator had processed all messages from the Observer, and the HMS had shut down.

Results

The results of the experiment with AIC are presented below. The design-time costs reflect the effort needed to apply Cronista. Unlike the previous Traffic Manager design-time costs, the effort included changing AIC's code beyond the application of the Cronista Observer, activity scopes, and model instruments. Runtime metrics and provenance query results follow after the design-time costs.

Design-time costs AIC had a simple design and a POJO model which closely resembled part of the problem domain: a representation of the game board. When designing the system, the creator of AIC did not consider collecting provenance for each game played. Thus, Cronista's provenance collection was retrofitted to AIC's code after its creation, and without consulting the original developer. As such, the retrofitting of Cronista required the original developer's code to be reviewed and interpreted as a sequence of activities that occurred during a game.

The first steps for applying Cronista involved applying some activity scopes and descriptions to AIC, in order to produce provenance of the activities that describe a game of Checkers being played. It was then necessary to refactor AIC to expose some relevant information to Cronista: some information was needed that was not part of the system's model but described something about the game being played. The refactoring was needed to overcome some of the limitations of AspectJ's Java field interception capabilities. Some concepts, like 'the number of best moves' found by the AI, were not explicitly modelled in the POJO representation: in this case, it was the size of an Array containing 'move' objects, which would not be directly accessible with Cronista's current POJO model pointcuts.

The general application of Cronista remained similar to that for previous experiments: i) integrate the Observer into the system, ii) delimit the high-level activities through activity scopes, iii) delimit model parts for observing, and iv) use the provenance graph to answer queries to feed explanations. Steps ii), iii) and iv) were iterated, slowly incrementing the amount of provenance collected. After

```
1  private static Game game;
2  private static Move move;
3  public static void main(String[] args) {
4      ...
5      if (player1iscomputer) {
6          try (ActivityScope scope1 = new ActivityScope("Player 1 plans")) {
7              move = cpu1.alphaBeta(game);
8          }
9          try (ActivityScope scope1 = new ActivityScope("Player 1 moves")) {
10             game.applyMove(move, game.board);
11         }
12         System.out.println("Player 1's move is ");
13         move.printMove();
14     } ... }
```

Listing 6.7: AIC: Activity Scopes for Player 1 planning and moving in a game turn

each iteration step, iv) was performed to check the quality of the provenance graph and to look for missing nodes and relationships. GraphExp [95] was used to inspect the provenance graph representation of the Checkers game being recorded. However, the Gremlin console [3] was also needed to analyse the graphs when it became too complex for visualisation. The steps used to create a provenance-enabled version of AIC are presented below.

First iteration The intention was to capture a Checkers game as a sequence of activities (e.g. move planning and movement) for each player (Listing 6.7), at a high level of abstraction. Method calls for move planning and execution were located in the system's main method, and were annotated with activity scopes covering both AI players. When defining the activity scopes, it was necessary to turn local variables such as `move` into object fields so that AspectJ could intercept them.

Variables passing information between two activity scopes needed refactoring to become fields, and therefore become part of the shared-knowledge model. This refactoring exposed the variable (and value) to a pointcut, which the Inspector identified as a model element that the Observer would create as an ENTITY connecting two ACTIVITIES. Other local variables such as `game` which contained the game board representation also needed to be refactored into fields.

Finally, string conversion methods (*toString()* in Java) were added to both `Move` (Listing 6.8) and `Game` (Figure 6.9). These methods created a textual representation of the information representing a 'move' and a 'game'. Cronista could then record these text representations in the provenance graph ENTITIES to provide human-readable representations of a move and a game (its board state).

```
1  gremlin> g.V().has('Entity', 'AttributeName', 'move').values('AttributeName','AttributeValue')
2  ==>move
3  ==>2,5 -> 4,3 pieces taken: (4,5) ,
```

Listing 6.8: AIC: move object value as String

```
1  gremlin> g.V().has('Entity', 'AttributeName', 'game').values('AttributeName','AttributeValue')
2  ==>Player: 1
3  [0, 1, 0, 1, 0, 1, 0, 1]
4  [0, 0, 1, 0, 1, 0, 1, 0]
5  [0, 0, 0, 0, 0, 1, 0, 1]
6  [0, 0, 1, 0, 1, 0, 0, 0]
7  [0, 0, 0, 2, 0, 0, 0, 0]
8  [2, 0, 0, 0, 0, 0, 2, 0]
9  [0, 2, 0, 2, 0, 2, 0, 2]
10 [2, 0, 2, 0, 2, 0, 2, 0]
```

Listing 6.9: AIC: game object value as String shows the board pieces.
0: empty square, 1: first player's pieces, 2: second player's pieces.

Second iteration Having established a provenance graph that can reproduce the state of the game over a sequence of turns in the order they occurred, the next iteration exposed details of the planning process that occurred in a turn. Move planning was based on a heuristic search process, which was likely to produce excessive amounts of provenance data. Therefore, the activity scope BESTMOVESELECTION was made to encompass only a high-level selection process in the code. The activity created entities for the BESTMOVE and BESTMOVEVALUE. This BESTMOVESELECTION activity occurred multiple times during a player's turn; a 'best move' was found for each depth of possible moves searched, up to the depth limit.

Third iteration Some concepts needed to be added to the system's model, such as the *number of equally best scoring moves found*, and the *chosen* move. The existing code collected the equally scoring 'best moves' in an array and randomly selected a move as a tie-breaker. As explained earlier, the number of best moves was a hidden property of the array, as was the random number used as an index to select one 'best move'. In order to be accessible to Cronista, this information required explicit Object field representations to be added to the shared-knowledge model. After this refactoring, Cronista could observe the system model and determine the quantity of 'best moves' found, and the selection of which move to play.

		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 6.2: 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.

Runtime costs Table 6.2 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 ended after similar durations with very little deviation (between 0.03s and 0.27s). Therefore, the changes to the system to have the AI players make deterministic decisions worked as intended. The time differences seen between NoProv and the aspect-based provenance collection (AspFull) are the added overhead of Cronista's Observer components. Table 6.2 shows a small increase in the total execution time of approximately 3.5s on average: this result is similar to using AOP with the Traffic Manager (Section 6.1.5, Table 6.1).

The metrics in Table 6.2 under NoProv conditions show that memory usage did not significantly change with the game duration for different game seeds. AspFull shows that 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 6.2 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 network I/O increased 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 that up to 15.32MiB of SD can occur even with an unused graph. Network I/O showed little to no change when replaying the same seed, which implies that the Observer was reporting similar data for each replay.

Querying provenance Applying Cronista to AIC involved an iterative development process, running frequent provenance queries with GraphExp and Gremlin to inspect provenance graph fragments. This was done to ensure the game's code was being correctly interpreted and that it followed the expected sequence of processes. A few turns of a game could be watched, and then provenance queries could be performed on the collected provenance to check the game states recorded in ENTITIES were correct, and the ACTIVITIES were connected in the expected order.

While applying Cronista, the provenance graph was reviewed to look for gaps in the chains of ENTITIES and ACTIVITIES. A gap in the chain could represent a gap in provenance knowledge (i.e. not knowing the source of something). Without a set of explanation requirements (as described in PrIME Section 2.5.2 and PLEAD in Section 2.5.3), no specific questions and answers could be expected. Collecting more detailed provenance was likely to enable more provenance questions to be answered, but this would have increased the costs (both design-time efforts to apply provenance-awareness, and runtime system resource usage). A developer might seek to achieve full coverage of the model and activities at a high level of abstraction (coarse-grained provenance), then break up the activities into smaller steps, with fewer ENTITIES involved (fine-grained provenance).

In the case of AIC, an assumed provenance question could be answered. The developer of AIC might ask for the provenance of any move an AI player makes; for example, was one move found to be the best, or was it randomly selected from a set of equally good moves? Figure 6.5 shows a Gremlin query which accessed the provenance graph produced from AIC to retrieve ENTITIES and ACTIVITIES that connect a `Move` back to the `NumberOfPossibleMoves` from which it was selected. Thus the number of alternative moves the AI Player found could be causally connected to the move played. As a developer of AIC, this query would tell us how often the move decision process ended with a random selection from a collection of equally scoring best moves. This information would indicate a need for further refinement of the decision-making process, by extending the search time or adding tie-breaking strategies.

6.2.6 Evaluation

This section presents the evaluation of the design and development cycle. The research questions are answered using the results and observations, followed by a discussion of the threats to validity. Finally, the section will close with a discussion of the findings.

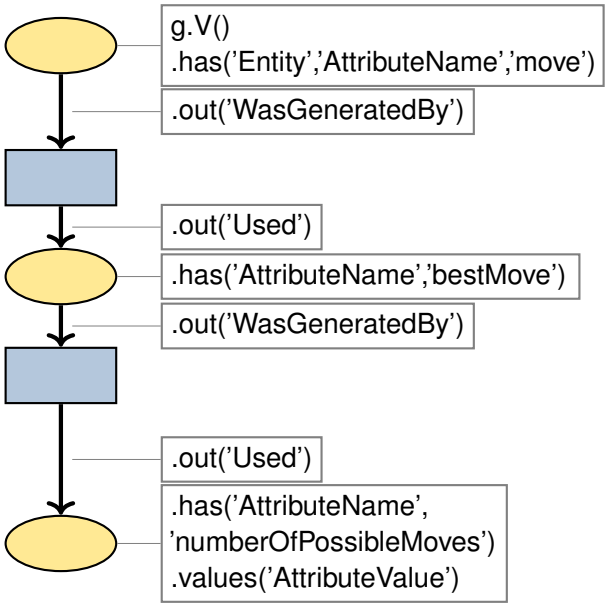

Figure 6.5: Gremlin graph query used to find number of best possible moves

Answers to research questions

Developing Cronista's AOP tools and applying them to AIC (a non-MDE-developed system) provided the knowledge to enable the research questions to be answered.

LRQ1

How can Cronista's automated AOP model instrumentation be extended to shared-knowledge model systems not developed using MDE?

LRQ1 answer Reusable AOP pointcuts can be designed for applying Cronista's instrumentation along the boundary between a system's logic and its shared-knowledge model. If a system's model is implemented using a consistent coding style, such as JavaBeans, the boundary can be found in the accessor methods of the model objects. However, when a system's model is implemented in a less structured ad-hoc style, this boundary can be more difficult to find.

Pointcuts can be designed to target and automatically instrument a system in either of the two above cases. To do this, the pointcuts require at least two reference points that will overlap where model interactions occur. These points can be a code pattern (e.g. based on naming conventions for accessor methods) plus a class annotation (indicating its instances are model objects), or can be a pair of a class annotation and a method annotation when a model has no coding conventions for accessor methods.

The design of the pointcuts is sensitive to the coding style used to implement a system and its shared-knowledge model. This is similar to instrumenting MDE framework models: each framework would require Cronista to have a specific set of model instruments, but pointcuts can be modularised for reuse between systems that have similar coding styles.

LRQ2

What are the design-time cost implications for using Cronista with a non-MDE shared-knowledge model system?

LRQ2 answer Cronista's reusable architecture and AOP automation could be used to collect provenance from a non-MDE-developed system. Model instrumentation requires a developer to apply small amounts of code across a system's codebase, as with an MDE-developed system. Using AOP, the pointcuts can use code pattern matching or annotations to automatically apply instrumentation, which would have been several lines of manually added imperative code. However, provenance collection requires a system to have features to support model element access, correlation, and versioning; MDE frameworks like EMF can provide these features. If a non-MDE system does not provide identifiers or versioning for its model elements, the design-time costs increase as a developer needs to add those features to the system, or to create modules allowing Cronista to provide them in a reusable manner.

While Cronista's reusable modular architecture and automation manage some design-time costs for implementing provenance collection, the use of a modelling framework like EMF also seems to reduce design-time costs for provenance collection. A modelling framework reduces design-time costs with model implementation features, as well as automating the model instrumentation process with a code generator. Having applied Cronista to AIC, and created some AOP model instruments to perform model element access, correlation, and versioning on POJO models, the benefit of the facilities like EMF's EObject IDs became quite apparent. Cronista's model object ID system that uses a Java *Object*'s *identityHashCode* method was sufficient for the AIC experiment, but it is not sufficiently tested or robust enough to recommend as a 'suitable solution' for POJO models in a production environment.

The AIC demonstration showed additional design-time costs that were unrelated to standard Cronista demands, such as adding Observers, activity scopes or model instrumentation. For example,

some of the information that enriched the provenance graph of a Checkers game existed in the system, but could not be collected with Cronista's AOP pointcuts for POJO models until the code was actively refactored. There are several reasons why AIC's code needed refactoring:

1. Cronista's pointcuts and/or AOP limitations could not reach the information available as a side effect of the code, e.g. an array size.
2. AIC's developer was unaware of provenance collection requirements.
3. AIC was not designed using MDE or creating a detailed shared-knowledge model containing more than the game board information.

Retrofitting Cronista to AIC exposed the design-time costs for designing a system with provenance collection as a concern. These are design-time costs that the PRIME/PLEAD methodologies could assist with, by establishing explanation requirements. A developer can then approach the system design knowing what information needs to be present in the shared-knowledge model. Similarly, without using PRIME or PLEAD to gather requirements, a developer could integrate Cronista while building the system. During the development of the system, developers could collect provenance in tests, and perform queries to evaluate the provenance graph descriptions of a system's execution. This approach to system development with integrated provenance collection would be an iterative process, as seen in the demonstration section (Section 6.2.5).

LRQ3

What are the additional overheads on runtime resources when using Cronista on a non-MDE system?

LRQ3 answer The results presented in Table 6.2 show the extent to which runtime resources increased (CPU time, memory, and provenance storage space).

Applying Cronista to a system with a small memory footprint like AIC appears to impose a significant memory increase; approximately 10MiB is added to AIC's 1MiB. Another perspective is the monetary costs this increase in memory might have on a system. If the system is running on a machine with more memory than required, and the amount of memory cannot be scaled down (for cost-savings), then provenance-awareness can be applied to a system for no additional monetary costs if the addition of provenance-awareness does not exceed the memory of the machine. This

perspective of increased monetary cost when machine resources are exceeded can also be applied to CPU resources: provenance-awareness may require more CPU resources, but the additional CPU resource required may not incur monetary costs.

In the experiment with AIC, it was necessary to disable a time limit that prevented an AI player from endlessly searching for moves. This was done to make the experiment repeatable, with a selection of seeded Checkers games that played out with identical moves and a predictable outcome. The results in Table 6.2 reflect the stability and repeatability that was achieved. All the game seeds appear to take an additional 3s, which is likely the processing of outstanding provenance messages and closing of Cronista; the actual time taken to play a game is not increased by 3s as this overhead seems consistent regardless of the total execution time. However, the timing metrics do not reflect the effect that timing changes caused by provenance collection could have on the AI player's movement selection process. A change in the execution time of the AI player's movement search and selection process could reduce the number of moves considered per second, which may effect the outcome of a game.

Threats to validity of the demonstration experiment

The classification from Feldt et al. [38] for threats to validity will be used in this section to consider the internal and external validity of the AIC experiment. Internal validity discusses whether the treatment caused the outcome, and external validity considers how well these results might be generalised.

Internal validity Integrating provenance collection into the AIC case study is dependent on a correct interpretation of the codebase and the original intentions of AIC's developer when writing the code. As such, the quality and detail of the provenance collected could be improved with a deeper knowledge of the system. Similarly, the chosen provenance graph query is based on an assumption that AIC's developer might consider the information useful, rather than having AIC's developer define an explanation requirement or question.

The in-memory database with JanusGraph was used, due to the same throughput limitations seen in the previous Traffic Manager demonstrations (Section 5.3.5). Cronista's minimal POJO instrumentation did not include a way to take snapshots of a POJO model, as this would require implementing features that can be seen in MDE frameworks and are outside the scope of this design iteration. As a result, the provenance of a game is recorded in a single time window, creating a single

large provenance graph, which could be successfully queried without a noticeable problem. However, this restriction on snapshotting could present a problem with games that run for longer. To handle this, future work could add a POJO model snapshot facility to Cronista, and then reproduce the AIC experiment with multiple time windows.

The JavaBeans AOP approach was not empirically evaluated in the demonstration. In part, an assumption was made that their likeness to an EMF model is a sufficient comparison. JavaBeans would have similar limitations to plain Java objects, e.g. lack of versioning and identification. Nevertheless, the JavaBeans AOP approach to applying provenance collection using AOP might tread a middle ground between modelling tools and software engineering practices of a wider audience.

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 application is created to allow end users 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) may be too demanding for typical non-technical end users. The experiment demonstrated collecting provenance and using a provenance query to answer a question with the collected provenance: the practical value of the AICs provenance graph or the question answered was not fully considered (i.e. there were no explicit explanation requirements for the experiment). AIC's developer might be interested in knowing when or how often the AI Players' heuristic search for best moves resulted in a randomised selection; if random selections occur too frequently, then it might mean the heuristic search process needs improving.

Cronista has been tested against two different systems, developed by different people who took different approaches to designing their systems. This gives some confidence that Cronista can extract meaningful provenance from systems where a shared-knowledge model is present or can be introduced. 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 the provenance reflect the internal processes or model representations in a way that can help answer the target audience's questions.

Discussion

The development cycle explored the potential to adapt Cronista for a non-MDE-developed system. Having previously removed the dependency on an MDE code generator for automating model instrumentation using AOP, non-MDE-developed systems could be considered for experiments. Removing the MDE framework from a system's implementation introduced several technical challenges with model element instrumentation (access/correlation/versioning). The immediate solutions for creating IDs for a POJO model are not as reliable or robust as the IDs provided by EMF. Thus, a modelling framework like EMF provides several supporting features that can reduce the design time costs of creating a provenance-aware system.

Retrofitting AIC with Cronista to collect the provenance of a Checkers game required several changes to be made to the system. Ideally, applying Cronista to a system would not require changing the code or model, aside from inserting activity scopes and annotations around the existing code. Unfortunately, AIC's shared-knowledge model of a Checkers game lacked representations of aspects about the AI players and the game being played. Additionally, the AOP framework used by Cronista imposed limitations that required some code changes, and the missing model representation also caused problems with collecting 'meaningful' provenance. For example, entities representing information passed between two activities were missing, such as the case where a move planning activity passes an entity representing a move to a "play move" activity. In this case, a move representation was missing.

Questions of provenance can only be answered if provenance data is recorded. Cronista's provenance collection is sensitive to the system's design: specifically, the shared-knowledge model level of detail, and the sequencing of activities. Thus, provenance data required to answer a question might not be recorded by Cronista because a system's design lacks information. For example, a facet of provenance questions may stem from a domain concept that the system's shared-knowledge model design does not include; these questions cannot be answered because of a system design problem, which should not be considered a failure of Cronista. This sensitivity relates to the PLEAD and PRIME work findings, which raise the need for explanation requirements and an SOA design view, which can be used to locate where provenance information needs to be recorded.

If a proposal was made to rebuild AIC with provenance collection as a requirement, then the one recommendation would be to use an MDE development approach with Cronista. Developing a

shared-knowledge model of a Checkers game using a framework like EMF creates a clear separation of concerns in the system design, separating the information in the shared-knowledge model from the system code. This could make it easier to consider the kind of provenance questions that could be answered using Cronista (see the answer to RQ1.3 in Section 7.1). Using the shared knowledge model designs, developers and stakeholders can discuss the information the shared knowledge model represents and how it can meet the explanation requirements, possibly adapting the PrIME or PLEAD methodologies (Section 2.5.2, 2.5.3) for having users/stakeholders pose provenance questions.

Chapter 7

Evaluation

The design and development phases of the research are completed, and the findings are evaluated in this section. In the subsection below, the answers to the questions raised in the thesis objectives (Section 3.3) are presented. These answers lead to a discussion, summarising the findings. Finally, future work is presented, and new research opportunities could build upon the work in this thesis.

7.1 Answering the research questions

The objectives of this thesis (Section 3.3) outlined the intent to create an approach to provenance awareness that used a system's shared-knowledge model. In this section the research questions are repeated and answered in the following order: RQ1 raised questions about how such an approach would be created, and what kind of answers it could provide to questions of provenance, RQ2 questioned the limitations and scope of the approach created in RQ1, and R3 asked about the costs of using the approach.

Research question 1

This question starts the development process for creating provenance awareness based on a shared-knowledge model.

RQ1

How can we implement an approach to provenance awareness for systems with a shared-knowledge base, which can answer a facet of provenance questions to support explanations for a system's decision-making (or behaviour) in terms of its execution without having to create an orthogonal design of the system to implement the provenance awareness?

Related Sections 4.1.6

RQ1 answer A conceptual model for collecting provenance of changes to a system's shared-knowledge model has been developed (Section 4.1). This model is based on W3C PROV (Section 2.3.4) and follows some rules from the provenance system architecture (Section 2.5.1). This model records the provenance of changes made to the system's shared-knowledge model by the system's agents (threads of execution), which perform activities defined by activity scopes placed in the system's code. The provenance collection approach for a shared-knowledge model system was then implemented as a set of reusable modular components, as demonstrated with Cronista (Sections 5.1 and 6.1).

The provenance of a system's shared-knowledge model enables facets of provenance questions to be answered, where a question asks for the source of a model state. For example, in the Traffic Manager demonstration, a developer asked questions about the light phase ending, and traced the cause (Section 5.2.5). The provenance collected for the Traffic Manager system cannot answer questions about the system if they are not represented in the shared-knowledge model (e.g. questions about CPU or memory consumption).

Figure 5.11 (on page 162) shows the metamodel for the shared-knowledge model of the Traffic Manager system. These same concepts should be present in a provenance graph of the system's execution as ENTITIES. This assumes that the entire shared-knowledge model has been instrumented, one or more activity scopes have been defined in the system's code, and at least one agent (thread) had an Observer. Cronista's Observer components create provenance messages to report interactions with a shared-knowledge model as a triple: agent, activity, and entity. The Observer creates provenance messages for the start and end of agent activities: these are needed to record

the provenance of activities that do not interact with a model, since without a provenance record for an activity, questions about that activity's occurrence and timing cannot be answered.

Cronista's approach to provenance collection is sensitive to a system's design, which can require some changes to expose certain information. However, there was no need to create an orthogonal design for the Traffic Manager system because a model-driven approach was used (Section 5.2.5). The Traffic Manager was designed with the intent of testing provenance collection for explaining its runtime behaviour (decision-making). As such, the metamodel (Figure 5.11) and shared-knowledge model within the system had been created to expose certain concepts and states for provenance collection.

The AIC system had an implicit shared-knowledge model (a Checkers game board), which also did not require an orthogonal design to be created for provenance collection (Section 6.2.5). However, the Checkers game board alone did not contain sufficient knowledge or information about other aspects of a Checkers game being played. For example, both AI players performed move-planning and move-playing activities, but the shared-knowledge model (game board) lacked important details relating to these activities, such as how many good moves an AI player found or if the AI player needed to tie-break the best move to play.

In the experiments with the Traffic Manager and AIC system, the provenance of execution was broadly collected from the system. PLEAD prescribes as part of its methodology that explanation requirements should be established first: this is then used to guide the design of the provenance awareness (collection) to be applied to a system. Designing provenance awareness ensures that 'enough' provenance is collected to answer specific questions about a system's execution. A similar approach could be taken to design a system to which Cronista would be applied. An approach driven by explanation requirements would highlight which agent, activities and shared-knowledge needed to be recorded. A system's shared-knowledge model design could then be checked to ensure sufficient intelligible representations exist; a thread of execution to be observed could be identified and activity scopes placed around sections of code that were of interest. Such an approach could reduce the number of iterations a developer might perform when applying Cronista's instrumentation and checking the provenance graph being produced. However, without explanation requirements, Cronista can be more broadly applied to a system as a bottom-up approach to provenance-based explanations. In this approach, Cronista observes and describes a system's execution and can answer some questions.

RQ1.1

How can provenance data be structured and stored in a way that enables storage costs to be managed by retention of time- or event-based periods?

RQ1.1 answer Collecting the provenance of a system's execution into a single provenance graph presents challenges, as discussed in Section 4.2.2. The concept of a *History Model* (HM) was designed (Section 4.2.4) to divide a system execution into a series of *time windows*, each containing a full snapshot of the model state and a provenance graph of changes that occurred to a model after the snapshot. Each time window in the HM is independent of all others: it contains all the information required to describe the evolution or changes to a system's model that occurred in the period it represents. Time windows permit a simple automated storage management strategy with the oldest time windows being deleted; there is no need to check or verify dependencies for the retained time windows or provenance graphs before deleting the data. Several other strategies can also be applied, where part of the time window's information could be deleted or marked for retention (Section 4.2.4).

RQ1.2

To what extent can the reuse of the existing infrastructure for the shared-knowledge system also reduce the costs of providing provenance awareness?

RQ1.2 answer A system's shared-knowledge model can be implemented using modelling frameworks like EMF, for which model repositories exist as a solution for model persistence. These modelling frameworks and model repositories provided several facilities for working with models, such as model object IDs and versioning. Implementing provenance collection for a shared-knowledge model requires a model implementation to provide interfaces to enable model element access detection, correlation, and versioning (Section 5.1.4). A modelling framework like EMF, when combined with a model repository like CDO, can reduce the design-time effort required to implement a system that will be made provenance aware because it provides model ID and versioning facilities. Conversely, a system like AIC built using POJO for its model representation requires additional effort to introduce the needed model IDs and versioning (Section 6.2.6).

A CDO model repository was used to implement the first History Model Store (Sections 5.1.4 and 5.2.4). This was because PROV-DM could be implemented as an EMF model and persisted in CDO, in much the same way as a system's knowledge model. The Traffic Manager's shared-knowledge model was developed using EMF, enabling the creation of an explicit model representing the system's concepts. CDO was then used to manage the changes that agents made to the model at runtime, using model versioning. The use of EMF and CDO for both History Model and shared-knowledge model simplified the maintenance of referential integrity between the shared-knowledge model and provenance (ENTITIES). Regarding ease of adoption, MDE developers might be more inclined to adopt provenance awareness if technologies they are familiar with are used for its implementation. The EMF model of PROV-DM could also be extended to create the history model, with time windows and model snapshots. The techniques used to create the History Model enable the Traffic Manager system to be created. In this way, a common set of MDE skills could be transferred between developing a domain-specific system and provenance collection.

EMF and CDO offer additional helpful conveniences. For example, snapshots of a system's shared-knowledge model can be taken using CDO's versioning system (as seen in the Traffic Manager experiment in Section 5.2.5). This had the advantage of being quicker and more efficient with storage space than simply saving a copy of a model to disk. Similarly, the representation in a system model can be directly seen in a provenance graph; small parts of the system model can be stored in the native types/format in their ENTITY nodes on a provenance graph.

MDE tool ecosystems such as the one present for EMF-based modelling languages can be used for the analysis of provenance. However, these tools are slightly cumbersome for traversing or querying a provenance graph (Section 5.3.2). Similarly, graph visualisation was possible via model transformations, but presented challenges when used on large graphs, as they can be hard to fit on screen and navigate. Future work could explore creating provenance graph exploration tools more suitable to the task, which could filter or limit the amount of provenance displayed on screen. These could be implemented using MDE and made compatible with EMF-based provenance graph representation.

RQ1.3

For which common facets of provenance questions can the collected provenance data provide some information to support an answer?

RQ1.3 answer The first phase in PrIME (Section 2.5.2 on page 58) involves eliciting provenance questions from a user as requirements. These questions are then transformed into provenance queries, which are likely to begin with a data item representing a state or value whose origins need to be traced. With a shared-knowledge model system, all these states/values of interest should be represented in the system's model. Thus, the facets of answerable provenance questions could be approximated from the shared-knowledge model. The provenance model (PROV-DM) implicitly answers several facets of provenance questions, like those of past history or data flow.

Zerva et al.'s paper on classification of provenance questions [116] identified some facets of provenance questions (Section 2.5.2), which could be organised by the source of information (model) used to answer a question. Cronista provides information for answering questions from two different models: the system's shared-knowledge model, or Cronista's provenance model (based on the provenance ontology PROV-DM). Thus, Zerva et al.'s facets could be organised as:

- **Shared-knowledge model sourced information**. A system's shared-knowledge model might contain knowledge representation (data items) for answering these questions; Cronista's provenance recordings capture interactions with these shared-knowledge model features to source information for answers. A system must be designed to be aware of these concepts or have knowledge of them to enable questions to be answered.
  - **Service and provider identity**
  - **Resource and physical deployment**
  - **Non-functional properties and quality of service**
- **Provenance model sourced information**. The provenance model created by Cronista enables some questions to be answered implicitly because of the nature of PROV-DM. In a shared-knowledge model system, data should flow between activities via the model, which would be recorded in provenance. These provenance recordings are the 'past history' of system executions and can be retained as long as necessary for answering questions.
  - **Data flow**
  - **Past history**
- **Shared-knowledge OR provenance model sourced information**. Some question facets can be answered using information sourced from either of the models.

  - **Time**. Provenance questions relating to time can be answered with time information contained in the shared-knowledge model. For example, the system model might track the time of an event or message. The provenance model created by Cronista is not dependent on timestamps (since timestamps are not used to form the provenance model). However, the Observer of an agent can report times from an agent's local clock in provenance messages (e.g. activity start/end times, which can then answer time questions for when activities occurred in relation to an agent's clock, etc.).
  - **Routes not followed.** Cronista's provenance records the routes followed, and does not directly record alternative activities or routes. However, if a system loops over the same section of code which may take a different route, then some provenance queries could be used to expose alternative routes. Alternatively, a system's shared-knowledge model could contain information for a path not taken; i.e. the shared-knowledge model contains information on the reason 'why' a decision/path was taken over an alternative, which could be recorded in provenance.
  - **Actors**. Cronista considers each system thread as an agent, responsible for executing activities and interacting with the shared-knowledge model. Cronista's concept of an agent could be used to assign responsibility to something external to the system (such as a user). To do this, a system designer would need to have the system create a thread of execution that handled the user's activities and model interactions. An Observer of the user's thread could then collect the provenance of the user's impact on the system, enabling questions of responsibility to be answered about a user.

- **External to Cronista**

  - **Design information**. Cronista does not record the provenance of the design-time activities for a system, and cannot directly answer these questions. However, when a system is designed in a model-driven way, the design artefact could be closely related to the runtime model representation, which Cronista reflects in a provenance graph. This could be advantageous if a design information question is raised that is related to other facets of questions for which Cronista could supply provenance information.

Cronista's approach to collecting the provenance of changes to a system's shared-knowledge model is sensitive to the system's design. This was seen in the AIC demonstration in Section 6.2.5,

where missing concepts needed to be added to the shared-knowledge model. When considering what questions Cronista might answer, this weakness becomes a strength; in some cases, a user's provenance question can be tested by examining the metamodel of a system's shared-knowledge model (like the one from Traffic Manager in Figure 5.11 on page 162). For example, the Traffic Manager's metamodel does not include a representation of the day's weather: therefore, any facet of a question relating to the weather cannot be answered directly using the provenance. By contrast, a direct answer can often be given to a question of provenance for something which *is* articulated in a shared-knowledge model, allowing answers for some of the more trivial provenance questions to be quickly validated.

Ensuring that a facet of a provenance question can be answered also requires that some activity scopes and threads of execution are being observed. While Cronista's model instrumentation can be liberally applied to an entire model with code generation, this does not guarantee that the provenance of specific model elements is recorded, since only those model elements accessed within an activity scope of an observed agent are recorded to provenance.

The Traffic Manager's MAPE architecture (Section 5.2.5) guided the application of the activity scopes. A developer could align activity scopes with high-level activities (e.g. monitoring) and then subdivide the activity scope into more granular activities, like reading the LAD sensors. This approach enabled facets of questions to be answered for each phase of the MAPE loop.

With the AIC system, the Checkers game activities were decomposed into more fine-grained activity scopes to create a description of a Checkers game being played. However, with AIC, some activity scopes were deliberately omitted to avoid collecting the provenance of some move-searching activities which would flood the provenance graph with data. Therefore, the second consideration for answering a user's provenance question is to identify an agent and activity scope that interacts with a shared-knowledge model concept relating to the question.

Research question 2

The line of questioning in research question 2 explores the limitations of the approach developed to answer RQ1, and the attempts to minimise the limitations.

RQ2

How can the approach generated by RQ1 be used with systems that have manually-written (less consistent) code implementing their shared-knowledge model?

RQ2 answer The conceptual design of the provenance collection (Section 4.1) and the initial implementation of Cronista (Section 5.1) were inspired by MDE practices and tools. In particular, the initial implementation took advantage of the EMF framework and its features, in order to reduce the development cost of the thesis research. Both the model code generator and EObject features provided solutions to practical problems; for example, persistent IDs for objects representing a model.

Removing the requirement for a model code generator motivated the design of an alternative automated approach to instrumenting MDE-developed shared-knowledge models. While the EMF code generator approach would work in most situations, there is the potential for Cronista's approach to be blocked. For example, a developer may have already replaced the EMF code generator base class with their own customised class, which can not be extended or replaced with Cronista's. Secondly, other model code generators may not provide customisation capabilities like those of EMF's code generator. Using AOP to automate the instrumentation of the shared-knowledge model at the Java compilation stage should work with other Java-based MDE frameworks, and some non-MDE developed systems.

Using AOP to automate the instrumentation of a shared-knowledge model also saves design-time effort. However, Cronista's Observer requires model elements to have a globally unique ID that persists for the lifetime of the model element, to support model element access, correlation, and versioning. Without a globally unique ID scheme, any design-time cost savings offered by AOP automation could be offset by the development effort of adding back these features to a model. For the AIC demonstration in Section 6.2.5 the Java *Object HashCode* was used to seed an ID for identifying the objects representing the system's shared-knowledge model. This approach was tested and found to work for the AIC demonstration, but the Javadocs do not guarantee globally unique *HashCodes*, and *HashCodes* only persist under certain conditions (i.e. lifetime of the Object instance). Therefore *HashCodes* only partially meet Cronista's model element ID requirements for model element access, correlation, and versioning.

When Cronista's Observer identifies model elements using Java *HashCodes*, the recorded

provenance entities are tightly coupled with the Java *Object* instance. Arguably, Cronista's Observer is recording the provenance of the Java *Object* instance and not the model element. For example, in the context of the shared-knowledge model, a model element may be unchanged during a certain activity. However, the Java *Object* representing the model element could be freed from memory and a new *Object* created (with the same shared-knowledge value) during the same activity: this new Java *Object* may have a new *HashCode* even though it represents the same model element and value. In this scenario, the provenance of the Java *Object* and the model element have diverged: a new Java Object *HashCode* implies a new model element, which results in a new ENTITY (a new entity ID is created using the new *HashCode*). The existing Java *Object* representing the model element should not have been released from memory unless the model element was also being replaced with a new instance.

RQ2.1

Using the new approach, can the results of the original evaluation experiment be reproduced, and does this approach add any new capabilities?

RQ2.1 answer The dependency on an MDE code generator was identified in Section 6.1 as a problem, which drove the development of a new AOP-based approach to automating the model instrumentation. To validate that this new approach worked correctly, the Traffic Manager experiment in Section 5.2.5 was repeated in Section 6.1.5. Provenance graphs of the Traffic Manager's execution produced with the different instrumentation approaches (code generation and AOP) were compared using a query to trace a system fault. The query could be used to find the fault in the system with both approaches; the provenance graph fragments produced by the query were equivalent. This gives confidence that provenance recorded using Cronista's code generation or AOP instrumentation approach is equivalent.

Cronista's AOP model instrumentation enables more control when applying the instrumentation to a model, because code annotations allow for more precise control of the automated process, unlike the EMF code generator approach that can only apply instrumentation to an entire model. However, this additional control of the automated model instrumentation using code annotations comes at a cost: a developer must (manually) apply the annotations to the shared-knowledge model. A developer

can reduce their manual effort with some careful consideration for the annotation approach they use. For example, they could use class-level annotations wherever possible, with an appropriate white-/black-listing approach for provenance to reduce the number of annotations required to achieve the desired coverage.

The evaluation of the runtime metrics in Section 6.1.6 (answer to LRQ3) indicates that the code generation approach has a slightly lower memory overhead than the AOP one. The table of results in Section 6.1.5 (on page 218) shows the memory for a 200 tick simulation without provenance to be 11.26MiB, and the relative increases in memory for each model instrumentation method were: CodeGen 37%, AspFull 44%, and AspPart 41%. There may be some edge cases where the extra memory overhead of AOP might prevent the approach from being used, but in most cases, AOP could be used in place of code generation to achieve equivalent provenance collection for similar overheads.

RQ2.2

What problems arise when applying provenance awareness to a system with a manually written shared-knowledge model, and do these problems relate to any known issues in the literature on provenance awareness?

RQ2.2 answer When using PrIME, there can be a problem with missing data items and hidden actors (PrIME phase 2 in Section 2.5.2 on page 58). PrIME identified a problem with hidden actors in SOA systems: these actors hold some provenance information which is not exposed for provenance collection, thus causing a gap in the recorded provenance information. Such gaps in provenance information could make some provenance questions unanswerable. A similar problem was seen while applying Cronista to AIC, which required the addition of some concepts not present in the system's shared-knowledge model (Section 6.2.5). Cronista and PrIME can both require a system's design or internal structures to be changed in order to expose provenance information. PrIME documents that in extreme cases of missing provenance information, extensive changes might be needed beyond the system being made provenance-aware, e.g. a company may have to computerise a paper process to acquire the missing provenance information.

RQ2.3

How can the approach answer a facet of provenance questions for program execution, using the shared-knowledge model and processes identified in a system with a manually-written shared-knowledge model?

RQ2.3 answer The answer to this question is similar to the answer to RQ 1.3 above. First, let us consider the kinds of provenance questions that could be answered from the perspective of PrIME: provenance questions are likely to start from a data item and trace its origin, and questions of provenance should be solicited from end-users to distil into provenance requirements. A system developed using an MDE approach should have a design view of the shared-knowledge model, and the system users may be able to identify the kind of data items whose origins they would want to trace. Thus, the use of an MDE approach could ease the process of soliciting and evaluating the kind of provenance question that can be answered, since users and developers can communicate using a common model.

In contrast, a system developed with a non-MDE approach might have some design view of the shared-knowledge model, but this is not necessary for implementing the system. A system developer could proceed to manually write the shared-knowledge model code, relying on the programming language representation to convey the model's design. For example, the source code is the only design-time representation of AIC. The lack of a design artefact or model that both developers and users understand could hinder the process of soliciting and evaluating answerable provenance questions. In the case of AIC, the shared-knowledge model and activities needed to be reverse-engineered or identified from the system code during the application of Cronista. Some broad assumptions were made that a game board representation existed within AIC, and that the provenance of a checkers game played with AIC could be tracked. This enabled some general provenance questions to be answered, such as "What did the board look like on Player 1's X turn?". Without in-depth knowledge of AIC's code or supporting design documentation, evaluating the kind of provenance questions that might be answerable was more difficult than the previous Traffic Manager system.

The second consideration is the effect the above problems have on answering the facets of provenance questions identified by Zerva [116]. In RQ 1.3, these facets of SOA provenance questions were grouped according to the source of available data to answer them. The implications of manually

written shared-knowledge models with no design views can be considered by following the same grouping:

- **Shared-knowledge model sourced information**. The ability to answer questions in this facet is subject to the sensitivity of provenance collection to the design of a system's shared-knowledge model. The ability to evaluate the kind of provenance questions that could be answered depends on being able to identify data items of interest in the system's shared-knowledge model. In the case of AIC, only the source code was available, and the shared-knowledge model was not consolidated within a single area of the codebase for the data item source to be evaluated. Thus, to answer certain provenance questions in this group required a refactoring of AIC to add features to the shared-knowledge model that were missing from its design or hidden from provenance.

  Zerva identified SOA questions around these facets:

  - **Service and provider identity**
  - **Resource and physical deployment**
  - **Non-functional properties and quality of service**

  These facets would require a developer to consider information about them 'important enough' to be included in the system's shared-knowledge model. For a manually written model, some of these facets could be easily overlooked. For example, the facet of 'non-functional properties and quality of service' is one that a developer might under-represent in a manually written shared-knowledge model because the required information may not be needed to meet the system's functional requirements.

- **Provenance model sourced information**. As before, these questions are implicitly answered by the nature of PROV-DM and provenance recording. However, there could be limitations on the extent or completeness of questions that can be answered using information inherited from the shared-knowledge model data source. A manually written model might be missing some knowledge or information, using an MDE approach might mitigate against this problem. These questions are dependent on provenance being recorded correctly over time. For AIC, the collected provenance could enable a game to be replayed as a sequence of recalled game board states.

  - **Data flow**
  - **Past history**

- **Shared-knowledge or provenance model sourced information**. This group includes facets that have dependencies on the shared-knowledge model, meaning that certain provenance questions may not be answerable.
  - **Time**: Questions concerning turns (next/prior) could be answered using AIC's provenance, in part because the provenance structure imposes a sequence on the captured changes to the game board.
  - **Routes not followed**: These can be explored to some extent with AIC's provenance, and the number of 'best moves found' can be recalled. However, the provenance of the heuristic search (which explores potential game paths resulting from a move) would need to be recorded in order for more detailed questions to be answerable about prospective game board states.
  - **Actors**: Actors are not used to represent each AI player in AIC. This would have required AIC to be refactored into a multi-threaded application in order for Cronista to create appropriate agent representations for players 1 and 2 (AI or Human). To overcome Cronista's limitation, separate ACTIVITY and ENTITY representations were needed for each player's specific activity or shared-knowledge representations.

- **External to Cronista**
  - **Design information**. For AIC, this consisted of the source code; this is a facet of questions that is outside of Cronista's scope of provenance collection.

Research question 3

The last question explored the costs of applying provenance awareness with the developed approach. A system being developed with provenance awareness would incur some additional costs, and these costs can occur at design-time in the form of development efforts, or at runtime as an increase in the system resource usage (e.g. CPU, memory, disk space).

RQ3

What new knowledge of provenance awareness costs was discovered by the design and development of an artefact to answer RQ1?

RQ3 answer The costs of making a system provenance-aware can be reduced if a system is created using a modelling framework like EMF. A modelling framework can be used to implement the code for a system model which has model IDs, versioning, and persistence facilities; these simplify the recording of provenance by addressing some of the basic requirements for tracking model element accesses, correlating and versioning (Section 5.1.4).

Without modelling framework facilities, a provenance collection system like Cronista needs to provide supplementary components (e.g. model ID and versioning solutions). The design of these supplementary components is sensitive to the technology used to implement a system and its model implementation. The experiment in Section 6.2.5 identified some of the challenges a developer faces without a modelling framework. This thesis only lightly explored solving the problems of recording provenance for non-MDE-developed systems when trying to reduce Cronista's dependencies on EMF. For example, the shared-knowledge model in AIC is tracked using a Java Object *hashcode*: this approach has limitations, discussed in relation to design and development back in Section 6.2.4, and further referenced in the answer to RQ2 above. Existing MDE literature would provide developers with the knowledge to implement more rigorous solutions to these problems.

RQ3.1

What are the design-time activities for using the new approach, compared with existing provenance awareness methodologies?

RQ3.1 answer The approach to provenance awareness in the literature is based on recording the provenance of interactions between 'Actors' in an SOA system. Provenance awareness can be applied to non-SOA systems by creating an SOA design view of a system, which is then used to design and implement provenance awareness. This additional SOA design view increases design-time costs, both in its initial creation and during its maintenance necessary to keep it consistent with the main design view of the system. Another problem with this approach can arise where the SOA design view of a system creates a disconnect between the system's main design views and

the representations in the provenance recordings. The provenance recorded based on the new SOA design view may not align with the concepts or components of a system's existing design.

Design-time costs for implementing provenance awareness could be reduced if a system's existing design views could be used to enable provenance awareness. This thesis demonstrated an approach to enabling provenance awareness for shared-knowledge model systems. In this approach, the system's existing shared-knowledge model representations are reflected in the provenance recordings, with additional agent and activity representations that produce a type of execution provenance. For shared-knowledge model systems, this approach reduces the design-time costs for enabling provenance awareness by omitting the need for an SOA design view.

In the literature, the PLEAD and PrIME methodologies require specifications or requirements to be established for explanations or provenance awareness. These requirements are important for locating the important provenance information to be recorded in an SOA system, and a shared-knowledge model system can potentially simplify this information location problem as follows: A system can be created where all the important information or knowledge in the system is organised into a shared-knowledge model, and this model can be made provenance-aware. Without explicit provenance or explanation requirements, a broad facet of provenance questions could be answered if those questions asked about the origin of some knowledge represented in the system's model.

It will still be necessary to establish provenance or explanation requirements for some systems, especially when legal or regulatory concerns exist. The taxonomy of explanations [110] and explainable-by-design methodologies [59] seek to guide designers through establishing these requirements. However, there is room for exploring and demonstrating provenance awareness without fully knowing what provenance information needs to be recorded. In the background, Section 2.4, some of the different motivations for 'explanations' were discussed, including unpredictable or emergent behaviours.

RQ3.2

What runtime overheads were seen on the systems used to evaluate the implementation, and how do they compare to the results published for provenance awareness?

RQ3.2 answer Cronista's runtime metric results can be found in the demonstration Sections 5.2.5, 5.3.5, 6.1.5, and 6.2.5. Broadly, for control applications that perform some short periodic execution of a control loop like the Traffic Manager, execution times are not significantly affected. On the other hand, more processing-intensive applications like AIC may see execution times increase: with AIC, this added approximately 3 seconds to each game seed played, which took between 106 (shortest) to 425 (longest) seconds with Cronista.

The memory footprint of a system is increased when Cronista's provenance awareness is applied. The Traffic Manager system saw a memory increase in the 5000-tick experiment: the baseline 14.74MiB rose to 227.65MiB with CDO and 127.44MiB with JanusGraph (results in Section 5.3.5). This increase in memory indicates a need for further investigation into memory optimisation. However, the memory footprint of AIC did not grow as significantly as that of the Traffic Manager; the results in Section 6.2.5 shows AIC's highest increase in memory usage in game Seed3, which rose from 1.55MiB to 15.18MiB. Interestingly, at 386 seconds this is not the longest-running game, compared to game Seed4 which ran for 425 seconds. The difference in total memory usage with Cronista between game Seed3 (15.18MiB) and Seed4 (15.11MiB) is small, combined with slight network I/O differences (Seed3 17.58MB, Seed4 17.41MB), suggesting that more provenance was recorded in Seed3. AIC's memory footprint was not increased as significantly as for the Traffic Manager, because AIC recorded less provenance (indicated by lower HMS network I/O metrics). Thus, a system's memory footprint will increase depending on the amount of provenance being recorded, which in turn is affected by the provenance requirements and nature of the system. The size of the provenance graph resulting from the provenance recordings will depend on the amount of duplicated information in the provenance recordings and the Curator's ability to re-use existing graph representations.

The reviewed provenance awareness literature (Section 2.5) focused on the requirements, methodologies, structures and representation of provenance information, and do not present runtime metrics. Collecting provenance from an SOA system would probably introduce a very small overhead on each service node. A small amount of provenance is recorded from each service node, with little noticeable effect on the wider distributed system. In contrast, Cronista's shared-knowledge model approach to provenance recording collects detailed provenance information from the system (node). As such, runtime metrics are a more important consideration when applying Cronista's style of provenance recording, which can become intensive.

Groth et al. investigated the recording of provenance from a protein compressibility experi-

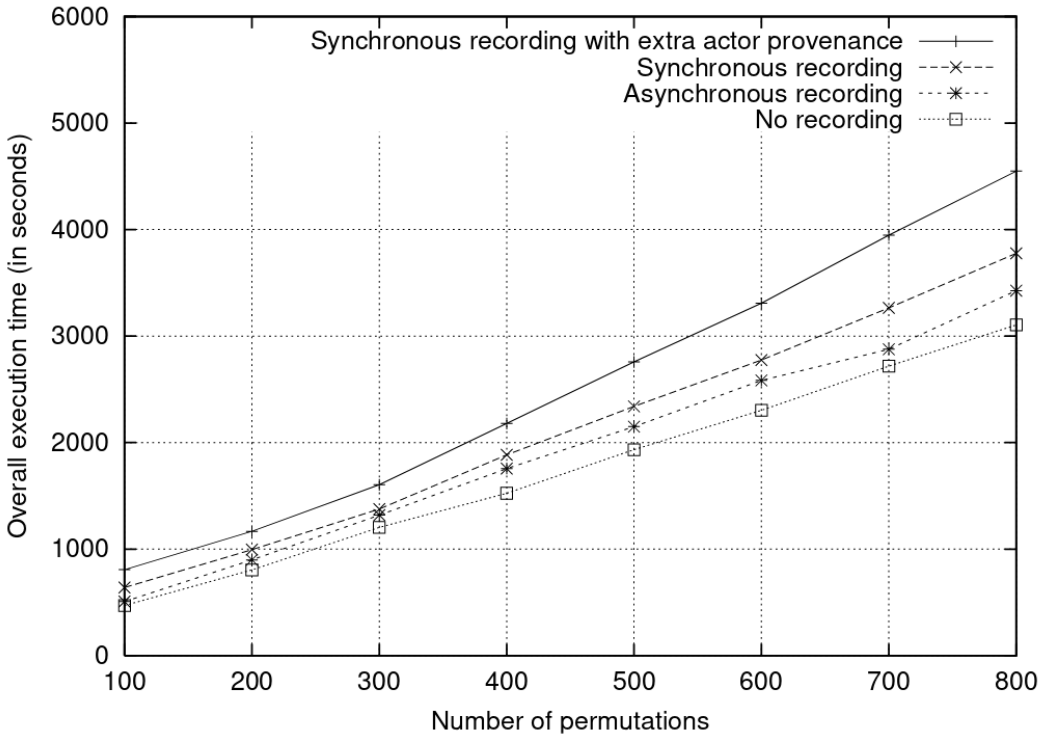

Figure 7.1: The overall execution time of the protein complexity workflow, for an increasing number of permutations, and with different provenance recording configurations (Results published by Groth et al. [50])

ment [50], publishing some metrics for the CPU overheads incurred using PReServ (PReServ is presented by Groth et al. in another paper [51]). Their results include a comparison between synchronous and asynchronous approaches to provenance recording. It is worth noting that Cronista's provenance recording is asynchronous, as the instrumented system does not wait for the Observer's provenance messages to be acknowledged by the Curator, and provenance recording activities do not block a system's execution. Groth et al. report a similar profile for the CPU overheads as seen with Cronista. The overall execution time of the system with provenance being recorded took longer than the established baseline (without provenance). When progressively increasing the number of permutations (i.e. the system workload) in each run of the experiment, a linear increase in the overall execution time was seen. They observed synchronous provenance recording to be slightly slower than the asynchronous approach, which added less than 10% to the overall execution time (as shown in Figure 7.1). Including extra provenance information in the synchronous provenance recording extended the overall execution time (presumably, this approach was taken to test the worst-case scenario); this is to be expected as recording more provenance information should increase overheads.

The metrics for PReServ (2005) and Cronista (2019) should not be directly compared to establish which approach is more 'efficient': with approximately 14 years between the experiments, it would be difficult to account for all the external factors that could affect the runtime metrics.

Cronista's Curator process removes a significant amount of duplicated information from provenance messages while recording data to the provenance graph. There is duplicate information in the provenance messages from the Observer to the Curator to enable stateless processing of the provenance messages, which is a recommendation in the literature around the architecture for provenance systems (Section 2.5.1). Some duplicate provenance information can be seen across time windows (Section 4.2.4): this duplication is necessary to enable time windows to be forgotten without causing a loss of information. Cronista conforms to the principle of the definition and does not create unnecessary duplication in the provenance recordings.

The PLEAD literature [60] defined valuable provenance information as follows: "all provenance information that enables a data controller to demonstrate a system is compliant is valuable". This statement highlights an interesting perspective that could be overlooked when evaluating the amount of provenance recorded, and the value of the provenance in terms of practical use. For example, a valuable use for provenance data would be tracking down the causes of a fault. If a system has no faults, the data collected might be considered excessive or worthless, but in fact additional value can be realised by using it to confirm a system is working as expected. Cronista's HMS and provenance recording enable both of these provenance values to be realised, as seen in the Traffic Manager experiments (Section 5.2.5). In this experiment, the provenance was queried to confirm the behaviour of healthy and faulty versions of the Traffic Manager system.

7.2 Discussion and Future work

The design science research described in this thesis has produced some artifacts that test an approach to automated provenance collection from a system with a shared-knowledge model. This initial work demonstrates the potential to collect provenance from a shared-knowledge model more directly, without using an SOA design view, where the provenance can answer facets of questions about the system's execution. A number of potential development directions were identified, which fall outside the scope of this thesis and are discussed here as potential future work. Some of these involve linking to the broader literature or aligning with existing frameworks. In other cases, artefacts

could be further refined to further generalise or prescribe approaches to collecting provenance from a shared-knowledge model - the potential refinements for each artefact type are discussed. Finally, a short summary closes the thesis.

7.2.1 Linking to / aligning with the broader theoretical and practical context

The existing literature for provenance-awareness and Model-Driven Engineering provide a wealth of knowledge for their respective domains; exploring the overlap has yielded this thesis, and there are further lines of research in this space.

Extension or adaptation of PrIME/PLEAD

The provenance-awareness literature reviewed in Section 2.5 was developed over several years, predating this thesis, and its most recent works (PLEAD) occurred in parallel with this thesis. Approaches to provenance-awareness in this literature use an SOA design view to achieve a generalised approach to provenance. Those findings might transfer or be applied to the research in this thesis on a view of provenance based on shared-knowledge models. PrIME and PLEAD could be extended or adapted to incorporate shared-knowledge model provenance in place of the SOA design views.

Deeper research into the overlaps and alignments between MDE and PROV-DM

The use of Model-Driven Engineering in the thesis has highlighted some interesting overlaps with the tools and approaches needed to enable systems with provenance. MDE techniques and tools seem well suited for creating systems where provenance collection or awareness is required. Swartout's [106] paper foreshadows the need for automation and intermediate layers of knowledge for explaining systems. Knowledge can be encoded as a model using MDE tools to create an appropriate modelling language for modelling. These models can then be used to generate code, thus automating the transformation of models to code, which is similar to Swartout's 'automated programmer'.

Provenance ontologies are languages to express provenance information, which can be used to model a system's workflow during execution.

PROV-DM (PROV Data Model) could be considered a modelling language in the context of MDE, since it has a language model shown as a set of UML representations in its documentation [49], and provenance recorded using PROV-DM can be presented/written using different notations. These are

similar characteristics to the kind of models used in MDE.

Further research into the overlap between MDE and provenance could further explore their interactions. For example, the management of unique IDs to distinguish components and the tracking of changes (versioning) are common implementation problems that both need to be overcome with minimal developer effort to manage system design costs.

7.2.2 Refinement of Cronista and shared-knowledge model provenance collection

Several lines of work could stem from the further refinement of Cronisa and the conceptual approach to collecting the provenance of a shared-knowledge model's evolution. The different lines of work are discussed in the subsections below.

Enhance the conceptual provenance model

The current mapping of the PROV-DM concepts to a system's interactions with its shared-knowledge model does not prescribe in detail all the data that should be collected. The PROV-DM ontology or Provenance-Templates can be used to further specify the information that needs to be collected. However, the concern for this research project was that being overly prescriptive could significantly hinder the scope of systems which the provenance collection approach could be used with. For example, the provenance collection and graph building do not require accurate timestamp information to be present in provenance messages: this lack of a timestamp requirement enables provenance to be collected from systems that do not have a time source (or no unified system-wide clock, when multiple clocks are present). A future line of work could examine different systems/domains against sets of explanation requirements, and offer more prescribed guidance or provenance instrumentation. These more prescribed approaches for applying provenance-awareness would enforce or check that specific kinds of provenance data are gathered, to meet the requirements for certain types of explanations.

Approaches to model instrumentation and describing the shared-knowledge model elements

During the design and development of Cronista, practical challenges with recording the provenance of model elements during execution dominated the development work, such as identifying the model objects in memory, or discovering the behaviours and execution details of EMF. For example, instances

of EMF model objects loaded from storage are unique copies that exist in memory and are not shared between threads. Cronista's implementation could be improved in future work: optimisations of the Observer components to reduce the overhead on a system's resources, or new programming techniques beyond code generation/AOP for model instrumentation could be introduced (e.g. Java bytecode instrumentation [90]). However, the work presented provides some knowledge of the practical issues a developer might experience, and identifies some of the requirements for tracking model elements (access, correlation, and versioning). Similarly, more prescribed requirements for the model information to record as reusable ENTITY representations (like PROV-DM or Provenance-Templates) for specific models/systems/domains could be added to Cronista's component library. These entity templates could reduce developer efforts by providing further guidance for implementing provenance, by automatically collecting some known-to-be-useful information.

Assignment of responsibilities to agents

Cronista maps PROV-DM's AGENT concept to a Java thread; this requires a developer to align a system's responsible 'Agents' to a programming construct in the system's design (the threads). Given the design challenges with tracking the model (described above), other approaches to mapping an agent were not explored; a broad assumption is made that threads could be aligned to a user's session or user interface, which could then attribute ACTIVITIES in the system that a user triggers via the system's user interface. With AIC (a single-threaded program with 2 AI players), the current requirement to align agents to threads prevented AGENT nodes from being used to represent the players' responsibility for a given ACTIVITY. As a workaround, ACTIVITY nodes with distinct descriptions for each player's activities were used: for example, two ACTIVITIES were used to represent which player was planning a move ('player 1 plans a move' and 'player 2 plans a move'). Ideally, this could have been represented with a single ACTIVITY 'player plans a move' with a *"wasAssociatedWith"* relation connecting it to the AGENT of a specific player. There could be an interesting line of design research in exploring how Cronista's Observer might detect and attribute responsibility within a thread where 'responsibility' is passed between agents, like the AI-Checkers player problem.

Detecting and describing the scope of an activity

Cronista's instrumentation approach to detecting activities during execution is based on 'activity scopes' that are aligned with Java's try-with-resource blocks. This limits an activity scope to be no smaller than a single line of code; how problematic this might be is not fully explored, but can be overcome by refactoring a system's code to enable more discrete activity provenance to be collected. Future work could explore instances where this limitation is problematic, and if the affected points in the code introduced 'intelligibility' problems; i.e. what the affected code does is hard to explain. However, Cronista's approach does cover potential issues that were considered as potentially very disruptive to the provenance collection. Unexpected errors during execution could cause program execution to leave the scope of an activity without the Observer detecting it. During the design and development of Cronista, 'unexpected' errors in Java were not an uncommon occurrence, and applying try-with-resource blocks to a section of code was one way of isolating where problems occurred in code, and often involved several layers of different components. Future work could seek to improve the try-with-resource approach to defining activities, which some developers might consider to be 'invasive' in a system's code. An alternative approach could explore the use of code annotations to determine the start/end of an activity scope, these might be 'less invasive' in a system's code.

The provenance model provided by Cronista is minimal, so as not to introduce limitations through over-prescribing the information required from a system when applying provenance collection. Cronista's provenance can uniquely identify instances of different activities (and their agents), detecting when they start and end, but the developer has to provide a text description of the activity to aid in answering provenance questions after an execution. An interesting line of research could examine ways to assist a developer with creating the activity description, such as templates for writing descriptions, or ways to link them to the system design documentation. For example, with some systems, it may be appropriate to connect an ACTIVITY with the system's version control system (VCS) containing its codebase. When a developer is reviewing provenance (possibly debugging a problem), from a link on an ACTIVITY they would be taken to the VCS and shown the section of code for the activity.

Provenance of the shared-knowledge commit or transaction processes

There is an unexplored aspect of a system's interaction with its shared-knowledge model, which is the internal processing and handling of the interaction. Cronista's approach to provenance collection only records provenance when an agent performs a successful read/write model access in the scope of an activity. There is another point of view that could be taken when recording the provenance of this interaction; a more detailed recording of the interaction process itself. This provenance could become tightly coupled with specific implementations of shared-knowledge models for multi-threaded systems. However, to use CDO's commit processes as an example, additional provenance information could record details of the commit transaction, commit messages, conflicts detected, and any additional mitigation (e.g. a merge). A CDO model repository has some additional features that have not been explored in this work: for example, CDO enables branches to be created for models. Branching a shared-knowledge model could enable an agent to explore what-if scenarios on an alternative version of the shared-knowledge model; Cronista's current ENTITY representation would need to reflect the branch a model element came from, in addition to the storage and memory versions.

Security and data protection

Cronista is able to automatically correlate and infer provenance information through the Observer, which is aware of agents, activities and the shared-knowledge model. It would be interesting to explore what kind of additional model implementation features might be required to enable the Observer to be aware of sensitive data. There are notable concerns with automated provenance collection, which could record sensitive data outside of a system's 'secure environment' [52]. This recording of sensitive data into a provenance system could constitute a breach of GDPR, which sets out requirements for the handling of data. Similarly, another approach could be to let Cronista record the provenance and sensitive data to the History Model, but flag the data and restrict access to it.

Extending Cronista for use with other programming languages and further experimentation

Cronista has models and instrumentation approaches for use with systems built using Java, but not all developers will build their shared-knowledge model systems using Java. Therefore, additional modules and instrumentation approaches are needed to bring Cronista and provenance-awareness to these other systems and developers. A motivation for researchers to do this work is that it would

enable experimentation with provenance collection using systems from other domains, and would allow researchers to investigate the use of provenance-based self-explanation to address domain-specific explanation requirements. In the background discussion (Section 2.4), some examples of the need for explanation to address specific problems can be seen, e.g. context-aware and knowledge-based systems.

The application and use of provenance-awareness for explanations can be explored using a top-down design process similar to PLEAD, which starts by gathering and analysing explanation requirements. Cronista enables a bottom-up approach to provenance-awareness, where provenance can be collected broadly from across a system, then through analysis and review of the provenance graphs produced, the kind of provenance and questions that can be answered could be discovered. This bottom-up experimental approach connects with ideas seen in PrIME, where the process of soliciting provenance requirements from users involves providing examples and explaining provenance concepts. Cronista could be used in an exploratory fashion to identify the potential provenance questions a system's existing shared-knowledge model and processes could answer.

Provenance-awareness user interfaces or summarising provenance

An area of work that was beyond the scope of research was the creation of user-friendly tools for interacting with provenance graphs. During the experiments with Cronista, existing developer tools were used to explore the collected provenance because developers may know of them already, lowering Cronista's adoption difficulty. However, as noted in the literature already, explanations can be audience-specific [60] and provenance data can quickly overwhelm users.

In the demonstrations where Cronista's provenance was explored, a difference in ease of use between query languages was observed: the declarative style of the Gremlin query language was preferable to the imperative style of EOL. With graph visualisation, the approach using Graphviz lacked facilities to control the visualisation of provenance. Whereas GraphExp offered a much better experience, it is specifically designed to visualise and explore complex graph databases. The cognitive load of exploring Cronista's history models could be lowered by designing a custom tool. This custom tool would have to provide better control of the amount of provenance information on-screen, by using filtering to exclude provenance information, limiting the number of connected provenance nodes to display, and organising the graph by order of occurrence (e.g. activity order, or entity versions).

More advanced approaches to providing a bridge between provenance data and explanations are discussed by Huynh et al. [61]. Provenance data needs to undergo some further synthesis to become an explanation that can be presented to an audience; the author of this thesis agrees with the view from Huynh et al. that the raw provenance data will overwhelm most audiences [61]. Reducing provenance information into shorter summaries is a possible solution: Moreau [79] discusses an approach to summarising provenance graphs by aggregating types of provenance.

Summarising the provenance graphs produced by Cronista could be very appropriate: this would imply reducing Cronista's provenance into a more manageable size with less detail, raising the level of abstraction for a human reader. The summarisation could be applied to the results of a provenance query before a graph is presented to the user asking a provenance question. A sub-graph of query results would be processed by the system, which would search for known patterns or sequences of ACTIVITIES with connected AGENTS and ENTITIES; sections of the sub-graph would then be replaced with smaller and less detailed sub-graphs, or replaced with text descriptions.

7.2.3 Closing summary

The use of provenance information collected at runtime to create a form of self-explanation seems possible with current technologies and techniques. Utilising automated programming techniques, like AOP or MDE code generation, some design-time costs can be mitigated. The use of MDE to design a system shared-knowledge model is a complementary approach that should be considered when a system is being designed that requires provenance awareness. Using MDE practices and tools to design a provenance-aware system offers design-time cost savings as a common model representation can be used for both a system's shared-knowledge and provenance system.

Bibliography

[1] A. Adadi and M. Berrada. Peeking inside the black-box: A survey on explainable artificial intelligence (xai). *IEEE Access*, 6:52138–52160, 2018. ISSN 2169-3536. doi: 10.1109/ACCESS.2018.2870052.

[2] E. Angelino, D. Yamins, and M. Seltzer. Starflow: A script-centric data analysis environment. In D. L. McGuinness, J. R. Michaelis, and L. Moreau, editors, *Provenance and Annotation of Data and Processes*, Lecture Notes in Computer Science, page 236–250, Berlin, Heidelberg, 2010. Springer. ISBN 978-3-642-17819-1. doi: 10.1007/978-3-642-17819-1_27.

[3] Apache Foundation. Apache tinkerpop homepage, 2022. URL `https://tinkerpop.apache.org/`. Date last checked: August 2024.

[4] Apache Foundation. Apache tinkerpop: Gremlin, 2024. URL `https://tinkerpop.apache.org/gremlin.html`. Date last checked: August 2024.

[5] P. Arcaini, E. Riccobene, and P. Scandurra. Modeling and analyzing mape-k feedback loops for self-adaptation. In *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, page 13–23, May 2015. doi: 10.1109/seams.2015.10.

[6] G. Barquero, J. Troya, and A. Vallecillo. Improving query performance on dynamic graphs. *Software and Systems Modeling*, Nov 2020. ISSN 1619-1374. doi: 10.1007/s10270-020-00832-3. URL `https://doi.org/10.1007/s10270-020-00832-3`.

[7] V. Bellotti and W. K. Edwards. Intelligibility and accountability: Human considerations in context-aware systems. *Human–Computer Interaction*, 16:193–212, 2001. doi: 10.1207/S15327051HCI16234_05.

[8] N. Bencomo and L. H. G. Paucar. Ram: Causally-connected and requirements-aware runtime models using Bayesian learning. In *Proc. of MODELS 2019*, pages 216–226. IEEE, 2019. doi: 10.1109/MODELS.2019.00005.

[9] N. Bencomo, J. Whittle, P. Sawyer, et al. Requirements reflection: requirements as runtime entities. In *Proceedings of ICSE 2010*, pages 199–202. ACM, 2010. doi: 10.1145/1810295.1810329.

[10] N. Bencomo, S. Götz, and H. Song. Models@run.time: a guided tour of the state-of-the-art and research challenges. *Software and Systems Modeling*, 18(5):3049–3082, 2019. ISSN 1619-1366. doi: 10.1007/s10270-018-00712-x.

[11] F. A. Bianchi, A. Margara, and M. Pezzè. A survey of recent trends in testing concurrent software systems. *IEEE Transactions on Software Engineering*, 44(8):747–783, 2018. ISSN 1939-3520. doi: 10.1109/TSE.2017.2707089.

[12] G. Blair, N. Bencomo, and R. B. France. Models@ run.time. *Computer*, 42(10):22–27, 2009. ISSN 1558-0814. doi: 10.1109/MC.2009.326.

[13] M. Brambilla, J. Cabot, and M. Wimmer. Model-Driven Software Engineering in Practice: Second Edition. *Synthesis Lectures on Software Engineering*, 3(1):1–207, Mar. 2017. ISSN 2328-3319, 2328-3327. doi: 10.2200/S00751ED2V01Y201701SWE004.

[14] D. Brody. AI-Checkers, Jan 2021. URL https://github.com/dbrody112/ai-checkers. Date last checked: August 2024.

[15] P. Buneman, S. Khanna, and W.-C. Tan. Data Provenance: Some Basic Issues. In G. Goos, J. Hartmanis, J. van Leeuwen, S. Kapoor, and S. Prasad, editors, *FST TCS 2000: Foundations of Software Technology and Theoretical Computer Science*, volume 1974, pages 87–93. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000. ISBN 978-3-540-41413-1 978-3-540-44450-3. doi: 10.1007/3-540-44450-5_6. URL http://link.springer.com/10.1007/3-540-44450-5_6.

[16] P. Buneman, S. Khanna, and T. Wang-Chiew. Why and Where: A Characterization of Data Provenance. In J. Van den Bussche and V. Vianu, editors, *Database Theory — ICDT 2001*, pages 316–330, Berlin, Heidelberg, 2001. Springer. ISBN 978-3-540-44503-6. doi: 10.1007/3-540-44503-X_20.

[17] J. Burrell. How the machine 'thinks': Understanding opacity in machine learning algorithms. *Big Data & Society*, 3(1):2053951715622512, June 2016. ISSN 2053-9517. doi: 10.1177/2053951715622512.

[18] J. Cámara, K. L. Bellman, J. O. Kephart, M. Autili, N. Bencomo, A. Diaconescu, H. Giese, S. Götz, P. Inverardi, S. Kounev, and M. Tivoli. *Self-aware Computing Systems: Related Concepts and Research Areas*, pages 17–49. Springer International Publishing, Cham, 2017. ISBN 978-3-319-47474-8. doi: 10.1007/978-3-319-47474-8_2.

[19] L. Carata, S. Akoush, N. Balakrishnan, T. Bytheway, R. Sohan, M. Seltzer, and A. Hopper. A primer on provenance. *Communications of the ACM*, 57(5):52–60, May 2014. ISSN 0001-0782, 1557-7317. doi: 10.1145/2596628.

[20] B. Chandrasekaran, M. Tanner, and J. Josephson. Explaining control strategies in problem solving. *IEEE Expert*, 4(1):9–15, 1989. ISSN 2374-9407. doi: 10.1109/64.21896.

[21] J. Cheney, L. Chiticariu, and W.-C. Tan. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases*, 1(4):379–474, 2007. ISSN 1931-7883, 1931-7891. doi: 10.1561/1900000006. URL http://www.nowpublishers.com/article/Details/DBS-006.

[22] M. Ciglan, A. Averbuch, and L. Hluchy. Benchmarking traversal operations over graph databases. In *2012 IEEE 28th International Conference on Data Engineering Workshops*, page 186–189, Apr 2012. doi: 10.1109/ICDEW.2012.47.

[23] T. Clark, B. Barn, V. Kulkarni, and S. Barat. Querying histories of organisation simulations. *Information Systems Development: Advances in Methods, Tools and Management - Proceedings of the 26th International Conference on Information Systems Development, ISD 2017*, 2017. URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-85081127542&partnerID=40&md5=1808bae0da77d949df795d13b7edee4c.

[24] K. Crawford, R. Dobbe, T. Dryer, et al. AI Now 2019 report. Technical report, AI Now Institute, 2019. URL https://ainowinstitute.org/AI_Now_2019_Report.html. Date last checked: August 2024.

[25] A. K. Dey, G. D. Abowd, and D. Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human–Computer Interaction*, 16(2–4): 97–166, Dec. 2001. ISSN 0737-0024. doi: 10.1207/S15327051HCI16234_02.

[26] A. Doan, A. Halevy, and Z. Ives. 14 - Data Provenance. In A. Doan, A. Halevy, and Z. Ives, editors, *Principles of Data Integration*, pages 359–371. Morgan Kaufmann, Boston, Jan. 2012. ISBN 978-0-12-416044-6. doi: 10.1016/B978-0-12-416044-6.00014-4. URL https://www.sciencedirect.com/science/article/pii/B9780124160446000144.

[27] D. Dominguez-Sal, P. Urbón-Bayes, A. Giménez-Vañó, S. Gómez-Villamor, N. Martínez-Bazán, and J. L. Larriba-Pey. Survey of graph database performance on the hpc scalable graph analysis benchmark. In *Web-Age Information Management*, LNCS, page 37–48. Springer, 2010. ISBN 978-3-642-16720-1. doi: 10.1007/978-3-642-16720-1_4.

[28] Eclipse Foundation. CDO Model Repository, Dec. 2019. URL https://www.eclipse.org/cdo/. Date last checked: August 2024.

[29] Eclipse Foundation. EMF Compare homepage, 2020. URL https://www.eclipse.org/emf/compare/. Date last checked: August 2024.

[30] Eclipse Foundation. The AspectJ project, Oct. 2021. URL https://www.eclipse.org/aspectj/. Date last checked: August 2024.

[31] Eclipse Foundation. EMFStore, 2022. URL https://www.eclipse.org/emfstore/. Date last checked: August 2024.

[32] Eclipse Foundation. Eclipse Modeling Framework (EMF), Aug. 2022. URL https://www.eclipse.org/modeling/emf/. Date last checked: August 2024.

[33] Eclipse Foundation. Eclipse Modeling Project, Aug. 2022. URL https://www.eclipse.org/modeling/. Date last checked: August 2024.

[34] Eclipse Foundation. Eclipse OCL (Object Constraint Language), 2022. URL https://projects.eclipse.org/projects/modeling.mdt.ocl. Date last checked: August 2024.

[35] Eclipse Foundation. SUMO homepage, 2022. URL https://www.eclipse.org/sumo/. Date last checked: August 2024.

[36] Eclipse Foundation. TraCI online documentation, 2022. URL https://sumo.dlr.de/docs/TraCI.html#introduction_to_traci. Date last checked: August 2024.

[37] J. Ellson, E. Gansner, Y. Hu, et al. Graphviz - graph visualization software, June 2020. URL https://graphviz.org/. Date last checked: August 2024.

[38] R. Feldt and A. Magazinius. Validity threats in empirical software engineering research - an initial survey. In *Proceedings of SEKE 2010*, Jan 2010. URL https://www.cse.chalmers.se/~feldt/publications/feldt_2010_validity_threats_in_ese_initial_survey.pdf.

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

[40] D. Garlan and B. R. Schmerl. Using architectural models at runtime: Research challenges. In *Proceedings of EWSA 2004*, volume 3047 of *LNCS*, pages 200–205. Springer, 2004. doi: 10.1007/978-3-540-24769-2_15.

[41] gdpr-info.eu. General Data Protection Regulation GDPR Website containing the official PDF of the Regulation (EU) 2016/679 (General Data Protection Regulation) . URL https://gdpr-info.eu/. Date last checked: August 2024.

[42] A. Gehani and D. Tariq. Spade: Support for provenance auditing in distributed environments. In P. Narasimhan and P. Triantafillou, editors, *Middleware 2012*, Lecture Notes in Computer Science, page 101–120, Berlin, Heidelberg, 2012. Springer. ISBN 978-3-642-35170-9. doi: 10.1007/978-3-642-35170-9_6.

[43] H. Giese, N. Bencomo, L. Pasquale, et al. Living with uncertainty in the age of runtime models. In *Models@run.time - Foundations, Applications, and Roadmaps*, volume 8378 of *Lecture Notes in Computer Science*, pages 47–100. Springer, 2011. doi: 10.1007/978-3-319-08915-7_3.

[44] H. Giese, T. Vogel, A. Diaconescu, S. Götz, N. Bencomo, K. Geihs, S. Kounev, and K. L. Bellman. *State of the Art in Architectures for Self-aware Computing Systems*, pages 237–275. Springer International Publishing, Cham, 2017. ISBN 978-3-319-47474-8. doi: 10.1007/978-3-319-47474-8_8.

[45] P. Gray, J. Williamson, D. Karp, and J. Dalphin. The research imagination: An introduction to qualitative and quantitative methods. *The Research Imagination: An Introduction to Qualitative and Quantitative Methods*, page 1–456, Jan. 2007. ISSN 9780521879729. doi: 10.1017/CBO9780511819391.

[46] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '07, pages 31–40, New York, NY, USA, June 2007. Association for Computing Machinery. ISBN 978-1-59593-685-1. doi: 10.1145/1265530.1265535. URL https://dl.acm.org/doi/10.1145/1265530.1265535.

[47] S. Gregor and I. Benbasat. Explanations from intelligent systems: Theoretical foundations and implications for practice. *MIS Quarterly*, 23(4):497, Dec. 1999. ISSN 02767783. doi: 10.2307/249487.

[48] P. Groth and L. Moreau. Recording process documentation for provenance. *IEEE Transactions on Parallel and Distributed Systems*, 20(9):1246–1259, Sept. 2009. ISSN 1045-9219. doi: 10.1109/TPDS.2008.215. URL https://eprints.soton.ac.uk/267309/.

[49] P. Groth and L. Moreau. PROV-Overview, 2013. URL https://www.w3.org/TR/prov-overview/. Date last checked: August 2024.

[50] P. Groth, S. Miles, W. Fang, S. Wong, K.-P. Zauner, and L. Moreau. Recording and using provenance in a protein compressibility experiment. In *HPDC-14. Proceedings. 14th IEEE International Symposium on High Performance Distributed Computing, 2005.*, page 201–208, July 2005. doi: 10.1109/HPDC.2005.1520960. URL https://ieeexplore.ieee.org/document/1520960.

[51] P. Groth, S. Miles, and L. Moreau. Preserv: Provenance recording for services. In *the UK e-Science All Hands Meeting 2005 (19/09/05 - 22/09/05)*, 2005. URL https://eprints.soton.ac.uk/262570/. Event Dates: September 2005.

[52] P. Groth, J. Sheng, S. Miles, S. Munroe, V. Tan, S. Meacham, and L. Moreau. *An Architecture for Provenance Systems*. Feb. 2006. URL https://www.researchgate.net/publication/39994555.

[53] M. Herschel and M. Hlawatsch. Provenance: On and behind the screens. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 2213–2217, New York, NY, USA, Jun 2016. Association for Computing Machinery. ISBN 978-1-4503-3531-7. doi: 10.1145/2882903.2912568.

[54] M. Herschel, R. Diestelkämper, and H. Ben Lahmar. A survey on provenance: What for? What form? What from? *The VLDB Journal*, 26(6):881–906, 2017. ISSN 0949-877X. doi: 10.1007/s00778-017-0486-1.

[55] F. Hilken and M. Gogolla. Verifying linear temporal logic properties in uml/ocl class diagrams using filmstripping. In *2016 Euromicro Conference on Digital System Design (DSD)*, page 708–713, Aug 2016. doi: 10.1109/DSD.2016.42.

[56] N. Hoang Thuan, A. Drechsler, and P. Antunes. Construction of design science research questions. *Communications of the Association for Information Systems*, page 332–363, 2019. ISSN 15293181. doi: 10.17705/1CAIS.04420.

[57] H. J. Holz, A. Applin, B. Haberman, D. Joyce, H. Purchase, and C. Reed. Research methods in computing: what are they, and how should we teach them? *SIGCSE Bull.*, 38(4):96–114, June 2006. ISSN 0097-8418. doi: 10.1145/1189136.1189180.

[58] huanjiayang et al. provpy project website. URL https://pypi.org/project/provpy/. Date last checked: August 2024.

[59] T. Huynh, N. Tsakalakis, A. Helal, S. Stalla, and L. Moreau. *Explainability-by-Design: A Methodology to Support Explanations in Decision-Making Systems*. arxiv.org, June 2022. URL http://arxiv.org/pdf/2206.06251.

[60] T. D. Huynh, S. Stalla-Bourdillon, and L. Moreau. *Provenance-based Explanations for Automated Decisions: Final IAA Project Report*. July 2019. URL https://kclpure.kcl.ac.uk/ws/portalfiles/portal/113483446/ico_iaa_report.v4.pdf.

[61] T. D. Huynh, N. Tsakalakis, A. Helal, S. Stalla-Bourdillon, and L. Moreau. Addressing regulatory requirements on explanations for automated decisions with provenance—a case study. *Digital Government: Research and Practice*, 2(2):1–14, Apr. 2021. ISSN 2691-199X, 2639-0175. doi: 10.1145/3436897.

[62] D. Inc. Docker website, May 2022. URL https://www.docker.com/. Date last checked: August 2024.

[63] S. Jiang, P. T. Groth, S. Miles, V. Tan, S. Munroe, S. Tsasakou, and L. Moreau. Client side library design and implementation workpackage: Wp9. Nov. 2006. URL https://www.academia.edu/2737288/Client_Side_Library_Design_and_Implementation_Workpackage_WP9.

[64] S. Jouili and V. Vansteenberghe. An empirical comparison of graph databases. In *2013 International Conference on Social Computing*, page 708–715, Sep 2013. doi: 10.1109/SocialCom.2013.106.

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

[66] G. Kiczales, J. Lamping, A. Mendhekar, et al. Aspect-Oriented Programming. In *Proceedings of ECOOP'97*, volume 1241 of *LNCS*, pages 220–242. Springer-Verlag, Jyväskylä, Finland, June 1997. ISBN 978-3-540-63089-0. doi: 10.1007/BFb0053381.

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

[68] T. Kovács, G. Simon, and G. Mezei. Benchmarking graph database backends—what works well with wikidata? *Acta Cybernetica*, 24(1):43–60, May 2019. ISSN 0324-721X. doi: 10.14232/actacyb.24.1.2019.5.

[69] T. Kühne. Matters of (meta-) modeling. *Software & Systems Modeling*, 5(4):369–385, Dec 2006. ISSN 1619-1374. doi: 10.1007/s10270-006-0017-9.

[70] B. Lim, A. Dey, and D. Avrahami. Why and why not explanations improve the intelligibility of context-aware intelligent systems. *Conference on Human Factors in Computing Systems - Proceedings*, 04 2009. doi: 10.1145/1518701.1519023.

[71] B. Y. Lim and A. K. Dey. Assessing demand for intelligibility in context-aware applications. In *Proceedings of the 11th international conference on Ubiquitous computing*, page 195–204, Orlando Florida USA, Sept. 2009. ACM. ISBN 978-1-60558-431-7. doi: 10.1145/1620545.1620576. URL https://dl.acm.org/doi/10.1145/1620545.1620576.

[72] B. Y. Lim and A. K. Dey. Toolkit to support intelligibility in context-aware applications. In *Proceedings of the 12th ACM international conference on Ubiquitous computing*, page 13–22, Copenhagen Denmark, Sept. 2010. ACM. ISBN 978-1-60558-843-8. doi: 10.1145/1864349.1864353. URL https://dl.acm.org/doi/10.1145/1864349.1864353.

[73] P. A. Lopez, M. Behrisch, L. Bieker-Walz, J. Erdmann, Y.-P. Flötteröd, R. Hilbrich, L. Lücken, J. Rummel, P. Wagner, and E. Wießner. Microscopic traffic simulation using sumo. In *The 21st IEEE International Conference on Intelligent Transportation Systems*. IEEE, 2018. URL https://elib.dlr.de/124092/.

[74] C. A. Lynch. When documents deceive: trust and provenance as new factors for information retrieval in a tangled web. *Journal of the American Society for Information Science and Technology*, 52(1):12–17, Jan. 2001. ISSN 1532-2882. doi: 10.1002/1532-2890(2000)52:1⟨12::AID-ASI1062⟩3.0.CO;2-V.

[75] 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*, ICSENT 2018, page 1–8. Association for Computing Machinery, Dec 2018. ISBN 978-1-4503-6101-9. doi: 10.1145/3330089.3330108.

[76] Microsoft. github.com, Nov 2022. URL https://github.com. Date last checked: August 2024.

[77] S. Miles, P. Groth, S. Munroe, and L. Moreau. Prime: A methodology for developing provenance-aware applications. *ACM Transactions on Software Engineering and Methodology*, 20(3): 8:1–8:42, Aug. 2011. ISSN 1049-331X. doi: 10.1145/2000791.2000792.

[78] P. Missier, K. Belhajjame, and J. Cheney. The w3c prov family of specifications for modelling provenance metadata. In *Proceedings of the 16th International Conference on Extending Database Technology - EDBT '13*, page 773, Genoa, Italy, 2013. ACM Press. ISBN 978-1-4503-1597-5. doi: 10.1145/2452376.2452478.

[79] L. Moreau. Aggregation by provenance types: A technique for summarising provenance graphs. *Electronic Proceedings in Theoretical Computer Science*, 181:129–144, Apr. 2015. ISSN 2075-2180. doi: 10.4204/EPTCS.181.9.

[80] L. Moreau. Provtoolbox website, Mar. 2017. URL `https://lucmoreau.github.io/ProvToolbox/`. Date last checked: August 2024.

[81] L. Moreau, B. Clifford, J. Freire, J. Futrelle, Y. Gil, P. Groth, N. Kwasnikowska, S. Miles, P. Missier, J. Myers, B. Plale, Y. Simmhan, E. Stephan, and J. V. den Bussche. The open provenance model core specification (v1.1). *Future Generation Computer Systems*, 27(6): 743–756, Jun 2011. ISSN 0167739X. doi: 10.1016/j.future.2010.07.005.

[82] L. Moreau, B. V. Batlajery, T. D. Huynh, D. Michaelides, and H. Packer. A templating system to generate provenance. *IEEE Transactions on Software Engineering*, 44(2):103–121, Feb. 2018. ISSN 1939-3520. doi: 10.1109/TSE.2017.2659745.

[83] B. Morin, O. Barais, G. Nain, and J. Jézéquel. Taming dynamically adaptive systems using models and aspects. In *Proceedings of ICSE 2009*, pages 122–132. IEEE, 2009. doi: 10.1109/ICSE.2009.5070514.

[84] B. M. MUIR. Trust in automation: Part i. theoretical issues in the study of trust and human intervention in automated systems. *Ergonomics*, 37(11):1905–1922, Nov. 1994. ISSN 0014-0139. doi: 10.1080/00140139408964957.

[85] L. Murta, V. Braganholo, F. Chirigati, D. Koop, and J. Freire. noworkflow: Capturing and analyzing provenance of scripts. In B. Ludäscher and B. Plale, editors, *Provenance and Annotation of Data and Processes*, Lecture Notes in Computer Science, page 71–83, Cham, 2015. Springer International Publishing. ISBN 978-3-319-16462-5. doi: 10.1007/978-3-319-16462-5_6.

[86] Object Management Group. Unified Modeling Language (UML), Dec. 2017. URL `https://www.omg.org/spec/UML/2.5.1/About-UML/`. Date last checked: August 2024.

[87] Oracle. API reference for Java 11 Platform, Standard Edition, 2024. URL `https://docs.oracle.com/en/java/javase/11/docs/api/`. Date last checked: August 2024.

[88] Oracle. API reference for Java 11 Platform, Standard Edition: hashCode(), 2024. URL `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html#hashCode()`. Date last checked: August 2024.

[89] Oracle. API reference for Java 11 Platform, Standard Edition: identityHashCode(), 2024. URL `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/System.html#identityHashCode(java.lang.Object)`. Date last checked: August 2024.

[90] Oracle. API reference for Java 11 Platform, Standard Edition: Package java.lang.instrument, 2024. URL https://docs.oracle.com/en/java/javase/11/docs/api/java.instrument/java/lang/instrument/package-summary.html. Date last checked: August 2024.

[91] C. L. Owen. Design research: building the knowledge base. *Design Studies*, 19(1):9–20, Jan. 1998. ISSN 0142694X. doi: 10.1016/S0142-694X(97)00030-6.

[92] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee. A design science research methodology for information systems research. *Journal of Management Information Systems*, 24(3):45–77, Dec. 2007. ISSN 0742-1222, 1557-928X. doi: 10/cxnmc8.

[93] B. Pérez, J. Rubio, and C. Sáenz-Adán. A systematic review of provenance systems. *Knowledge and Information Systems*, 57(3):495–543, Dec 2018. ISSN 0219-1377, 0219-3116. doi: 10/gf8q84.

[94] O. J. Reynolds. Cronista (git repository), Dec. 2022. URL https://gitlab.com/sea-aston/cronista. Date last checked: August 2024.

[95] B. Ricaud. Graphexp github project, 2021. URL https://github.com/bricaud/graphexp. Date last checked: August 2024.

[96] A. Rodrigues da Silva. Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems & Structures*, 43:139–155, Oct. 2015. ISSN 14778424. doi: 10/gdtg4p.

[97] J. Rothenberg, L. Widman, K. Loparo, and N. Nielsen. The nature of modeling. *RAND Corporation*, 01 1989. URL https://www.researchgate.net/publication/277298611_The_Nature_of_Modeling. Chapter for "AI, Simulation & Modeling" Lawrence E. Widman, Kenneth A. Loparo, and Norman R. Nielsen, editors John Wiley & Sons, Inc., August 1989, pp. 75-92 (Reprinted as N-3027-DARPA, The RAND Corporation, November 1989.).

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

[99] C. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering*, 25(4):557–572, July 1999. ISSN 1939-3520. doi: 10.1109/32.799955.

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

[101] A. D. Selbst and J. Powles. Meaningful information and the right to explanation. *International Data Privacy Law*, 7(4):233–242, Nov 2017. ISSN 2044-3994. doi: 10.1093/idpl/ipx022.

[102] D. Seybold, J. Domaschka, A. Rossini, C. B. Hauser, F. Griesinger, and A. Tsitsipas. Experiences of models@run-time with emf and cdo. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering - SLE 2016*, page 46–56. ACM Press, 2016. ISBN 978-1-4503-4447-0. doi: 10.1145/2997364.2997380.

[103] Software Freedom Conservancy. Git project homepage, Feb. 2020. URL https://git-scm.com/. Date last checked: August 2024.

[104] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse modeling framework*. Addison-Wesley Educational, Boston, MA, 2 edition, 2008. ISBN 9780321331885.

[105] Sun Microsystems. Javabeans, 08 1997. URL https://download.oracle.com/otndocs/jcp/7224-javabeans-1.01-fr-spec-oth-JSpec/. Version 1.01-A Date last checked: August 2024.

[106] W. Swartout. Explaining and justifying expert consulting programs. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence (IJCAI '81)*, page 815–823, Jan. 1981. ISBN 978-1-4612-9567-9. doi: 10.1007/978-1-4612-5108-8_15.

[107] W. R. Swartout and S. W. Smoliar. On making expert systems more like experts. *Expert Systems*, 4(3):196–208, 1987. ISSN 1468-0394. doi: 10.1111/j.1468-0394.1987.tb00143.x.

[108] The Inkscape Leadership Committee. Inkscape homepage, 2022. URL https://inkscape.org/. Date last checked: August 2024.

[109] The Linux Foundation. Janus graph homepage, 2022. URL https://janusgraph.org/. Date last checked: August 2024.

[110] N. Tsakalakis, S. Stalla, T. Huynh, and L. Moreau. *A taxonomy of explanations to support Explainability-by-Design*. arvix.org, June 2022. doi: 10.48550/arXiv.2206.04438.

[111] W3C. The World Wide Web Consortium (W3C) website, 2022. URL https://www.w3.org/Consortium/. Date last checked: August 2024.

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

[113] D. A. Wheeler. sloccount homepage, 2013. URL https://sourceforge.net/projects/sloccount/. Date last checked: August 2024.

[114] R. J. Wieringa. *Design Science Methodology for Information Systems and Software Engineering*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. ISBN 978-3-662-43838-1. doi: 10.1007/978-3-662-43839-8. URL https://link.springer.com/10.1007/978-3-662-43839-8.

[115] A. F. Winfield, S. Booth, L. Dennis, T. Egawa, H. Hastie, N. Jacobs, R. Muttram, J. Olszewska, F. Rajabiyazdi, A. Theodorou, M. Underwood, R. Wortham, and E. Watson. Ieee p7001: A proposed standard on transparency. *Frontiers in Robotics and AI*, 8:665729, July 2021. doi: 10.3389/frobt.2021.665729.

[116] P. Zerva, S. Zschaler, and S. Miles. Towards design support for provenance awareness: a classification of provenance questions. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, EDBT '13, page 275–281, New York, NY, USA, Mar. 2013. Association for Computing Machinery. ISBN 978-1-4503-1599-9. doi: 10.1145/2457317.2457364. URL https://dl.acm.org/doi/10.1145/2457317.2457364.