



UNIVERSIDAD DE LA RIOJA

TESIS DOCTORAL

Título
Towards a framework for making applications provenance-aware
Autor/es
Carlos Sáenz Adán
Director/es
Beatriz Pérez Valle y Francisco José García Izquierdo
Facultad
Facultad de Ciencia y Tecnología
Titulación
Departamento
Matemáticas y Computación
Curso Académico



Towards a framework for making applications provenance-aware, tesis doctoral de Carlos Sáenz Adán, dirigida por Beatriz Pérez Valle y Francisco José García Izquierdo (publicada por la Universidad de La Rioja), se difunde bajo una Licencia Creative Commons Reconocimiento-NoComercial-SinObraDerivada 3.0 Unported.

Permisos que vayan más allá de lo cubierto por esta licencia pueden solicitarse a los titulares del copyright.

UNIVERSIDAD DE LA RIOJA

DOCTORAL THESIS

Towards a framework for making applications provenance-aware

Author:

Carlos Sáenz Adán

Supervisors:

Dr. Beatriz Pérez Valle

Dr. Francisco José García Izquierdo

*A thesis submitted in fulfillment of the requirements
for the degree of Doctor of Philosophy*

in the

**Grupo de Informática
Departamento de Matemáticas y Computación**

July 2019



**UNIVERSIDAD
DE LA RIOJA**

This work is partially supported by the Spanish Ministry of Economy and Competitiveness (Project MTM2014-54151-P and Project EDU2016-79838-P), the University of La Rioja (Grant FPI-UR-2015), the University of La Rioja together with the Government of La Rioja (Research stay grants 2016, 2017, and 2018), and Santander Bank (ATUR2016, ATUR2017, and ATUR2018).

Declaration

All uses of “we”, “our”, and “us” herein refer to Beatriz Pérez Valle, Francisco José García Izquierdo, and Carlos Sáenz Adán.

Carlos Sáenz Adán
July 2019

Acknowledgements

I would firstly like to acknowledge Luc Moreau for always being available and for giving me the opportunity to be part of his group. Undoubtedly, this thesis would not have been possible without his help. The next part of this chapter is written in Spanish since I would like for those Spanish speakers who have supported me to know how I feel.

A continuación, voy a presentar un texto que durante mucho tiempo ha sido una fuente de motivación para mí.

Están los que usan siempre la misma ropa.
Están los que llevan amuletos.
Los que hacen promesas.
Los que imploran mirando al cielo.
Los que creen en supersticiones.
Y están los que siguen corriendo cuando les tiemblan las piernas.
Los que siguen jugando *cuando se les acaba el aire*.
Los que siguen luchando *cuando todo parece perdido*.
Como si cada vez fuera la última vez.
Convencidos de que la vida misma es un desafío.

Sufren, pero no se quejan.
Porque saben que el dolor pasa, el sudor se seca, y el cansancio termina.
Pero hay algo que nunca desaparecerá:
la satisfacción de haberlo logrado.
En sus cuerpos hay la misma cantidad de músculos.
En sus venas corre la misma sangre.
Lo que los hace diferentes es su espíritu.
La determinación de alcanzar la cima.
Una cima a la que no se llega superando a los demás,
Sino superándose a uno mismo.

Este texto habla del sacrificio, del esfuerzo en conseguir una meta, y afirma que lo que nos hace diferentes es *nuestra determinación de alcanzar la cima*. ES MENTIRA. En esos momentos en los que el *aire se acaba* y *todo parece perdido*, lo que de verdad nos hace diferentes y nos empuja a alcanzar la cima es la gente que tenemos alrededor. Por eso siento la necesidad de agradecer a esas personas (una vez más) su apoyo, parte de esta tesis es suya.

Como expertillo en *provenance*, me siento en la obligación de empezar por el origen de este viaje, cuyo destino nunca fue Ítaca. Eloy, causante de mi inclusión en el “Miriam Andrés”, y donde conocí a Gadea, Félix, Rubén y Nuria. Investigación no sé si aprendí, pero de la vida, bastante. Muchas gracias a todos por acogerme desde el primer día como uno más, y sobre todo por hacerme parte de vuestro equipo. De entre ellos, me siento en la obligación de destacar a Gadea, la cual ha sido un pilar fundamental durante todo este tiempo, todo lo que pueda decir de ella se queda corto; sin lugar a duda, uno de los mejores resultados de esta tesis es su amistad.

Posteriormente llegaron Emilio, Ángel Luis y Francisco, los cuales me permitieron jugar con sus preciados RCMs y me contagiaron el germen de la investigación. Esto me llevó a dar mi primera charla en un congreso internacional; si un año antes me lo llegan a decir, no me lo hubiera creído. Muchas gracias por darme la oportunidad.

Me gustaría agradecer también a todas las personas del Departamento de Matemáticas y Computación de la Universidad de La Rioja todo su apoyo durante esta andadura, especialmente en los últimos momentos. Me resulta inimaginable el poder haber hecho la tesis en un ambiente diferente a este. En este departamento no solo he encontrado apoyo, si no que también he descubierto referentes tanto a nivel de investigación como a nivel personal. De entre todos ellos, tengo que destacar a todos aquellos profesores que me han guiado durante la carrera, los cuales van desde Judit, que impartió mi primera clase, hasta Ángel Luis, que me impuso la beca de Graduado en Informática. No solo os agradezco las enseñanzas técnicas sino también las personales, de las cuales me siento más orgulloso. Entre otras cosas, siempre tendré en cuenta que la informática es poder, que herrar es de humanos, y que una buena asignatura empieza con un buen cronograma.

Un apartado especial se merecen Francisco y Bea, mis directores de tesis. A ellos quiero mostrar mi más sincera gratitud. Cuando empecé la tesis no entendía por qué la gente se refería a sus directores como “padres” y “madres”, ahora que la he terminado lo comprendo perfectamente. Cada uno de ellos ha conseguido guiarme hasta aquí, lo cual no siempre ha sido fácil. De ellos tengo que destacar su paciencia y su implicación. Es un orgullo el haber podido aprender de ellos. Muchas gracias por todo.

También me gustaría agradecer a todos aquellos amigos que durante este tiempo se han preocupado por mí y me han animado a seguir adelante.

Que sería de mí sin mi familia, los que me han podido acompañar físicamente durante este camino y los que lo han hecho de otra manera. El término “físicamente” no ha sido escogido de forma aleatoria. A pesar de mi testarudez, siempre han estado ahí, les daba igual que fuera un pueblo perdido por Francia, que el gran Londres, ahí han estado, orgullosos de mí. Muchísimas gracias por haber hecho este camino conmigo y por haberme hecho creer en mi mismo cuando yo no lo hacía.

Finalmente, mis últimas líneas de agradecimientos son para Paula. Gracias por tu comprensión, por tu apoyo y por tu cariño. Gracias por darme aire cuando parecía que se acababa y mostrarme que todo no estaba perdido. En resumen, gracias por haber estado siempre a mi lado sacándome una sonrisa.

¡Muchas gracias a todos!

To all of you, thanks.

Abstract

Aiming at shedding light on data produced by systems, *provenance* has emerged to refer to the entire amount of information that contributes to the existence of a piece of data. The capture of provenance entails a number of benefits, from reproducibility to accountability, including assessing data quality and validity. With such tangible benefits, it is no wonder that the ability to consider the use of provenance from the early stages of the software development cycle, such as the design phase, has become critically important to support software designers in making provenance-aware applications; that is, applications with the functionality to answer questions regarding the provenance they produce. However, current approaches considering provenance during the design phase do not integrate with existing software engineering methodologies. This makes them challenging to use in practice.

UML2PROV is a novel framework intended to bridge the gap between application design and provenance design, minimising software engineers intervention and without requiring them to have provenance skills. With UML2PROV, designers can follow their preferred software engineering methodology in order to create the UML diagrams representing an application's design, and then, UML2PROV comes into play to automatically generate: (1) the design of the provenance to be generated (expressed as *PROV templates*); and (2) a software module for collecting values of interest as application is running (encoded as variable-value associations referred to as *bindings*), and which can be deployed in the application with a minimal developers intervention. The combination of the PROV templates with the bindings generates high-quality provenance ready to be exploited. Hence, UML2PROV ultimately comes to help software engineers in making applications provenance-aware.

Around UML2PROV, this thesis presents three main contributions. First, a systematic review of provenance systems, which, among other results, provides a six-dimensional taxonomy of provenance characteristics that can help researchers analyse provenance systems. Second, the conceptual definition of UML2PROV, consisting of a rigorously defined set of 17 patterns mapping UML diagrams to PROV templates, along with the requirements that any generated software module to be deployed in the application for collecting bindings must meet. This approach has been proposed aiming at minimising the intervention on software designers' and developers' modus operandi, as well as at facilitating the maintenance of the

provenance-aware applications. Third, a reference implementation of UML2PROV based on Model Driven Development techniques. This implementation automatically generates, starting from the UML diagrams of an application, both the PROV templates and the module to collect bindings. Additionally, this reference implementation provides potential users with mechanisms for managing the collection of bindings in different ways. Thus, users may choose the mechanism that best suits their needs attending to the persistence system, the run-time overhead, and storage needs, among others.

UML2PROV has also been systematically evaluated. We analysed the quality and efficiency of the provenance generated by our reference implementation, to show the benefits and trade-offs of applying UML2PROV, yielding relevant conclusions for the software engineering community. In particular, as the UML design drives both the design and capture of provenance, we study how different strategies followed during the UML design phase can affect aspects such as provenance design generation, application instrumentation, provenance capability maintenance, run-time overhead and storage needs, and quality of the generated provenance.

Resumen

Con el objeto de arrojar luz sobre los datos producidos por los sistemas, el término *provenance* ha surgido para referirse al conjunto de información que contribuye a la existencia de un dato. La captura de *provenance* conlleva una serie de beneficios que van desde la reproducibilidad hasta la responsabilidad, incluida la evaluación de la calidad y validez de los datos. A la vista de estos beneficios, resulta natural que el tener en cuenta el *provenance* desde las primeras etapas del ciclo de desarrollo del software, como la fase de diseño, haya adquirido una importancia crítica para ayudar a los diseñadores de software a hacer aplicaciones con capacidades de *provenance*; es decir, aplicaciones con la funcionalidad para responder preguntas sobre el *provenance* que producen. Sin embargo, los enfoques actuales que consideran el *provenance* durante la fase de diseño no se integran con las metodologías de ingeniería de software existentes. Esto los hace difíciles de usar en la práctica.

UML2PROV es un novedoso entorno de desarrollo orientado a cerrar la brecha entre el diseño de aplicaciones y el diseño de *provenance*, minimizando la intervención de los ingenieros del software y sin necesidad de que estos tengan conocimientos sobre *provenance*. Con UML2PROV, los diseñadores pueden seguir la metodología de ingeniería del software que prefieran para crear los diagramas UML con el diseño de la aplicación, y posteriormente, UML2PROV entra en juego para generar automáticamente: (1) el diseño del *provenance* que va a ser generado (expresado a través de *PROV templates*); (2) un módulo software para capturar valores de interés mientras la aplicación está ejecutándose (codificados como asociaciones variable-valor a las que nos referiremos como *bindings*). La combinación de los *PROV templates* con los *bindings* generará *provenance* de gran calidad, listo para ser explotado. Por lo tanto, UML2PROV ayudará a los ingenieros del software a hacer aplicaciones con capacidades de *provenance*.

En relación a UML2PROV, esta tesis presenta tres contribuciones principales. Primero, una revisión sistemática de sistemas de *provenance*, la cual, entre otros resultados, ofrece una taxonomía de seis dimensiones que puede ayudar a los investigadores en el análisis de diferentes sistemas de *provenance*. Segundo, una definición conceptual de UML2PROV, la cual consiste en (1) un conjunto de 17 patrones rigurosamente definidos y que establecen una correspondencia entre elementos UML y *PROV templates*; y (2) un conjunto de requisitos

que cualquier módulo software de captura de *provenance* debe satisfacer. Este módulo se integrará dentro de la aplicación para la obtención de *bindings*. Tercero, una implementación de referencia de UML2PROV basada en técnicas pertenecientes al Desarrollo Dirigido por Modelos. Partiendo de los diagramas UML, esta implementación genera automáticamente tanto los *PROV templates* como el módulo para obtener *bindings*. Además, esta implementación de referencia ofrece a los posibles usuarios de UML2PROV mecanismos para configurar la obtención de los *bindings* de diferentes formas. Por lo tanto, los usuarios pueden elegir el mecanismo que mejor se adapte a sus necesidades, atendiendo aspectos como el sistema de persistencia que deseen utilizar, el coste que supone disponer la captura de *provenance* en tiempo de ejecución, así como las necesidades de almacenamiento, entre otros.

UML2PROV también ha sido evaluado de una forma sistemática. Concretamente, se presenta un análisis de la calidad y la eficiencia del *provenance* generado por nuestra implementación de referencia, con el objetivo de mostrar beneficios e inconvenientes de utilizar UML2PROV. Este análisis nos ha permitido extraer conclusiones relevantes para la comunidad de ingenieros de software. En particular, como el diseño UML controla tanto el diseño como la captura de *provenance*, se ha estudiado cómo diferentes estrategias adoptadas en la fase de diseño UML pueden afectar a aspectos como: la generación del diseño del *provenance*, la instrumentación de aplicaciones, el mantenimiento de los artefactos que proveen a la aplicación de capacidad de *provenance*, el coste que supone disponer de dichos artefactos en tiempo de ejecución, así como las necesidades de almacenamiento, y calidad del *provenance* generado.

Table of contents

List of figures	xvii
List of tables	xxi
List of Abbreviations	xxiii
1 Introduction	1
1.1 Main goal and contributions of the thesis	4
1.2 Thesis overview	5
1.3 Publications	6
2 Background	9
2.1 Conceptual Background	9
2.1.1 The Unified Modeling Language	10
2.1.2 Provenance-related background	18
2.2 Implementation Background	24
2.2.1 Model-Driven Approaches	24
2.2.2 Aspect Oriented Programming	27
3 State of the art: A systematic review of provenance systems	31
3.1 Introduction	31
3.2 Towards the definition of a taxonomy of provenance systems characteristics	34
3.3 Taxonomy of provenance systems characteristics: an overview	36
3.3.1 <i>General Aspects</i> dimension	36
3.3.2 <i>Subject</i> dimension	37
3.3.3 <i>Storage</i> dimension	39
3.3.4 <i>Data Capture</i> dimension	39
3.3.5 <i>Data Access</i> dimension	40
3.3.6 <i>Non-functional Requirements</i> dimension	41

3.4	Open problems	42
3.4.1	Integration	42
3.4.2	Interoperability	43
3.4.3	Computational overhead	44
3.4.4	Querying	46
3.5	Conclusions	46
4	Conceptual definition of UML2PROV	49
4.1	Motivation for UML2PROV	50
4.2	UML2PROV architecture	51
4.2.1	Stakeholders	51
4.2.2	Key facets	52
4.2.3	How to use UML2PROV	53
4.3	From UML to PROV: the transformation patterns	54
4.3.1	Considered UML elements	55
4.3.2	Principles for the definition of UML to PROV patterns	58
4.3.3	Structure of the patterns	59
4.3.4	Overview of the defined transformation patterns	63
4.3.5	Three patterns as an example	69
4.4	Towards the generation of <i>bindings</i> . BGM features and requirements	84
4.5	The consistency between templates and bindings	88
4.6	Conclusions	92
5	Implementation of UML2PROV	95
5.1	Introduction	96
5.2	A Proof of concept for implementing UML2PROV	97
5.2.1	Automatization of the transformation patterns: an XSLT-based approach	97
5.2.2	Automatization of the generation of the BGM: a <i>Proxy-pattern</i> and XSLT-based approach	100
5.3	Reference implementation	104
5.3.1	Automatization of the transformation patterns: an ATL and XPand-based approach	104
5.3.2	Automatization of the generation of the BGM: an event-based and an XPand-based approach	108
5.3.3	Fulfilment of BGM requirements	116
5.4	Conclusions	117

6	Evaluation	119
6.1	Introduction	119
6.2	GelJ and the Design Strategies	120
6.2.1	UML design of GelJ	122
6.2.2	Strategies for obtaining GelJ design	123
6.3	Analysing the benefits and trade-offs of using UML2PROV	127
6.3.1	Aspect 1: Generation of the provenance design	127
6.3.2	Aspect 2: Instrumentation of the application	128
6.3.3	Aspect 3: Maintenance of provenance capabilities	129
6.3.4	Aspect 4: Run-time overhead and storage needs	130
6.3.5	Aspect 5: Quality of provenance	134
6.4	Conclusions and discussion	135
7	Conclusions and future work	139
7.1	Summary of results	139
7.1.1	A systematic review of provenance systems	140
7.1.2	A conceptual definition of UML2PROV	140
7.1.3	An implementation of UML2PROV	141
7.2	Limitation of the study and future work	143
	Conclusiones	147
	References	149
	Appendix A Provenance systems analysed in the systematic review	165
	Appendix B Description of the transformation patterns used in this memoir	167
	Appendix C Sequence of interactions with GelJ	195

List of figures

2.1	The taxonomy of structure and behaviour diagrams [1]	10
2.2	UML sequence diagram showing the interactions between objects of classes Student, Seminar, and Course	11
2.3	An excerpt of the UML class diagram for the University example	13
2.4	UML state machine diagram for class Seminar	15
2.5	The definition of the <i>stereotype</i> named <code>Creator</code> together with an example of application of such a stereotype	17
2.6	Organization of the PROV family of documents [2]	18
2.7	The core concepts of PROV-DM together with their relations	19
2.8	A graph notation for PROV elements	21
2.9	A PROV document in PROV-N together with its graphical representation	21
2.10	The architecture of the templating approach proposed in [3]	22
2.11	A graphical illustration of a <i>template</i>	22
2.12	An example of <i>bindings</i> for the <i>template</i> of Figure 2.11	23
2.13	Simplified model driven process inspired by the one presented in [4]	25
2.14	An AspectJ <i>aspect</i> implementing a crosscutting concern	29
3.1	Our taxonomy of Provenance Systems Characteristics	35
4.1	The architecture of UML2PROV	52
4.2	This chart is focused on those patterns that rule the transformation of SqDs	64
4.3	This chart is focused on those patterns that rule the transformation of CDs	68
4.4	This chart is focused on those patterns that rule the transformation of SMDs	69
4.5	An excerpt of the SqD showing the interaction between <code>Student</code> and <code>Seminar</code> when a student enrolls in a seminar	70
4.6	PROV template obtained by applying <i>SeqP2</i> to the SqD depicted in Figure 4.5	74
4.7	An excerpt of a CD (together with an equivalent representation) showing those elements that are involved in the enrolment of a student into a seminar	75

4.8	PROV template obtained by applying <i>CIP10</i> to the CD highlighted in Figure 4.7	79
4.9	An excerpt of an SMD showing how the enrolment of a student into a seminar affects the state of the objects of the class <code>Seminar</code>	80
4.10	PROV template obtained by applying <i>StP3</i> to the SMD depicted in Figure 4.9	83
4.11	Possible set of bindings obtained from the execution of the <code>enrolStudent</code> operation	88
4.12	Three circles enclosing the variables that identify the elements in the PROV templates generated by applying <i>SeqP2</i> , <i>CIP10</i> , and <i>StP3</i>	89
4.13	PROV documents obtained by expanding the PROV templates shown in Figures 4.6, 4.8 and 4.10 with the set of bindings in Figure 4.11. We do not show the PROV attributes to avoid overburdening the figure.	90
4.14	PROV document obtained by merging the documents shown in Figure 4.13	91
5.1	<i>Proxy-pattern</i> architecture	101
5.2	Excerpts of Java code depicting the construction of an object as usual (without provenance capture), and the construction of an object by following the <i>Proxy-pattern</i> approach (with provenance capture)	103
5.3	Detailed MDD-based implementation of the templates generation process	105
5.4	XPand templates defined for each <code><document></code> and <code><entity></code> in the <i>template models</i>	107
5.5	PROV activity in PROV-N [5] with different types of variables	108
5.6	UML Class diagram depicting our reference implementation for a BGM	109
5.7	Structure overview of a reference implementation of the <i>BGMEventInstrumenter</i> in AspectJ	111
5.8	Graphical representation of the three defined configurations	113
5.9	Detailed MDD-based implementation of the BGM for an application	115
6.1	Workflow of GelJ's <i>experiment wizard</i>	121
6.2	Number of instructions executed for collecting provenance data organised by strategies	131
6.3	Information about run-time overhead (%)	132
6.4	Information about storage needs (MB)	133
B.1	UML representation that models the context given by <i>SeqP1</i>	168
B.2	PROV template generated from the UML representation used in <i>SeqP1</i> (Figure B.1)	169
B.3	UML representation that models the context given by <i>SeqP2</i>	173

B.4	PROV template generated from the UML representation used in <i>SeqP2</i> (Figure B.3)	173
B.5	UML representation that models the context given by <i>CIP10</i>	178
B.6	PROV template generated from the UML representation used in <i>CIP10</i> (Figure B.5)	178
B.7	UML representation that models the context given by <i>CIP10</i> , including Output Parameters	182
B.8	PROV template generated from the UML representation used in <i>CIP10</i> , including Output Parameters (Figure B.7)	182
B.9	UML representation that models the context given by <i>StP1</i>	184
B.10	PROV template generated from the UML representation used in <i>StP1</i> (Fig- ure B.9)	185
B.11	Excerpt of a UML State Machine diagram locating the UML elements from <i>StP1</i> in a <i>simple composite state</i>	187
B.12	PROV template generated from the UML diagram in Figure B.11	187
B.13	UML representation that models the context given by <i>StP3</i>	190
B.14	PROV template generated from the UML representation used in <i>StP3</i> (Fig- ure B.13)	190
B.15	Excerpt of a UML State Machine diagram locating the UML elements from <i>StP3</i> in a <i>simple composite state</i>	192
B.16	PROV template generated from the UML diagram in Figure B.15	193
C.1	Sequence of interactions with GelJ used in the evaluation of UML2PROV	196

List of tables

2.1	Prefixes and Namespaces used in this document	20
3.1	List of the reviewed systems together with their references	34
4.1	Extension of the taxonomy of UML <i>class' operations</i> given in [6]	66
4.2	Matching between the UML elements identified in <i>SeqP2</i> and the UML elements from Figure 4.5	70
4.3	Matching between the UML elements identified in <i>CIP10</i> and the highlighted UML elements from Figure 4.7	76
4.4	Matching between the UML elements identified in <i>StP3</i> and the UML elements from Figure 4.9	81
5.1	An excerpt of the XSLT template focusing on the implementation of <i>StP1</i>	98
5.2	An excerpt of the ATL rule focusing on the implementation of <i>StP1</i>	106
6.1	Overview of tasks performed to obtain the UML design in each strategy	124
6.2	Questions identified from Q1 to Q9 raised by GelJ users, together with GelJ classes involved in answering those questions	125
6.3	Information about PROV templates and BGMs generated per strategy	128
6.4	For each strategy, it is indicated whether questions Q1-Q9 of Table 6.2 can be answered completely, partially or cannot be answered	134
A.1	A brief description of the surveyed systems (I)	165
A.2	A brief description of the surveyed systems (II)	166
B.1	Set of patterns used to illustrate this memoir	167

List of Abbreviations

AOP	Aspect-Oriented Programming
ATL	ATLAS Transformation Language
BGM	Bindings Generation Module
CD	Class Diagram
CIP_{<i>i</i>}	Transformation Pattern number <i>i</i> for Class Diagrams
JSON	JavaScript Object Notation
JSON-LD	JavaScript Object Notation for Linked Data
M2M	Model to Text
M2T	Model to Model
MDA	Model Driven Architecture
MDD	Model Driven Development
OMG	Object Management Group
OOP	Object Oriented Programming
OS	Operating System
OWL	Web Ontology Language
OWL2	Web Ontology Language version 2
PIM	Platform Independent Model
PrIME	Provenance Incorporation Methodology
PROV	Standard for the representation of provenance
PROV-AQ	PROV Access and Query.
PROV-DC	PROV Dublin Core
PROV-DM	PROV Data Model
PROV-N	PROV Notation
PROV-O	PROV Ontology
PROV-SEM	PROV Semantics
PSM	Platform Specific Model
RCM	References-enriched Concept Map
RDF	Resource Description Framework
SeqPi	Transformation Pattern number <i>i</i> for Sequence Diagrams
SMD	State Machine Diagram
SqD	Sequence Diagram
StPi	Transformation Pattern number <i>i</i> for State Machine Diagrams

UML	Unified Modeling Language
UML2PROV	A framework for making applications provenance-aware
W3C	World Wide Web Consortium
WfMS	Workflow Management Systems
XMI	XML Metadata Interchange
XML	Extensible Markup Language

Chapter 1

Introduction

Algorithms are everywhere. They constitute the basis of whichever computational system. They could be defined as a self-contained step-by-step set of operations that computers or other ‘smart’ devices carry out to perform calculation, data processing, and automated reasoning tasks [7]. They have five important features as stated in [8]: (1) *finiteness*, “an algorithm must always terminate after a finite number of operations are executed”; (2) *definiteness*, “each step of an algorithm must be precisely defined, the actions to be carried out must be rigorously and unambiguously specified for each case”; (3) *input*, “an algorithm must have zero or more inputs: quantities that are given to it initially before the algorithm begins, or dynamically as the algorithm runs”; (4) *output*, “an algorithm has one or more outputs: quantities that have a specified relation to the inputs”; and (5) *effectiveness*, “the operations must be sufficiently basic that they can in principle be done exactly and in a finite length of time.”

Having said this, it is a widespread belief that algorithms are objective because they strictly rely on data. However, this belief is not entirely true, because of the fact that algorithms are created and maintained by people. There are even more sensitive cases, such as machine learning algorithms, as they adapt their behaviour relying upon data usually provided by people, what makes them highly dependent on those people. One example of this is the chatbot released by Microsoft Corporation via Twitter on March 23, 2016 [9]. After a day learning from the user’s comments, the chatbot started to tweet misogynistic and racist comments. Additionally, to make matters worse, there is also growing evidence that algorithms may be opaque [7], which means that it is impossible to determine if their resulting data are skewed or wrong. Some of the factors leading to this are technical (e.g., the algorithm is not easily explainable), economic (e.g., trade secrets or it is not cost-effective), and social (e.g., privacy policy) [7].

To address this challenge, the term *provenance* has emerged to refer the entire amount of information, comprising all the elements, and their relationships, that contribute to the existence of a piece of data [10]. Provenance provides a great number of benefits such as verifying a product, result reproducibility, sharing and reuse of knowledge, or assessing data quality and validity. Throughout this thesis, we will use the term *provenance-aware* application to refer to those applications that have the functionality to answer questions regarding the provenance they produce [11]. With such tangible benefits, it is no wonder that in recent years, research on provenance has grown exponentially, and has been applied to a wide range of different scientific disciplines. An example of this interest is ACM U.S. Public Policy Council (USACM)'s statement [7], which includes seven principles for algorithmic transparency and accountability. Among these principles, it is worth noting the *data provenance* principle, which is described as follows:

“A description of the way in which the training data was collected should be maintained by the builders of the algorithms, accompanied by an exploration of the potential biases induced by the human or algorithmic data-gathering process. Public scrutiny of the data provides maximum opportunity for corrections. However, concerns over privacy, protecting trade secrets, or revelation of analytics that might allow malicious actors to game the system can justify restricting access to qualified and authorized individuals.”

This interest in provenance is not new. In 2006 the *first provenance challenge* [12] was launched with the aim of establishing an understanding of the capabilities of available provenance-aware applications and, in particular, to grasp concrete details about the representations of the provenance used, capabilities for answering concrete provenance-related questions, and which aspects are considered by each system in the scope of provenance. Seventeen teams responded to the challenge and submitted their provenance-aware applications addressing it from different perspectives. After the first provenance challenge, three more challenges were also launched seeking to the interoperability of provenance-aware applications. The result was the definition of the Open Provenance Model (OPM) [13], which demonstrated that provenance interoperability was achievable and thus mature enough to begin standardisation by an organization like W3C [14]. The result of such a standardization was the W3C PROV standard [2] for the representation of provenance. Since the PROV standard's aim is the interoperable exchange of provenance information, toolkits supporting PROV, such as ProvPy [15] or ProvToolbox [16] have been facilitating the software engineer's task of creating, storing, reading and exchanging provenance. Similarly, the PROV-Template [3] approach is a development for helping design the provenance data to be produced and making easier its generation in a PROV compatible format. This approach allows the structure of

provenance to be described declaratively: a *template* is a document containing placeholders (referred to as *variables*). An *expansion algorithm* generates the provenance to be exploited by instantiating a *template* with values, which are contained in *bindings* associating *variables* with concrete values. Although this approach reduces the development and maintenance effort, separating responsibilities between software and provenance designers, it still requires designers to be knowledgeable in provenance. Additionally, we note that in contrast to the aforementioned toolkits, the PROV-Template approach helps decide what information should be included in the generated provenance by means of the templates; however, as the toolkits do, it does not show how software should be designed to allow for its capture. Therefore, the ability to consider the use of provenance, specially in early stages of the software development cycle, such as the design phase, has become critically important to support the software designer in making provenance-aware applications. To address this challenge, the Provenance Incorporation Methodology (PrIME) [11] emerged for adapting applications to be provenance-aware. However, although the application of this methodology showed promising results in a variety of areas [11], PrIME is standalone. It does not integrate with existing software engineering methodologies, which makes it challenging to use in practice.

In contrast, in the realm of software engineering, design techniques have been proposed to shorten the development time of software products, as well as to increase their quality. Among such techniques, the Unified Modeling Language (UML) [1] is widely accepted as the de-facto method for designing, or even simply describing, object-oriented software systems. However, although UML comprises many types of diagrams in order to describe different perspectives of a system (for instance, structural or behavioural information), it lacks specific support for provenance. Specifically, UML does not provide specific tools to design systems that can answer questions such as the activity that lead to specific resulting data, or the elements that were involved in its creation. In fact, developing software applications by including provenance in the design phase can entail significant changes to an application design [11]. This is a cumbersome task for the software engineers, since they have to be knowledgeable about provenance in addition to having to deal with complex diagrams, and, what is worse, to maintain an application's provenance-specific code.

In short, there exists a gap between application design and provenance design, which may result in applications generating provenance that is not aligned with what the application actually does, or that does not fit for its purpose.

1.1 Main goal and contributions of the thesis

The main goal of this thesis is the definition of an overall framework, coined as *UML2PROV*, which allows software engineers to bridge the gap between application design and provenance design. This framework is intended to provide any application with the ability to generate provenance information, by means of a fully automatic process that takes as source the UML design of the application and creates the artefacts needed to generate provenance, without requiring the manual application's code instrumentation. Hence, this framework ultimately comes to help software engineers in making provenance-aware applications, minimising their intervention and without requiring them to be knowledgeable about provenance.

We have achieved this ambitious goal through the following three main contributions:

- *A systematic review of provenance systems.* We have undertaken a thorough analysis of the literature, providing a broad overview of the existing provenance systems, identifying the characteristics such systems are expected to have, and analysing the techniques and methods proposed to address them. Based on the reviewed works, we have defined a taxonomy of provenance characteristics attending to: general aspects, subject, storage, data capture, data access, and non-functional requirements. The proposed taxonomy can help other researchers analyse provenance systems. Ultimately, the results provided in this phase helped us identify and contextualise the open problems of the provenance research that motivated the UML2PROV approach and that served as the basis for this thesis.
- *A conceptual definition of UML2PROV.* We have established the conceptual aspects regarding the definition of the UML2PROV framework. Considering UML diagrams as key elements of an application design, and PROV templates as cornerstone elements of the provenance design, we have proposed a feasible procedure to obtain the two key elements of the PROV-Template approach (*templates* and *bindings*). As for the creation of *templates*, we define a set of transformation patterns that ultimately associates commonly appearing structures in UML diagrams with PROV templates. Regarding the creation of *bindings*, we provide each application with a concrete module that collects provenance data in the form of bindings. This module has to be deployed in the application, without requiring more intervention from developers than choosing the configuration for that module. The disparate nature of applications prevents us from establishing a generic module for all the applications; instead, we have provided a set of requirements each module for provenance capture must fulfil. The conceptual definition of the transformations from UML to PROV templates specifying the provenance design of an application, together with the requirements a module for capturing provenance must fulfil, lay the foundations for

making any application provenance-aware with UML2PROV. This conceptual definition of UML2PROV serves as the basis for future implementations.

- A *UML2PROV reference implementation*. We have provided a reference implementation of our conceptual proposal for UML2PROV. Using this implementation, software engineers are allowed to follow their preferred software engineering methodology to create the UML diagrams for their application and, then, UML2PROV comes into play to take those UML diagrams as a starting point to *automatically* generate the artefacts to make such an application provenance-aware: (1) the PROV templates with the provenance design, and (2) the module for collecting bindings. This implementation prevents software engineers not only from being knowledgeable about provenance, but also from the need of including the provenance aspects into the application's design diagrams (usually making them complex and unreadable). It also prevents the provenance generation code from being interweaved between the application code. Both achievements help designers and developers make transparent the generation of provenance and make it easier the maintenance of provenance-aware applications.

Finally, aiming at showing the benefits and trade-offs of applying UML2PROV for software engineers seeking to make applications provenance-aware, we have performed a systematic evaluation of our UML2PROV reference implementation based on quantitative data and qualitative arguments. Although our proposal is intended to be applied to any application no matter it is newly developed (thus, probably, UML designs are available) or it is a legacy application (thus, it can lack UML designs), we have considered the worst-case scenario, evaluating our proposal by making provenance-aware a legacy application not initially designed with UML. Additionally, in order to make a more comprehensive assessment of our proposal, we have applied different reverse engineering techniques to obtain several UML application designs with different levels of detail. This has allowed us to study how those levels of detail in UML diagrams affect aspects such as provenance design generation, application instrumentation, provenance capability maintenance, run-time overhead and storage needs, and quality of the generated provenance. Likewise, in the evaluation, we have also tested different configurations of the generated module to collect bindings that have shown us how they affect run-time overhead and storage needs.

1.2 Thesis overview

This thesis is organised as follows:

- Chapter 2 introduces some background information that we have considered of interest for understanding the subsequent chapters of this memoir. Concretely, we distinguish two blocks related to the two main contributions of this thesis. The first block introduces background information with regard to the conceptual definition of UML2PROV, whereas the second block provides relevant information about the implementation of UML2PROV.
- Chapter 3 plots the landscape of published studies in the field of provenance by means of a thorough overview of provenance systems. This chapter is twofold. First, we provide a consistent background of provenance concepts and techniques that helps readers grasp this memoir. Second, we justify the open research problems in the provenance field that served as a basis for this thesis.
- Chapter 4 explains the conceptual definition of UML2PROV. We start by motivating the need of this framework relying upon the aforementioned open problems. Next, we describe the architecture we propose for UML2PROV, and finally, we explain its foundations together with its contributions and benefits drawn from its usage.
- Chapter 5 provides a reference implementation of UML2PROV, that, following the aforementioned conceptual definition, automatically generates those elements (the templates and the module for collecting bindings) needed to obtain provenance.
- Chapter 6 includes an evaluation of UML2PROV that shows the benefits and trade-offs of using UML2PROV attending to five aspects regarding provenance. Additionally, this evaluation also shows the implications of using (1) UML diagrams with different levels of detail, and (2) different configurations for creating bindings during the execution of the application.
- Chapter 7 outlines various avenues for future work and concludes the thesis.

1.3 Publications

The work in this thesis is derived from the following publications:

- Carlos Sáenz-Adán, Francisco J. García-Izquierdo, Ángel Luis Rubio, Eduardo Sáenz-de-Cabezón Irigaray, Emilio Rodríguez Priego, Oscar Díaz: *A Tool for Management of Knowledge Dispersed throughout Multiple References*. ICISOFT-PT 2015: 79-86
- Beatriz Pérez, Julio Rubio, Carlos Sáenz-Adán: *A systematic review of provenance systems*. In *Knowledge and Information Systems* 57(3): 495-543 (2018)

-
- Carlos Sáenz-Adán, Beatriz Pérez, Trung Dong Huynh, Luc Moreau: *UML2PROV: Automating Provenance Capture in Software Engineering*. SOFSEM 2018: 667-681
 - Carlos Sáenz-Adán, Luc Moreau, Beatriz Pérez, Simon Miles, Francisco J. García-Izquierdo: *Automating Provenance Capture in Software Engineering with UML2PROV*. IPAW 2018: 58-70
 - Carlos Sáenz-Adán, Beatriz Pérez, Francisco J. García-Izquierdo, Luc Moreau: *Integrating Provenance Capture and UML with UML2PROV: Principles and Experience*. Submitted for publication in IEEE Transactions on Software Engineering

In addition, results of this thesis were used in the work published in:

- Michael A. C. Johnson, Luc Moreau, Adriane Chapman, Poshak Gandhi, Carlos Sáenz-Adán: *Using the Provenance from Astronomical Workflows to Increase Processing Efficiency*. IPAW 2018: 101-112

Chapter 2

Background

This chapter provides background information required to understand the overall thesis work. More specifically, in the line with the main contributions of this thesis – to provide (1) a conceptual definition of UML2PROV, and (2) a reference implementation conforming to such a conceptual definition – this chapter is divided into two main sections. The first section introduces the background information that is needed to understand the conceptual definition of UML2PROV, i.e., UML and concrete provenance-related information. The second section presents implementation-related background: the Model Driven Development approach and the Aspect Oriented Programming paradigm.

We note that this chapter’s content is not intended to be exhaustive, but to provide sufficient information to grasp our proposal, as well as references for the reader to explore further. If the reader is familiar with any subject presented here, he can skip the corresponding subsection since this chapter does not contain any new research.

Finally, it is worth remarking that to illustrate this chapter, in particular, and the thesis, in general, we will use a running example (which we will refer to as *University example*). This running example is concerned with a system that manages the enrolment and attendance of students to seminars of a specific University course. We have chosen this example for being intuitive, thereby it does not require much effort from the readers for understanding it.

2.1 Conceptual Background

As the UML2PROV acronym suggests, there are two topics on which the conceptual background necessary to understand this thesis must be provided: UML and provenance. Here, we first introduce the Unified Modeling Language, and then we provide information regarding concrete provenance aspects used in this thesis.

2.1.1 The Unified Modeling Language

Our approach is intended to make applications provenance-aware taking the design of such applications as primary source. Concretely, we advocate using designs specified by UML diagrams [1]. UML is a general purpose modeling language developed by the Object Management Group (OMG). More specifically, UML has been widely accepted and supported by the software industry as the de-facto standard language for specifying, constructing, and documenting the artefacts of object-oriented systems [1]. One of its main advantages is that it is a general-purpose language, and therefore, it can be used to model a wide range of problems in different contexts of application. The fact that UML is a so general purpose modeling language could be considered both a strength and a weakness. Particularly, a language so general and of such a broad scope as UML may limit its effectiveness in modeling some specific domains. In these situations, it would be desirable to have at ones disposal specialized languages which may be more appropriate to express specific concepts of the system to model. Faced with this situation, UML provides different types of approaches that allow customizing and extending its own syntax and semantics to be adapted to certain application domains. Among such approaches, we note UML *stereotypes* for being the basic customization and extension mechanisms.

UML is defined by the MOF UML metamodel, which gives a systematic description of the syntax and semantics of the UML. UML provides two major types of *UML* diagrams (*structural* and *behavioural*) used to diagrammatically model the structure and the behaviour of object-oriented software systems (see Figure 2.1). While *structural* diagrams represent the static structure of a system, *behavioural* diagrams capture the behavioural features, including aspects concerning its runtime execution. Among the types of UML diagrams, in this research

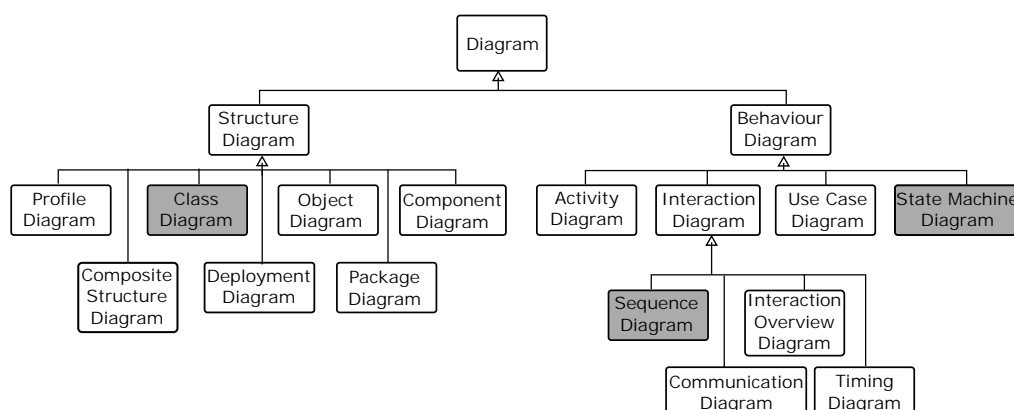


Fig. 2.1 The taxonomy of structure and behaviour diagrams [1]. In grey, the diagrams of interest for this thesis.

work we have focused on those that not only have a strong relation with provenance (later we will explain the reasons), but that also are mostly used by *software designers* [17]. These diagrams are UML Sequence Diagrams (SqDs) and UML State Machine Diagrams (SMDs), which are two types of *behavioural* diagrams, and UML Class Diagrams (CDs), which are part of the *structural* diagrams category. Next, we will give an insight into these UML diagrams focusing on those elements that will play a key role in this work. Readers interested in a deeper explanation about the elements included in these types of diagrams are referred to the UML specification [1].

UML Sequence diagrams

Sequence diagrams (SqDs) are used to model the interactions among collaborating objects and the exchange of information between them. They mainly consist of the following elements: *lifelines*, *messages*, and *execution specifications*. The SqD in Figure 2.2 describes the interactions between a student, a seminar, and a course in our University example.

Each object participating in an interaction is represented as a *lifeline*. Each *lifeline* is normally shown by a rectangle, containing the object's name, followed by a dashed line which extends down vertically, indicating the object's existence over some period of time. Examples of *lifelines* are `st:Student`, `s:Seminar` and `c:Course`, depicted in the Figure 2.2.

A *message* specifies a communication from a *sender* to a *receiver lifeline*, in which a sender makes a request for either an *operation* call or *signal* reception by a receiver. While a *signal* refers to a communication between objects in which a reaction is asynchronously triggered in the receiver without a reply, an *operation* call can be either synchronous or asynchronous. A *message* is represented by a horizontal line with an arrow pointing in the direction in which the *message* is sent (e.g., `enrolStudent(st)` and `includeAssociatedSeminar(s)` in Figure 2.2). UML distinguishes six sorts of messages reflecting different kinds of communications (they are specified by the shape of the arrow head) [1]:

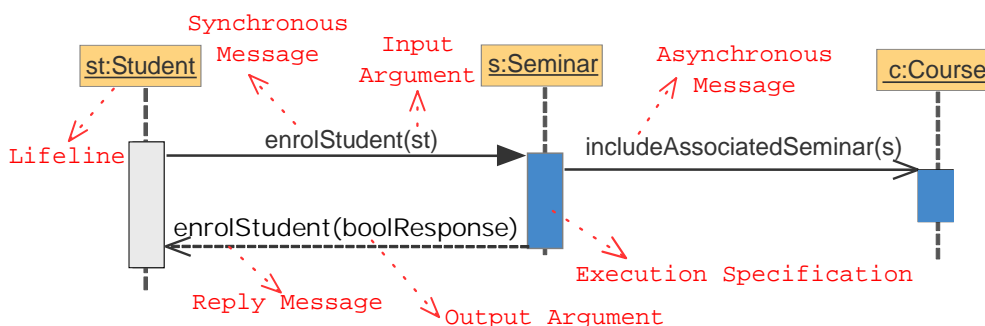


Fig. 2.2 UML sequence diagram showing the interactions between objects of classes Student, Seminar, and Course. The name of the UML elements of interest are in red courier font.

- A *synchronous message* (filled arrow head). The message was generated by a synchronous call to an operation. The sender waits for a response.
- An *asynchronous message* (open arrow head). The message was generated by an asynchronous call to an operation. The sender continues immediately after sending it, without waiting for a response.
- A *reply message* (dashed line with either an open or filled arrow head). The message represents the return from a synchronous call to an operation.
- An *object creation message* (dashed line with an open arrow head). This is the message designating the creation of another lifeline object.
- An *object deletion message* (ends in a cross *X* at the bottom of the lifeline). The message designates the termination of the receiver lifeline.
- An *asynchronous signal* (open arrow head). The message represents the asynchronous sending and reception of an instance of the signal.

The name of the *message* must be the same as the name of the referenced signal or operation. If the *message* signature is a signal, the *arguments* of the *message* correspond to the attributes of the signal. Otherwise, if the *message* signature is an operation, two options are possible. When it is a *synchronous* or *asynchronous message*, the *arguments* of the *message* correspond to the *in* and *inout parameters* of the operation, in the same order. When it is a *reply message*, the *arguments* of the *message* refer to the *out*, *inout* and *return parameters* of the operation, in the same order. *Parameters* have a direction: *in*, *inout*, *out* and *return*. The *in* direction indicates that the *parameter* values are passed by the caller. The *inout* direction means that *parameter* values are passed in by the caller and (possibly different) values are passed out to the caller. The *out* direction indicates that *parameter* values are passed out to the caller. Finally, the *return* direction means that *parameter* values are passed as return values back to the caller.

An *execution specification* is a fragment which represents a period during which a participant's *lifeline* is executing a unit of behaviour or action (such as an operation). *Execution specifications* can be represented by a bar reached by a *message* at its top. In Figure 2.2, we depict in blue two *execution specifications* representing the executions of the `enrolStudent` and `includeAssociatedSeminar` operations, respectively. In the former case, the *execution specification* is started by a *synchronous message* and is finished by a *reply message*, while in the latter, it is started by an *asynchronous message*.

In addition to the previous elements, which appear in Figure 2.2, there are also common UML elements that are worth mentioning. For instance, *combined fragments* are used to model various control structures explicitly, which permit to describe a number of possible execution paths compactly and precisely. The *combined fragments* are represented by a rectangle with interactions inside, and with the *operator* type specified by the respective keyword in a small pentagon in the upper left corner of this rectangle. The UML specification provides 12 different types of *operators* [1]. For instance, the *alt operator* shows an alternative interaction, whereas the *opt operator* indicates that an interaction is optional. Alternatively, the *loop operator* specifies an iterative interaction.

UML Class diagrams

Class diagrams (CD) are without a doubt the most widely used UML diagrams [18]. They are utilized to model the static structure of a system, therefore describing the elements of the system and the relationships between them. The main elements are *classes*, which contain *attributes* and *operations*, and the *relationships* between the *classes*. In Figure 2.3 we show an excerpt of the class diagram specifying the structure of the University example.

In UML, the objects are classified attending to their characteristics and behaviour by means of *classes* (e.g., *Student*, *Seminar*, and *Course* in Figure 2.3). Concretely, the characteristics are specified by *attributes* (e.g., *identifier* and *studentName* in *Student*), whereas the specific behaviour that can be triggered on individual objects is specified by the *operations* (e.g., *setName* in *Student*). As for the notation of *classes*, they are represented by a rectangle that can be subdivided into multiple compartments. The first one must contain the name of the class, and it is mandatory. The second and third may contain the *attributes* and *operations*,

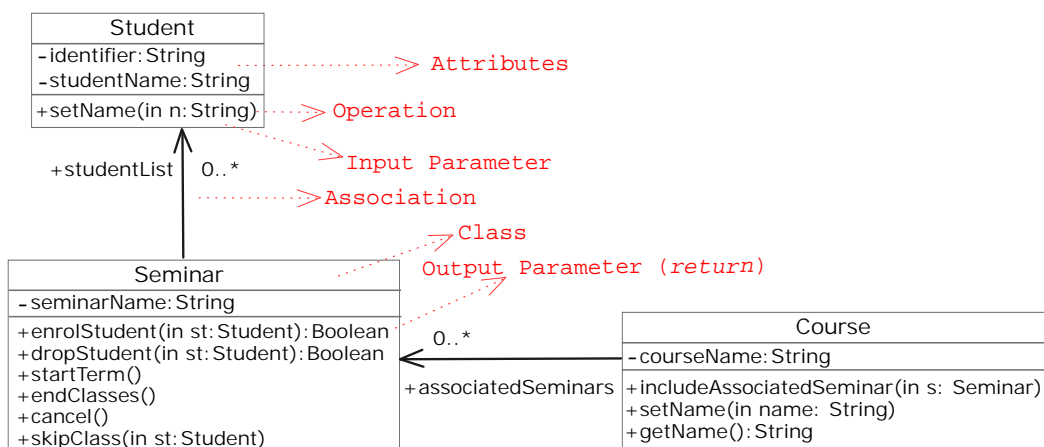


Fig. 2.3 An excerpt of the UML class diagram for the University example. The name of the UML elements of interest are in red courier font.

respectively. On the one hand, the *attributes* allow storing information that is known for all instances but that generally have different specific values for each instance. An *attribute* has at least a name, and its type may be specified after the name using a colon. The multiplicity of an *attribute* indicates how many values an *attribute* can contain, and it is shown as an interval enclosed by square brackets in the form $[\text{min}, \text{max}]$, whereby *min* and *max* are natural numbers indicating the lower and upper limits of the interval, respectively. On the other hand, the *operations* are characterised by their name and their *parameters*. When an *operation* is called, the behaviour associated with it is executed. The name of the *operation* is followed by a list of *parameters* in parentheses, which may be empty. Each *parameter* has a name (mandatory), type, multiplicity, and a direction of type *in*, *inout*, or *out*. In case the *operation* has a *parameter* with *return* direction, it must be specified after the list of parameters using a colon and followed by the type of the return value (see operation `enrolStudent` in Figure 2.3).

The *associations* between *classes* model possible relationships, known as *links*, between instances of the *classes*. The *associations* are shown as edges between *classes*. For instance, the CD in Figure 2.3 shows one association between `Seminar` and `Student`, and another one between `Course` and `Seminar`. If the edge is directed (i.e., at least one of the two ends has an open arrowhead), navigation from an object to its partner is possible and the object can access its partner's visible *attributes* and *operations*. Due to the fact that *associations* are realised as references to the associated objects, an *association* can also be represented as an *attribute* with the appropriate multiplicity, whereby the type of the *attribute* is the *class* of the partner objects (page 206 in [1]). Similarly to multiplicities of *attributes* and *parameters*, the multiplicities of *associations* are given as an interval in the form $\text{min}.. \text{max}$, specifying the number of objects that may be associated with exactly one object of the opposite side. Finally, it is important to note a concrete form of *associations* called *aggregations*, used to represent the relationships between a whole and its parts. UML differentiates between two types of *aggregations*: *shared aggregations* and *composite aggregations*. The *shared aggregation* means that parts also exist independently of the whole. The *composite aggregations* express that a specific part can only be contained in at most one composite object at one specific point in time. It means that if the composite object is deleted, its parts are also deleted. For example, in our University example, each study program could be considered to be made up of any number of courses (which will be represented by a *shared aggregation* between a `StudyProgram` class, representing the whole, and the `Course` class, representing the parts). Similarly, a composition association could be given among class `Building`, being the whole, and `LectureHall`, being the parts, for example [18]. The differentiation between *composite aggregation* and *shared aggregation* is indicated by a solid diamond for a *composite aggregation* and a hollow diamond for a *shared aggregation*.

The *generalization relationship* between two *classes* is used to express that the characteristics (*attributes* and *operations*) and *associations* that are specified for a general *class* (referred to as *superclass*) are passed on to its *subclasses* [18]. A *generalization relationship* is represented by an arrow with a hollow, triangular arrowhead from the *subclass* to the *superclass* (for example, in our example, a superclass `Person` could be considered to represent a generalization of the subclasses `Student` and `Professor`).

Finally, we would like to comment on another UML elements that may appear in Class diagrams: *abstract classes* and *interfaces*. *Classes* that cannot be instantiated themselves are modeled as *abstract classes* which are used to highlight common characteristics of the *classes* related to it [18]. Similarly to the *abstract class*, an *interface* also does not have an implementation or any direct instances. An *interface* represents a contract so that the *classes* that implement an *interface* obligate themselves to provide the behaviour specified by the *Interface* [18]. As for their graphical representation, *abstract classes* are written in italic font or indicated by the keyword `{abstract}` before their name. Conversely, an *interface* is denoted like a class but with the additional keyword `<interface>` before the name.

UML State Machine diagrams

State Machine diagrams (SMD) specify the discrete behaviour of individual elements of a system. They are basically made up of *states*, *pseudostates*, and *transitions*. The SMD in Figure 2.4 depicts the set of *states* an object of the class `Seminar` can go through during its lifetime.

A *state* basically denotes a situation of an object during which some condition holds. For instance, based on the SMD from Figure 2.4, objects belonging to class `Seminar` may be in the *states* performing (enrolling or studying), and examining. A *state* may have *internal*

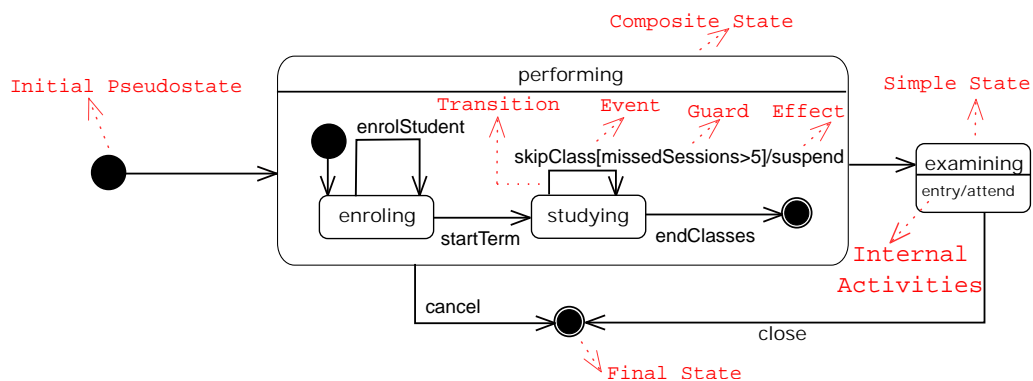


Fig. 2.4 UML state machine diagram for class `Seminar`. The name of the UML elements of interest are in red courier font.

activities, whereby an *activity* may consist of multiple actions [18]. If the *activity* is specified after the keyword *entry*, such an *activity* is executed when the object enters the *state* (e.g., examining); as opposed to an *exit activity*, which is executed when the object exists the *state*. An *activity* preceded by the keyword *do* is executed while the object remains in the *state*. All the *activities* are always specified with a prepended forward slash that clearly identifies it as an *activity*.

There are three types of *states*: *simple*, *composite*, and *submachine states*. *Simple states* are characterised by not having nested *states*, also known as *substates* (e.g., enrolling, studying, or examining). Conversely, *composite states* can hold a connected set of *substates*, *pseudostates* and *transitions* (e.g., performing). *Composite states* are divided into *orthogonal composite states*, which model concurrent behaviours where several *substates* are active simultaneously, and *simple composite states*, which specify that only one of their *substates* must be active at same time (this is the case of performing in Figure 2.4). A *submachine* is a special kind of *state* that refers to another defined State Machine diagram. *States* are represented by rectangles with rounded corners. A particular *state* referred to as *final state* represents the completion of an *activity* within the enclosing *state* or that the overall SMD is completed. It is depicted as a filled circle nested inside another (see Figure 2.4).

A *transition* is a mechanism by means of which an object leaves a *state* and changes to a new *state*. Each *transition* can have a *guard* condition that indicates if the *transition* can even be considered (enabled), a *trigger* specified by an *event* that causes the *transition* to execute if it is enabled, and the *effect* that will be executed when the *transition* occurs [19]. They are represented by labelling the *transition* with `trigger[guard]/effect` (e.g., `skipClass[missedSessions>5]/suspend`). For example, in Figure 2.4, *guard* `missedSessions>5` is checked as soon as *event* `skipClass` occurs. If the *guard* is false, there is no change of *state* and the *effect* `suspend` is not executed. Alternatively, if the *guard* is true, there is a change of *state*, and the *effect* `suspend` is executed during the change to the *studying* target *state*. A *transition* has a source and a target *vertex*, where each *vertex* can be either a *pseudostate* or a *state*. A *pseudostate* is an abstraction used to connect multiple *transitions* into more complex *state transitions* paths. An example of a *pseudostate* is the *initial pseudostate*, which is represented by a filled circle, which can refer to the creation of an object or to the source of a transition to the default state of a composite state (see Figure 2.4).

UML extension mechanisms

OMG provides two types of extension mechanisms: *heavyweight extension mechanisms* and the *lightweight extension mechanisms* [20]. While the *heavyweight extension mechanisms* provide mechanisms to change and adapt UML by modifying the standard UML metamodel,

the *lightweight extension mechanisms* are built into UML for enabling the adaptation of UML to the requirements of a specific domain, without modifying the metamodel. Particularly, this latter extension mechanism is supported by *profiles*. The *profiles* clause describes capabilities that allow UML metaclasses to be extended to adapt them for different purposes [1], for instance, to tailor the UML metamodel for different platforms (such as J2EE or .NET) or domains (e.g., Service Oriented Architecture).

A UML *stereotype* is defined as part of a *profile* for specifying how an existing UML metaclass (such a `class`, an operation, etc.) may be extended. As a way of example, in Figure 2.5 we show an example extracted from the UML specification (page 265 in [1]) which illustrates both the definition and the application of a *stereotype* named `Creator`. In particular, the left-hand side of Figure 2.5 shows the definition of this *stereotype*, which is specified to be applicable to instances of the metaclass `Class` to provide them with extra semantics regarding their creator. A *stereotype* is denoted like a UML `Class`, linked to the metaclass to which it extends, and with the keyword `«stereotype»` above the name (see Figure 2.5). Like a UML *class*, a *stereotype* may have *properties*, which have traditionally been referred to as *tag definitions*. A *stereotype's* *property* specifies a new kind of property that may be attached to the model element to which the *stereotype* extends (e.g., `name` and `date` in the stereotype `Creator`). When a *stereotype* is applied to a model element (an instance of a *stereotype* is linked to an instance of a metaclass), the name of the *stereotype* is normally shown within a pair of guillemets (`« »`). Similarly, when a *stereotype* is applied, the values of its *properties*, which are traditionally referred to as *tagged values*, can be shown in a comment symbol attached to the model element with the form `name = value`, where `name` is the name of the *tag definition* and `value` denotes its value. For example, in the right-hand of Figure 2.5 we show the application of the stereotype `Creator` to the particular class `StopWatch`, with concrete *tagged values*.

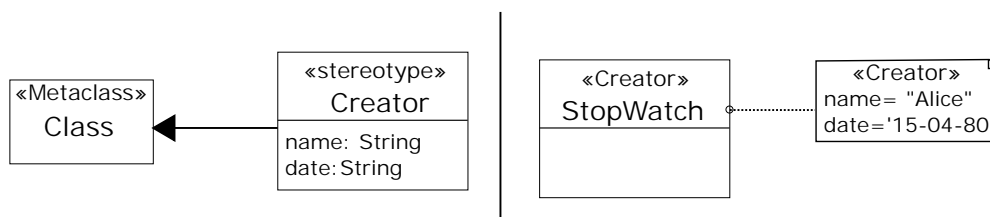


Fig. 2.5 The left-hand side shows the definition of the *stereotype* named `Creator`. The right-hand side depicts the application of such a stereotype.

2.1.2 Provenance-related background

In this section we will present two crucial approaches to understand the conceptual definition of UML2PROV. First, we provide an overview of the PROV standard [2] for representing provenance information; later, we explain the PROV Template approach [3] for designing provenance.

The PROV standard

In the past, most provenance systems had their own model for defining and managing provenance, which caused a hindrance to exchange and use the generated provenance by other users [10]. Against this background, the PROV standard [2] emerged. PROV is a World Wide Web Consortium (W3C) standard that aims to facilitate the publication and interchange of provenance among applications. PROV is fully specified in a family of documents allowing provenance to be modeled, serialised, exchanged, accessed, merged, translated, and reasoned over [14]. Among this family of documents, there are a conceptual data model (PROV-DM) [21], an OWL ontology (PROV-O) [22], an XML serialisation format (PROV-XML) [23], a human-readable text notation (PROV-N) [5], a formal semantics of the conceptual model (PROV-SEM) [24], a set of constraints and inference rules (PROV-CONSTRAINTS) [25], and a mapping to Dublin Core (PROV-DC) [26]. Inspired by [2], Figure 2.6 shows the organisation of this family of documents and (roughly) how they depend on each other. More specifically, the core is the conceptual data model (PROV-DM), which defines the vocabulary for describing provenance. To allow the implementation of provenance validators, a set of constraints (PROV-CONSTRAINTS) has been defined over PROV-DM. The provenance defined with the vocabulary stated in PROV-DM can be serialised in: RDF (PROV-O), a readable text notation (PROV-N), and XML (PROV-XML).

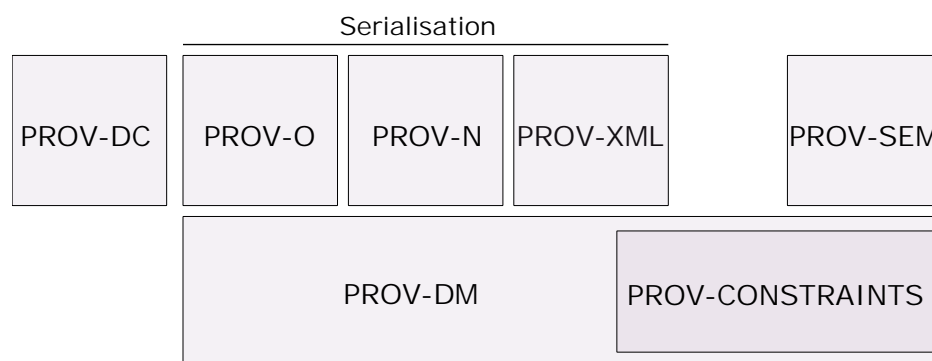


Fig. 2.6 Organization of the PROV family of documents [2]

Additionally, PROV-DM is complimented by a formal semantics (PROV-SEM). Finally, PROV-DC is an additional specification for representing dictionary style collections.

One of the purposes of our approach will be to obtain, from applications' execution, provenance information that is compatible with the PROV standard, in order to allow its exchange and use by other users. Thus, next, we will focus on explaining the elements that make up PROV-DM [21]. Concretely, PROV-DM is based around three classes or concepts (*Agent*, *Entity*, and *Activity*) and their relations, which are shown in Figure 2.7. Below, in our explanations, we will use the PROV-N [5] human-readable text notation for these concepts, aimed at illustrating them.

- *Entity is a physical, digital, conceptual or other kind of thing with some fixed aspects. An entity is written in PROV-N as `entity(id, [attributes])`, where `id` is an identifier for the *entity*, and `attributes` is an optional set of attribute-value pairs (e.g., `(attr1, val1),...`) representing additional information about the fixed aspects of the *entity*.*
- *Activity is something that occurs over a period of time and acts upon or with entities; it may include consuming, processing, transforming, modifying, relocating, using, or generating entities. An activity is written in PROV-N as `activity(id, startTime, endTime, [attributes])`. In such an expression, the `id` is an identifier for the *activity*; `starTime` is an optional time for the start of the *activity*; `endTime` is an optional time for the end of the *activity*; and `attributes` is an optional set of attribute-value pairs representing additional information about the *activity*.*
- *Agent is something that bears some form of responsibility for an activity taking place, for the existence of an entity, or for another agent's activity. An agent is written in PROV-N*

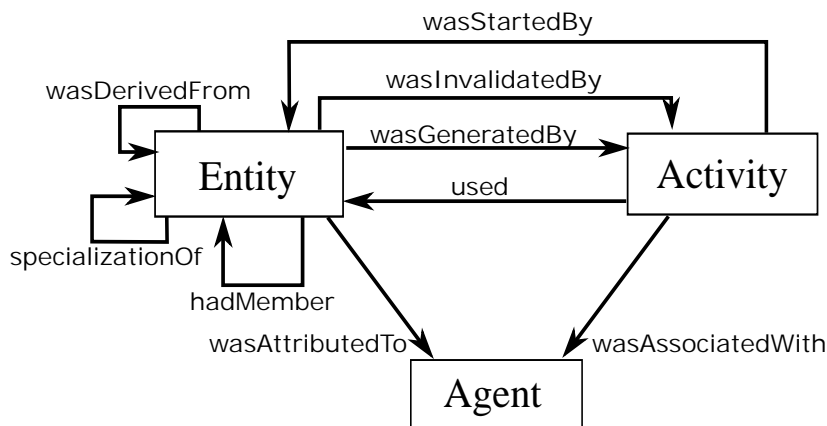


Fig. 2.7 The core concepts of PROV-DM together with their relations

as `agent(id, [attributes])`, whereby `id` is the identifier of the *agent*, and `attributes` is a set of attribute-value pairs representing additional information about the *agent*.

These elements can be associated with each other by means of relations, among which we highlight (see Figure 2.7): *used*, which is the beginning of utilizing an *entity* by an *activity*; *wasGeneratedBy*, which is the completion of production of a new *entity* by an *activity*; *wasDerivedFrom*, which is a transformation of an *entity* into another; *wasAssociatedWith*, which is an assignment of responsibility to an *agent* for an *activity*; *wasAttributedTo*, which is the ascribing of an *entity* to an *agent*; *wasStartedBy*, which is when an *activity* is deemed to have been started by an *entity*; *wasInvalidatedBy*, which is the start of the destruction, cessation or expiring of an existing *entity* by an *activity*; *specializationOf*, utilised for showing an *entity* which shares the aspects of another *entity*, but also has more aspects; and finally *hadMember*, used for stating the members of an *entity*.

Notational Conventions Throughout this memoir, in the depicted PROV documents, we use *qualified names* (e.g., `prov:value`) conforming to PROV-DM [21]. In compliance with PROV-DM, a *qualified name* can be mapped to an Internationalized Resource Identifier (IRI) [27] by concatenating the IRI associated with the prefix (e.g., `prov`) and the local part (e.g., `value`). Every *qualified name* with a prefix refers to the namespace of the prefix. The prefixes and namespaces used in this thesis are depicted in Table 2.1. We note that the three first prefixes refer to well-known namespaces as provided in the corresponding definitions (see column “Definition”), whereas the two last namespaces refer to concrete prefixes provided in this research work. More specifically, the namespaces associated with `var` and `tmpl` belongs to the PROV-Template approach [3]. Whereas `var` is used to represent variables in such a proposal (it will be described in the next section), `tmpl` is used to control the expansion of variables meant to generate time and string values (we refer readers to [28] for details about its usage). Conversely, `u2p` has been defined to identify concrete information as stated by our proposal. Later on,

Table 2.1 Prefixes and Namespaces used in this document

Prefix	Namespace IRI	Definition
prov	http://www.w3.org/ns/prov#	The PROV namespace
xsd	http://www.w3.org/2000/10/XMLSchema#	XML Schema namespace
ex	http://example.com	It represents application-dependent IRIs
var	http://openprovenance.org/var#	The namespace for PROV template variables
tmpl	http://openprovenance.org/tmpl#	The prov-template namespace
u2p	http://uml2prov.unirioja.es/ns/u2p#	UML2PROV namespace

we will present how this prefix is used, but the reader interested in further information about `u2p` is referred to its namespace IRI (see column “Namespace IRI”).

On the other hand, in some cases, when the information in a PROV document is not very extensive, a graphical notation could help us get an idea of such information at a glance. The most widely used graph notation for PROV is presented in [29]. Although it cannot be considered as a standardized notation for expressing provenance, we have decided to use it throughout this thesis in order to show the essence of the depicted PROV documents. Concretely, this notation will allow us to see at a glance PROV *entities*, *activities*, and *agents*, as well as their relations. As we can see in Figure 2.8, *entities*, *activities* and *agents* are represented as nodes, with oval, rectangular, and pentagonal shapes, respectively. The *relations* are represented as directed edges, while *attribute-value pairs* are depicted inside a note.

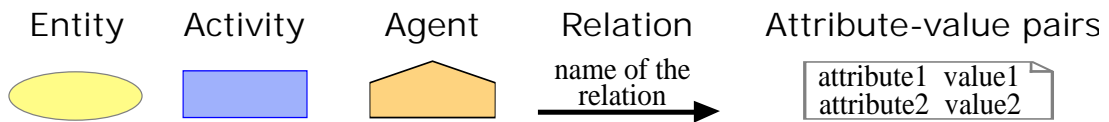


Fig. 2.8 A graph notation for PROV elements

For instance, the execution of an operation that changes to upper case the name of a student (e.g., from “avelina” to “AVELINA”) in the University example, could be explained by the PROV documents depicted in Figure 2.9. The left-hand side of this figure depicts the PROV document in PROV-N format, whereas the right-hand side depicts the same information using the above graphical notation. Both representations show that an *activity* with the identifier `ex:toUpperCase` used the *entity* `ex:inParam1` with the attribute-value pair `prov:value="avelina"`, and generated (*wasGeneratedBy*) the *entity* `ex:outParam1` with the attribute-value pair `prov:value="AVELINA"`; thus, it could be said that the latter *entity* derived (*wasDerivedFrom*) from the former *entity*.

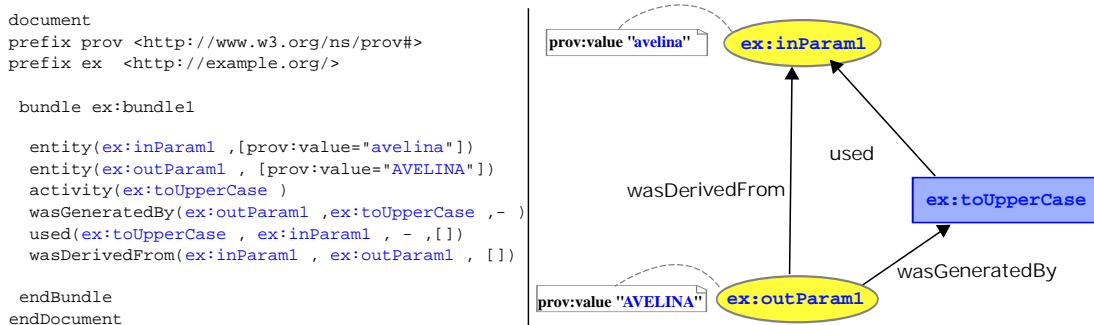


Fig. 2.9 A PROV document in PROV-N together with its graphical representation

PROV-Template approach

In addition to obtaining provenance information compatible with the PROV standard, our proposal aims at obtaining the design of the provenance directly from the design of the application. In particular, to express the provenance design, we will use the PROV-Template approach [3]. PROV-Template is a declarative approach that is intended to enable designers and developers to design and generate provenance compatible with the PROV standard. The architecture of this templating approach is made up of four main elements: *PROV template*, *binding*, *expansion algorithm*, and the final *PROV document*. In Figure 2.10, we present an overview of the PROV-Template architecture where all these elements are included.

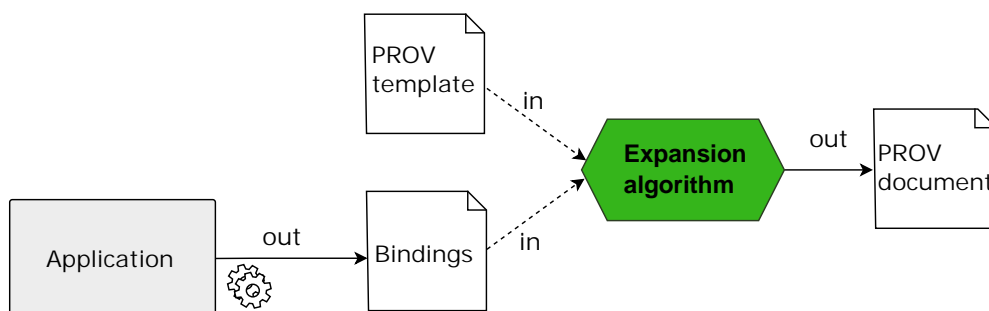


Fig. 2.10 The architecture of the templating approach proposed in [3]

First, the design of the provenance to be generated is defined by means of a *PROV template* (see Figure 2.10). A *Template* is a provenance document expressed in a PROV-compatible format, containing placeholders (referred to as *variables*) for values. As a way of example, Figure 2.11 shows a *PROV template* that would correspond to the design of the provenance depicted in Figure 2.9. In this template there is an *activity* with the identifier `var:operation`

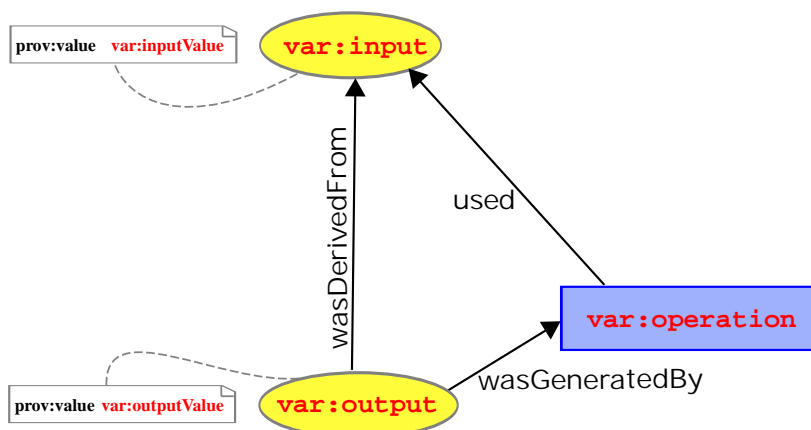


Fig. 2.11 A graphical illustration of a *template*, where variables are qualified names with prefix `var` and appear in red font.

(the prefix `var` denotes that it is a *variable*). This *activity* is related to the *entity* `var:input` by means of the relation *used*, and it is also associated with the *entity* `var:output` through the relation *wasGeneratedBy* (the relation *wasDerivedFrom* associates the latter *entity* with the former). Additionally, we can see two more variables in the list of *attribute-value* pairs of both *entities*: `var:inputValue`, in *entity* `var:input`, and `var:outputValue`, in *entity* `var:output` (later we will see the information that these attributes provide).

The second step consists of obtaining the *bindings* (see Figure 2.10) from the application execution. A *binding* is an association between a template's *variable* and value(s), constructed at runtime by a provenance-aware application. They are usually grouped in *sets of bindings* (or *bindings* for short). The PROV-Template approach is agnostic about the mechanism used to create *bindings*. For instance, an option could be to log the values of the bindings' variables and later to create the *bindings* from them, or, alternatively, to directly obtain the *bindings* as the application executes. An example of *bindings* for the *template* of Figure 2.11 is shown in Figure 2.12. In particular, these *bindings*, which are serialised in JSON-LD [30], include: the `"var"` key, that contains the bindings represented as a JSON dictionary; `@id`, which is the identifier of the corresponding value; `@type`, that is the data type of the corresponding value; `@value`, which is the string serialisation of the value; and finally, the `"context"` key, which provides the information to interpret the prefixes used in values. All the variables in the template of Figure 2.11 are associated with a value: `var:operation` with `ex:toUpperCase`, `var:input` with `ex:inParam1`, `var:inputValue` with `"avelina"`, `var:output` with `ex:outParam1`, and `var:outputValue` with `"AVELINA"`.

```
{
  "var":{
    "operation": [ { "@id": "ex:toUpperCase" } ],
    "input": [ { "@id": "ex:inParam1" } ],
    "inputValue": [ { "@type": "xsd:string", "@value": "avelina" } ],
    "output": [ { "@id": "ex:outParam1" } ],
    "outputValue": [ { "@type": "xsd:string", "@value": "AVELINA" } ]
  }
  "context":{
    xsd: "https://www.w3.org/2000/10/XMLSchema#",
  } ...
}
```

Fig. 2.12 An example of *bindings* for the *template* of Figure 2.11. They are encoded in JSON-LD with variable names in bold red font, and values in underlined blue font.

Finally, the *expansion algorithm* (see Figure 2.10) takes as input the *template* and the *bindings*, and generates the *PROV document* with the provenance information to be exploited. This resulting *PROV document* contains the shape expressed in the *template* and the values, associated with each variable, as given by the *bindings*. For example, the *PROV document* obtained from the *template* in Figure 2.11 and the *bindings* in Figure 2.12 would correspond to the one shown in Figure 2.9. We would like to note that, in a *binding*, a variable identifying

a PROV template's element (*entity*, *activity* or *agent*) can be associated with several values. In this case, the *expansion algorithm* will generate, in the resulting PROV document, one PROV element for each associated value. Consequently, the associations in which the PROV element is involved will be replicated in the resulting PROV document.

We note that, since the previous process can be performed several times, it would result in many PROV documents. Such documents can be merged in a single PROV document, gathering together all the provenance information in a single place. To do this, we can use the tool ProvToolbox [16] which has the functionality for merging PROV documents, ensuring that the PROV elements are uniquely represented with all their attributes.

As Moreau et al. stated in [3], the PROV-Template approach provides several benefits in terms of spatial overhead: *bindings* have a size that is typically 40 percent of that of expanded provenance templates. Additionally, Moreau et al. also claim that the PROV-Template approach offers several benefits from the software engineering perspective such as separation of responsibilities, provenance maintenance, and so on.

2.2 Implementation Background

This section is devoted to those strategies that we have used to provide an implementation of UML2PROV. We start by describing some notions of the Model Driven Development approach, and then we give an insight into the Aspect Oriented Programming paradigm.

2.2.1 Model-Driven Approaches

This section introduces the notions of Model Driven Development (MDD) and Model Driven Architecture (MDA) as they are described in literature, and gives a global overview of the MDA approach.

As it is presented in [31], *model-driven development is simply the notion that we can construct a model of a system that we can then transform into the real thing*. The MDD approach is focused on models, rather than on computer programs, thereby the source code is automatically generated from them through a refinement process. In this process, a high-level abstract application model is successively translated into increasingly more detailed models until one of these models can be directly translated into a final running system executed by some platform. Concretely, UML2PROV advocates transforming UML diagrams, which are considered a partial graphical representation of a model of a system, into artefacts for generating provenance. Thus, the use of MDD to perform this task is quite natural.

Possibly, the most well-known initiative of MDD is the Model Driven Architecture (MDA) proposal [4, 32]. This framework leverages OMG's standards to support the creation of models. One of the main elements introduced by MDA is *meta-modeling* [32]. The main idea behind *meta-modeling* is that models always conform to domain application *metamodels*. A model defines what elements can exist in a system, whereas a *metamodel* defines every kind of element that makes up their models. Hence, it is allowed to use a specific element in a model because its metamodel considers a meta-element that defines the kind of that element. For instance, since UML diagrams are considered a partial graphical representation of a model of a system, a concrete model must conform to the UML metamodel. Additionally, one of the main goals of MDA is to separate the specification of a system from its implementation aspects on a specific platform. Particularly, MDA mainly identifies two types of models of a system, conforming to the corresponding metamodels (see Figure 2.13): the Platform Independent Models (PIMs), which describe the system without any implementation detail being independent of the technological platform used to implement the solution, and the Platform Specific Models (PSMs), which refine the PIM model by including implementation details. Particularly, PSMs should be automatically obtained from PIMs and then, the PSMs will be finally implemented in the target platform. As it is depicted in Figure 2.13, usually, the model driven process comprises defining the PIM model, transforming it into the PSM, and finally generating the code to be executed on the chosen platform, obtaining the final software product. This figure also depicts how each model, independently of its type, conforms to a metamodel.

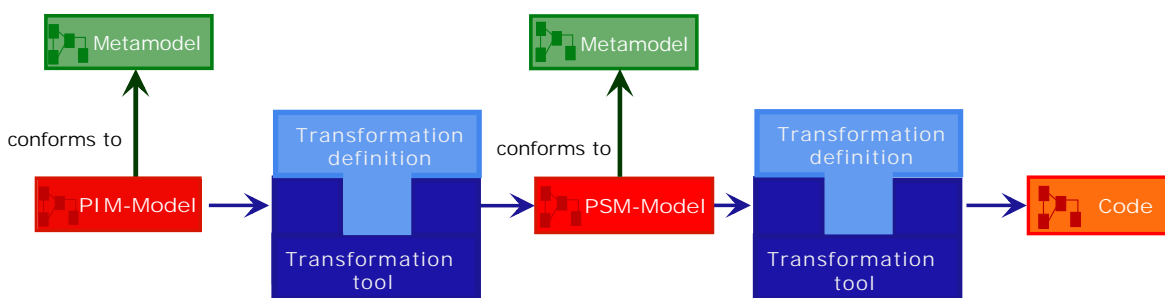


Fig. 2.13 Simplified model driven process inspired by the one presented in [4]

Depending on the type of artefacts (models and code) that takes place in the MDA approach, it can be distinguished between *model to model transformations* (M2M), in which both the source and the target are models, and *model to text transformations* (M2T), which define transformations from a model to the final text. Among the different MDA-based tools, in this thesis, we have used the ATLAS Transformation Language (ATL) [33] to carry out the M2M transformations, whereas the M2T transformations have been performed by means

of the Extensible Stylesheet Language Transformations (XSLT) [34] and XPand [35]. Next, we provide a brief explanation of these tools:

XSLT is a specification for transforming XML documents into another text format (e.g., HTML, XML, or anything else). For example, since UML models can be serialised as XML using the XML Metadata Interchange (XMI) [1], XSLT could be used to implement UML model transformations. A transformation in the XSLT language is expressed in the form of a *stylesheet*, which is made up of one or more XML documents. The transformation is achieved by a set of *template rules*. A *template rule* associates a source pattern, which typically matches nodes in the source document, with fixed text containing instructions to obtain information from the source model. The resulting text document will contain the fixed text together with the text resulted from the execution of such instructions. XSLT uses the XML Path Language (XPath) [36] to define such instructions. XPath is an expression language that allows navigating through elements and attributes in an XML document.

ATL is a model transformation language that enables to specify how one (or more) target model(s) can be produced from a set of source models. ATL introduces a set of concepts that make it possible to describe model transformations by providing both declarative and imperative constructs. More specifically, in ATL, the main concept is the *rule*, which can be a *matched rule* or a *called rule*. *Matched rules* constitute the core of an ATL declarative transformation since they make it possible to specify (1) for which kinds of source elements the target elements must be generated, and (2) the way the generated target elements have to be initialised. Conversely, *called rules* provide developers with convenient imperative programming facilities. They have to be explicitly called to be executed and they can accept parameters. Developed on top of the Eclipse platform, the ATL Integrated Environment [37] provides several development tools aiming at facilitating the development of ATL transformations.

XPand is a statically-typed template language that enables the generation of text (e.g., code) from models. Concretely, XPand transformations rely upon *templates*. An XPand *template* associates a given element type in the source model with a block of text (which could be code) containing placeholders. During the transformation, the placeholders are filled with the concrete values obtained from the addressed element in the source model. Similar to ATL, the XPand team provides users with a set of plugins [38] for the Eclipse platform in order to ease development of XPand templates.

2.2.2 Aspect Oriented Programming

It is very common that developers come across issues that are not possible to address in an accuracy way by means of the usual paradigms such as procedural, functional, or Object-Oriented Programming (OOP). Concretely, developers still have difficulties placing concerns that do not fit naturally into a single module [39] (e.g., transaction management, security, caching, and so on). Thus, they have to make decisions that usually incur in code located across all the application. As a result, developers may write *tangled code*, in which a class or module implements additional behaviour not related to its original behaviour, or *scattered code*, in which the implementation of a behaviour is located across different parts of the system [39]. These situations (*tangled* and *scattered code*) are deemed to affect both the design and the development of applications in several ways, making the code unclear, and difficult to maintain.

Against this background, the aspect-oriented programming (AOP) [40] paradigm aims to increase modularity of applications by allowing the separation of *crosscutting concerns* from *core concerns*. A *core concern* refers to the main behaviour of a module (e.g., book and ATM transactions, management of clients and accounts, and so on), whereas a *crosscutting concern* refers to secondary behaviours that *crosscut* several modules (e.g., concurrency control and tracing). For instance, a well-known potential *crosscutting concern* in the University example could be to trace operation calls. A developer who wants to implement this concern should add code in every method of the application to be traced, crosscutting the *core concerns* of the application and incurring in tracing code *scattered* throughout all the application classes.

The modular unit in AOP is the *aspect*, which encapsulates a concrete *crosscutting concern* and identifies those points in the source code in which the *crosscutting concern* is executed. Later, the *aspect weaver* composes the final system by combining the *crosscutting concern*, defined in an *aspect*, and the *core concerns* through a process called *weaving*. There are two main types of *aspect weavers* depending on the moment in which the weaving is performed. While *compile-time weavers* perform the weaving as an additional phase in the compilation phase, *run-time weavers* carry out the weaving “on the fly” as the application is running. To implement a *crosscutting concern*, an AOP system may include many of the following concepts defined as follows in [39]:

- *Join points*. They are well-defined points in the execution of a program. These may include execution of methods, creation of objects, or throwing of exceptions. Such identifiable points in the system are called *join points*.
- *Pointcut*. Implementing a crosscutting concern requires selecting a specific set of *join points*. A *pointcut* selects any *join point* that satisfies some condition. For instance, a

pointcut could refer to all the operations of a concrete class or package, or alternatively, to those operations that start with a concrete string of characters.

- *Advice*. After a *pointcut* selects *join points*, the focus is augmenting those *join points* with additional or alternative behaviour. The *advice* construct in AOP provides a facility to do so. An *advice* adds behaviour *before*, *after*, or *around* the selected *join points*. A *before advice* executes before the *join point*, whereas an *after advice* executes after it. An *around advice* surrounds the *join point* execution and may execute it zero or more times.
- *Aspect*. Since the end goal of AOP is to have a module that embeds or encapsulates crosscutting logic, a place to express that logic is needed. An *aspect* provides such a place. Concretely, an *aspect* contains *pointcuts* and *advices*. It is worth noting that *aspects* become a part of the system and therefore, they can use the elements of the system (e.g., classes in it) to perform their work.

There is a huge number of systems that have been defined on top of widely used programming languages in order to provide them with AOP functionality. One of them is *AspectJ* [41], which is an AOP extension to the Java programming language, *Ajaxpect* [42], which is an AOP library for Ajax with JavaScript, and *Aspects* [43], which is a Python module that enables AOP. As we will see in Chapter 5, which introduces our reference implementation in Java, we have used AspectJ for being the most relevant AOP proposal for Java [39]. Thus, next we give a brief introduction to the main features of AspectJ used later in such a chapter.

AspectJ

AspectJ [41] is an AOP extension to the Java programming language that, particularly, allows a clean modularization of *crosscutting concerns*. AspectJ supports two styles for implementing the aforementioned AOP concepts. One is a superset of Java syntax with additional keywords (i.e., a *code-based style*), whereas the other one is a set of Java annotations (i.e., an *annotation-based style*). Below, we will explain the AspectJ code-based style for implementing the aforementioned AOP concepts, which is the one used in our approach. In order to illustrate such an explanation, we will suppose that we are interested in measuring the execution time of all the operations with the name `setName`, from the University example. But, instead of manually adding the corresponding Java instructions throughout the code, we decide to define an AspectJ *aspect*. The AspectJ *aspect* implementing this *crosscutting concern* (hereinafter we will refer to it as `UniversityExampleAspect`) can be seen in Figure 2.14.

Join point As we have mentioned previously, they are well-defined points in the execution of the program. AspectJ permits a subset of all possible *join points*. Examples of these

```

aspect UniversityExampleAspect {
    void around(): call(* *.setName(..) {
        long startTime = System.currentTimeMillis();
        proceed();
        long stopTime = System.currentTimeMillis();
        long elapsedTime = stopTime - startTime;
        System.out.println(elapsedTime);
    }
}

```

Diagram annotations:

- aspect**: A vertical line on the left side of the code block, spanning from the opening curly brace to the closing curly brace.
- advice**: A vertical line on the left side of the code block, spanning from the `around()` method signature to the closing curly brace.
- pointcut designator**: A red arrow points to the `call(* *.setName(..)` expression.
- type of advice**: A red arrow points to the `around()` method signature.
- actual behaviour**: A red arrow points to the `proceed();` line.
- behaviour implementing the crosscutting concern**: A red arrow points to the `long startTime = System.currentTimeMillis();` line.
- crosscutting concern**: A red arrow points to the `long stopTime = System.currentTimeMillis();` line.

Fig. 2.14 An AspectJ *aspect* implementing a crosscutting concern to the University example.

supported *join points* are method/constructor calls, method/constructor executions, read/write attributes, exception handler executions, class initializations, object initializations, and object pre-initializations. For instance, in the `UniversityExampleAspect`, we are interested in capturing calls to operations with the name `setName`.

Pointcut Aiming at creating the sets of *join points*, AspectJ includes several primitive *pointcut designators*. A simple way to think about a *pointcut designator* is in terms of matching certain *join points* at runtime. In order to capture *join points* that share common characteristics, AspectJ defines *wildcards*. The *pointcut* defined in the `UniversityExampleAspect` has the *pointcut designator* `call` and the *signature* “`* *.setName(..)`” (see Figure 2.14). The `call` *pointcut designator* denotes that the *pointcut* refers to all the operation calls that match a *signature*. As far as the *signature* is concerned, it is defined using two types of wildcards (“`*`” and “`..`”). The wildcard “`*`” denotes any number of characters except for the “`.`” character. Thus, the first “`*`” refers to any type of return parameter, whereas the second “`*`” means any class, in any package. The “`..`” wildcard denotes any number of characters including the “`.`” character. When it is used in the parameters section of an operation, it refers to an arbitrary number of parameters, none or any parameters (see Figure 2.14).

Advice AspectJ distinguishes among three different types of *advices* depending on the moment in which the *advice* is executed: *before*, *after*, and *around*. When there is a *before/after advice*, the behaviour in the *advice* is executed before/after the actual behaviour proceeds when its *pointcut* is matched. An *around advice* can modify the execution of the code that is at the *join point*, by replacing it, or even bypassing it [39]. Additionally, the behaviour inside the *around advice* could proceed with the actual behaviour when it

considers it necessary. To do this, AspectJ defines the `proceed()` statement for carrying out the actual behaviour. In the `UniversityExampleAspect`, we define an *around advice* associated with the *pointcut* “`call(* *.setName(..))`”, so that the behaviour included in the *advice* will be executed each time the *pointcut* is matched (i.e., when there is a call to an operation with the name *setName*). We have used an *advice* of type *around* in order to wrap the advised operation and to execute additional instructions both before and after it. As we can see in Figure 2.14, the `proceed()` statement, which carries out the advised operation, is wrapped by instructions implementing the crosscutting concern.

Aspect In AspectJ, an *aspect* can contain data, methods, and nested class members, just like a normal Java class; and additionally, it also contains the AOP constructs that define the *crosscutting concern* such as *pointcuts* and *advices*. In the `UniversityExampleAspect`, we can see that an AspectJ aspect is similar than a Java class, but with AOP constructs inside such as an *advice*. In this case, it only defines an *around advice*.

Finally, as we have mentioned previously, an *aspect weaver* composes the final system by combining the classes and *aspects* through a process called *weaving*. AspectJ offers three types of compile-time weavers (*source*, *binary* and *load-time weaving*) [39].

Chapter 3

State of the art: A systematic review of provenance systems

The overall objective of this chapter is to plot the landscape of published studies in the field of provenance, with two main purposes for this thesis. First, to present a consistent background of remarkable aspects on the provenance issue. Second, to justify those open research problems in the provenance field that serve as the basis for this thesis.

The information provided in this chapter is a concise version of a systematic review that was conducted to focus the work developed in this thesis. Among other results of this review, the proposal of a taxonomy of provenance systems stands out. Such a taxonomy is in itself a relevant result of the research carried out in this thesis.

Interested readers can find this revision work in the following reference:

- Beatriz Pérez, Julio Rubio, Carlos Sáenz-Adán: *A systematic review of provenance systems*. In *Knowledge and Information Systems* 57(3): 495-543 (2018) [10]

In addition, to support part of the conclusions presented in the abovementioned paper [10], we used the work presented in:

- Carlos Sáenz-Adán, Francisco J. García-Izquierdo, Ángel Luis Rubio, Eduardo Sáenz-de-Cabezón Irigaray, Emilio Rodríguez-Priego, Oscar Díaz: *A Tool for Management of Knowledge Dispersed throughout Multiple References*. In the *International Conference on Software Paradigm Trends (ICSOFT-PT) 2015*. [44]

3.1 Introduction

In recent years there has been a rapid growth of the provenance field, in general, and of provenance systems, in particular, which has derived into a large and heterogeneous research

corpus of approaches to address a variety of provenance concerns. Even so, at the present time there does not appear to be a clear consensus or common ground on aspects such as what requirements a provenance system should support, or what technical details are involved in making these systems possible [45, 46]. Among such corpus of research, a non-negligible number of surveys have touched the provenance field [45–63]. Although all these surveys provide to some extent an overview of research issues in provenance, they address them from different perspectives, even using different terminology to refer to the same concept. Only a few of these works explicitly present some kind of categorisation scheme or taxonomy for provenance characteristics ([49–51, 59]), making it difficult to provide a comparison and to identify equivalent provenance aspects among these surveys. Although these works identify a significant number of dimensions of comparison, they miss tackling aspects such as integration of provenance capture with applications, interoperability of provenance data, computational overhead or querying of provenance data. Some of these surveys apply their classifications to specific provenance systems, being [61] the most complete proposal comparing just 13 systems among the wide number of existing solutions.

Given the size and heterogeneity of the provenance literature, and taking into account that, to the best of our knowledge, existing surveys not only are restricted to specific provenance issues but they have also been conducted neither with an appropriate level of formality nor in a systematic way, we have seen the need to organise and synthesise the existent research corpus as a first step of this thesis. This task will offer us several benefits at the early stage of this thesis. Firstly, it will provide a complete and consistent background of provenance concepts and techniques. Secondly, not only will it inform about up-to-date available approaches and technologies, but it will also help us uncover open research problems that serve as motivation for this thesis. This work was performed by means of a *systematic review* [64–66]. A systematic review is a mean of identifying, evaluating and interpreting all available research relevant to a particular phenomenon of interest in a thorough and unbiased manner [64]. In contrast to a usual survey, it follows a controlled, rigorous and auditable procedure for searching, collecting and selecting the acceptable quality published studies to be considered in the review. Another difference from usual surveys is that the process followed in systematic reviews is formally documented and hence it is repeatable.

Our systematic review on provenance systems is published in [10], providing the following contributions to the state of the art in the provenance field, in general, and to this thesis, in particular.

- Based on a comprehensive, thorough overview covering both early and recent work in the area, the systematic review proposed a *unified taxonomy of provenance systems characteristics*. This taxonomy encompasses the overall classifications considered by

other works, and more fully captures the unique characteristics of provenance systems. In contrast to other proposals, it is not limited either to a specific area of data management or to a domain of application. This taxonomy, together with the process to obtain it, provided us with clear benefits at the beginning of this thesis, because it gave us a broad overview of the desired characteristics that provenance systems are expected to have, and familiarised us with the terminology used in the field of provenance.

- We identified existing provenance systems considering not only those which were in wide use for a long time but also those which emerged recently. We reviewed a representative set of systems that we considered as the most representative or influential. More specifically, based on our taxonomy, we performed an exhaustive analysis and comparison of 25 systems, which illustrated different solutions covering a broad spectrum of alternatives along the aspects identified in our taxonomy. This analysis gave us an idea about the current available approaches and technologies, and additionally, it also allowed us to discover the open research problems that motivate this thesis.

Additionally, this systematic review is intended to provide several benefits to potential readers. First, the resulting taxonomy and review of related background aim at informing and enhancing the understanding of the field to potential researchers, software developers or provenance technology users. Second, this work can enable general users to distinguish different perspectives of provenance and guide them in their decision towards the selection of the most suitable solution for their needs. Third, our target readership could also be those potential researchers and software developers who are interested in up-to-date available approaches, as well as open problems being seen in practice. Finally, the results provided by this review could be particularly relevant for researchers aimed at identifying provenance research issues that have been already tackled or directions for future research.

The thoroughness and size (more than 45 pages) of this systematic review prevent us from giving a complete and detailed description of it in this memoir. Instead, we refer readers to [10] for a detailed explanation, and herein, we will restrict attention to those aspects of interest for this thesis. Concretely, we will give (1) an overview of the process for obtaining the taxonomy of provenance systems characteristics, (2) a brief explanation of the dimensions making up such a taxonomy, giving examples of systems and paying special attention to those aspects needed to understand our proposal, and, finally, (3) based on the previous explanation, the uncovered open problems that serve as foundations for this thesis.

3.2 Towards the definition of a taxonomy of provenance systems characteristics

As a starting point for the definition of our taxonomy, we performed a search strategy, paying special attention to published surveys and reviewing papers (hereinafter we will refer to these types of studies as simply surveys) on the provenance topic. Given the cost and time involved in conducting these type of reviews, we performed our systematic review in two review processes. The *first review process* was carried out from September 2015 to June 2016, and included research studies published up to and including December 2015. The *second review process* took place between July and August 2017, and updated the *first review process*, covering research studies published from January 2016 to July 2017. Therefore, the overall systematic review covers studies published up to and including July 2017.

As a result of the overall search, we selected a set of 19 surveys, identified by “Su<N>”, being *N* a numeric identifier: Su1 [45], Su2 [56], Su3 [58], Su4 [53], Su5 [54], Su6 [55],

Table 3.1 List of the reviewed systems together with their references

[ID]	System	Authors' Rep. Name	References
[S1]	VisTrails	J. Freire et al.	[67–71]
[S2]	myGrid/Taverna	K. Wolstencroft et al.	[72–75]
[S3]	Kepler provenance	I. Altintas et al.	[76, 77]
[S4]	PASS	D. A. Holland et al.	[78–80]
[S5]	Trio	J. Widom et al.	[81–86]
[S6]	Karma	L.Y. Simmhan et al.	[87–90]
[S7]	Chimera	I. Foster et al.	[91–93]
[S8]	ZOOM	O. Biton et al.	[94–97]
[S9]	Cui 2000	Y. Cui et al.	[98–101]
[S10]	Swift	I. Foster et al.	[102–105]
[S11]	PASOA/PreServ	P. Groth et al.	[106]
[S12]	Tioga	M. Stonebraker et al.	[107–109]
[S13]	Wings-Pegasus	E. Deelman et al.	[110–116]
[S14]	Buneman	P. Buneman et al.	[117]
[S15]	SPADE	A. Gehani et al.	[118–122]
[S16]	ORCHESTA	Z.G. Ives et al.	[123–127]
[S17]	Perm-GProM	B. Glavic et al.	[128–137]
[S18]	ES3	J. Frew et al.	[138, 139]
[S19]	COMAD	S. Bowers et al.	[140]
[S20]	DBNotes	L. Chiticariu et al.	[141, 142]
[S21]	Lipstick	Y. Amsterdamer et al.	[143]
[S22]	Redux	R. Barga et al.	[144]
[S23]	BURRITO	P.J. Guo et al.	[145, 146]
[S24]	PLUS	B. Blaustein et al.	[147–150]
[S25]	RAMP	R. Ikeda	[151, 152]

Su7 [46], Su8 [48], Su9 [47], Su10 [51], Su11 [49], Su12 [59], Su13 [52], Su14 [57], Su15 [50], Su16 [60], Su17 [61], Su18 [63], and Su19 [62]. From these surveys, and other papers (more than 500) obtained through a snowballing process from these surveys, we identified a total of 251 provenance systems. Among these systems, we selected 25 for being, by far, the most referenced. Table 3.1 shows the list of the 25 selected systems together with a representative authors' name, and a list of references that (1) specially deals with the system, or (2) performs any type of quality analysis of the system (such as surveys or reviewing papers). Each selected provenance system is referred to as a unique identifier of the form “[S<N>]”, where N is the numeric identifier, followed by the name of the system, including a reference to its citation when no name is given (e.g., [S1] VisTrails and [S9] Cui 2000, respectively). Additionally, we refer readers to Appendix A for a brief description of each system. As a result, we selected a total of 105 papers classified into two groups attending to (1) those which refer to surveys (19 papers), and (2) those that tackle any of the concrete provenance systems classified for our review (86 papers).

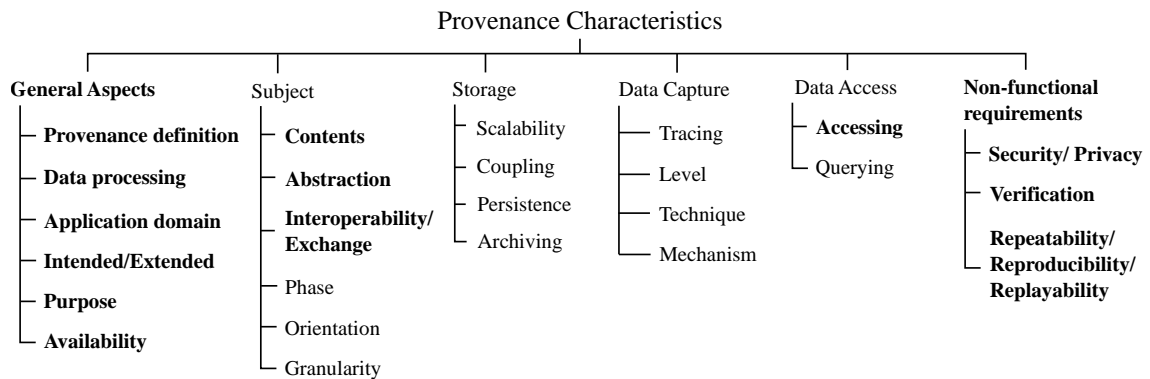


Fig. 3.1 Our taxonomy of Provenance Systems Characteristics

Relying upon the information extracted from such selected papers, we defined our taxonomy by enriching and revisiting the taxonomy proposed in Su15, since we considered it the most complete one. Additionally, we also analysed the remainder works, matching common aspects and incorporating additional ones not considered in Su15. In Figure 3.1, we present the resulting taxonomy in which we have depicted in bold those aspects that do not appear in the original taxonomy Su15. Our taxonomy establishes six dimensions for comparing provenance systems, attending to:

- *general aspects*, which are related to general background regarding provenance systems.
- *subject*, which refers to the different subjects or levels of detail in which provenance data can be represented, also considering interoperability aspects.

- *storage*, which describes the different approaches used by provenance systems to register provenance information.
- *data capture*, which deals with the way in which provenance data can be captured on the existing provenance systems.
- *data access*, which refers to how users can access provenance data repositories.
- *non-functional requirements*, comprising non-functional requirements of provenance systems (such as security, verification, and so on).

Additionally, in each dimension, we have identified several categories that correspond to particular aspects to focus on when comparing provenance systems. For example, regarding *data capture* in Figure 3.1, we can distinguish among *tracing*, *level*, *technique*, and *mechanism*.

3.3 Taxonomy of provenance systems characteristics: an overview

3.3.1 General Aspects dimension

As we previously stated, the provenance field has derived into a large and heterogeneous research corpus of approaches to address a variety of provenance concerns from different points of view. This dimension has six categories referring to general background regarding provenance systems.

Provenance definitions It refers to the definitions of provenance considered by the different authors in order to be analysed by means of the technique called References-enriched Concept Maps (RCM) [153]. This technique, inspired by Concept Maps [154], can be used to compare definitions and therefore improve the understanding of terms, keeping track of the publications in which the different definitions were proposed. Later, the RCM can be automatically created using the tool that Sáenz-Adán et al. presented in [44]. This analysis was performed based on 34 definitions of provenance that were obtained relying upon the identified surveys. Based on this analysis, which can be seen in [10], we can infer that there is not a clear consensus on a definition of the provenance concept.

Data processing It refers to the strategy followed to manipulate data. The most notable strategy followed is *Workflow Management Systems (WfMS)-based*. This strategy requires

instructions expressed in a specific process definition language and the registration or wrapping of external code. Most of the works that follow this strategy advocate declaring the entire workflow in advance using a WfMS (e.g., [S1] VisTrails, [S3] Kepler provenance, [S8] ZOOM), bridging the gap between experimentation and provenance management.

Application domains The selected systems have a wide variety of *application domains* ranging from meteorology (e.g., [S1] VisTrails) to geology (e.g., [S3] Kepler provenance) going through medicine (e.g., [S2] myGrid/Taverna). This fact gives an idea of how heterogeneous is the provenance field.

Intended/Extended This category aims at identifying if the system corresponds to a tool explicitly developed for provenance purposes (e.g., [S4] PASS and [S11] PASOA/PreServ) or if it is a tool which has been extended with provenance capabilities (e.g., [S1] VisTrails and [S3] Kepler provenance).

Purposes It refers to the *purpose* for which a system acquires provenance capabilities. Concretely, *understanding derived data* is one of the most important purposes when provenance capabilities are implemented (e.g., [S7] Chimera, [S9] Cui 2000, [S15] SPADE, and [S17] Perm-GProM).

Availability We note that 21 out of the 25 reviewed systems are released under an open-source license (e.g., [S8] ZOOM and [S17] Perm-GProM).

3.3.2 Subject dimension

Provenance systems may represent provenance data in terms of distinct subjects and different levels of detail. This dimension encompasses six categories associated with these aspects.

Contents This category only applies in the context of databases. It refers to the most common forms of *database provenance*, which describe relationships between data in the source and in the output, regarding the effect of queries. We have identified four main approaches: *lineage*, *where-provenance*, *why-provenance*, and *how-provenance*. *Lineage* considers tuples to belong to the provenance if they “contribute” to an output tuple [98]. *Where-provenance* refers to the identification of the source elements *where* the data in the target are copied from, that is, it refers to *where* output data came from in the input [53]. *Why-provenance* aims at explaining *why* a piece of data is being created. Finally, *how-provenance* explains how source data are involved in the creation of the result.

Abstraction It refers to the use of approaches which help structure the provenance information in order to deal with its complexity and size, easing the understanding of provenance

and making sense of it. Among the surveyed systems, we remark the *accounts* approach. *Accounts* allow for multiple descriptions of a given execution to co-exist in a provenance trace [56] (e.g., [S5] Trio).

Interoperability/exchange It is related to the great efforts made on the interoperability and exchange of data among the different provenance systems. Several attempts have been carried out in order to establish a model to particularly represent workflow executions as provenance records. In this category, the W3C PROV standard¹ stands out for the representation of provenance [2]. In order to add PROV capabilities, some systems have implemented a plugin (e.g., [S2] myGrid/Taverna or [S18] ES3), whereas other systems have developed export/import provenance functionalities (e.g., [S3] Kepler provenance and [S24] PLUS). It is worth noting that *interoperability* issues are discussed later in Section 3.4.

Phase It refers to the different moments in time in which provenance is captured. We distinguish between execution (or *retrospective* provenance) and composition (or *prospective* provenance). Between these two approaches, we remark the *retrospective* approach for being the most widely used. This approach is adopted by systems which capture information while the workflow is executing (e.g., [S11] PASOA/PreServ and [S18] ES3).

Orientation Provenance data can be either available explicitly (i.e., *data-oriented model*) or deduced indirectly (i.e., *process-oriented model*). In a *data-oriented model*, provenance meta-data is specifically gathered about the data product (e.g., [S20] DBNotes and [S25] RAMP). In a *process-oriented model*, the primary entities for which provenance is collected are the deriving processes, while the inputs and outputs of these processes are inspected to obtain the data provenance (e.g., [S11] PASOA/PreServ and [S22] REDUX).

Granularity It refers to the level of detail of a specific data product. More specifically, it is related to the types of objects for which a system maintains provenance. The granularity gathered by a specific proposal constitutes an aspect specially remarkable in such a proposal since it establishes the usefulness of provenance. We distinguish systems that capture provenance with a high level of detail (*fine-grained* provenance) from those that capture provenance with a low level of detail (*coarse-grained* provenance). Most of the systems capture *fine-grained* provenance (e.g., [S5] Trio and [S4] PASS), although we have considered that systems such as [S21] LipStick and [S24] PLUS can capture both *fine-grained* and *coarse-grained* provenance.

¹Background information about PROV is provided in Section 2.1.2

3.3.3 *Storage dimension*

This dimension is made up of four categories that describe the different approaches that provenance systems may use to record provenance information.

Scalability The stored information can grow larger than the data it describes due to several aspects such as the number of datasets, their granularity, the depth in the lineage, their geographical distribution, etc. [50, 59]. With storage scalability of the provenance data we refer to whether the data is stored in a *centralized* or *distributed* way. Most of the surveyed systems follow a *centralized* provenance storage (e.g., [S1] VisTrails and [S19] COMAD). Only eight of them store provenance data *distributively* (e.g., [S21] LipStick and [S24] PLUS).

Coupling This category describes the relationship between the provenance data and the data which is the target of provenance recording [51]. A *high-coupling* system stores provenance directly associated with the data for which provenance is recorded (e.g., [S3] Kepler provenance and [S20] DBNotes). Alternatively, a *loose-coupling* approach uses a mixed storage scheme where provenance and data are stored in one storage system but logically separated (e.g., [S8] ZOOM and [S9] Cui 2000). Finally, *no-coupling* systems store provenance information in one or many provenance repositories, which store only provenance data (e.g., [S12] Tioga and [S24] PLUS). The most used strategies are *high-coupling* and *loose-coupling*.

Persistence It refers to the expressiveness of the conceptual models used by a provenance management system to persist provenance. A provenance conceptual model defines what information is supported in a provenance system. A wide variety of persistence approaches have been used for storing provenance ranging from relational DB (e.g., [S5] Trio and [S7] Chimera) and XML dialects (e.g., [S18] ES3 and [S19] COMAND), to noSQL databases (e.g., [S23] BURRITO).

Archiving This category is related to the different approaches taken to archive provenance. Most provenance systems advocate using *time-stamps* to mark the existence of data products at various times (e.g., [S1] VisTrails and [S14] Buneman).

3.3.4 *Data Capture dimension*

The existing provenance systems follow different approaches to capture provenance data. This dimension states four categories that define how they do it.

Tracing The approaches adopted for tracing data provenance are *eager*, when the provenance is computed immediately (e.g., [S4] PASS and [S25] RAMP), and *lazy*, when it is computed on demand (e.g., [S9] Cui 200 and [S12] Tioga).

Mechanisms It refers to the mechanism used to capture provenance data [155]. Some systems advocate using *internal structures* for capturing data (e.g., [S4] PASS and [S23] BURRITO), whereas others rely upon *external services* (e.g., [S4] SPADE and [S23] PLUS). In particular, the *external services* strategy is adopted to collect provenance from both distributed and heterogeneous environments [155].

Level This category refers to the levels of capture; that is, the particular point in the software or application stack where the provenance is collected [155]. It is related to the relevant details of the computational tasks the capture mechanism needs access to. We have identified five levels of capture: *workflow-level*, *operating system (os)-level*, *process-level* (also called *activity-level*), *services-level*, and *database-level*. For their interest in this thesis, we will focus on the first three levels. The *workflow-level* is the most common proposal (e.g., [S1] VisTrails and [S13] Wings-Pegasus). At this level, each WfMS is responsible for gathering all the provenance information [155]. Thus, in systems adopting this level of capture the provenance gathering mechanisms are either attached to or integrated into a WfMS [46]. Another remarkable approach is the capture of provenance at *os-level* (e.g., [S4] PASS and [S18] ES3). In this level of capture, provenance is collected at the API system level, relying on the availability of specific functionality at the *os-level*. Finally, we note the *process-level* proposal. At this level, each process involved in a computational task is required to capture its own provenance information [155, 46].

Technique It is related to the techniques that existing provenance systems use to capture provenance [49, 155]. On the one hand, the *annotation* approach refers to the process of adding to or “making up” existing data (e.g., [S11] PASOA/PreServ and [S24] PLUS). Alternatively, the *inversion* method leverages a property by means of which some derivations can be inverted to find the input data used to derive the output data (e.g., [S8] ZOOM and [S12] Tioga).

3.3.5 Data Access dimension

Once the provenance data is captured, the provenance systems may provide a way through which provenance data can be accessed and explored. This dimension encompasses two categories to describe the way in which provenance is accessed and queried.

Accessing It refers to whether it is provided a way through which provenance data can be accessed and explored. There exist two main approaches for accessing data provenance, which are not mutually exclusive. Firstly, most of the systems allow data access by *browsing*, that is, visualising and navigating data and processes dependencies. To do it, the most commonly way to depict data relies on *derivation graphs* (e.g., [S4] PASS and [S15] SPADE). Secondly, provenance systems usually provide an *API* which allows users to implement their own usage mechanisms to access data [50] (e.g., [S2] myGrid/Taverna and [S15] SPADE).

Querying This category refers to the querying of provenance data. Concretely, it is related to two issues: (1) the way in which querying is *formulated*, and (2) the *query language* provided by the proposal, if any. As for the way in which querying is formulated, we distinguish between browsing data (*exploratory*) and using a directed language (*directed*). Regarding the *query language*, we draw a distinction between proposals that use languages not specifically designed for provenance (e.g, [S6] Karma and [S8] ZOOM, which use SQL, and [S15] SPADE, which uses Neo4J), and proposals that use their own querying language (e.g., [S4] PASS, which uses PQL, and [S7] Chimera, which uses VDL-SQL).

3.3.6 *Non-functional Requirements dimension*

This dimension refers to non-functional requirements of provenance systems by means of three categories.

Security/Privacy *Security* is especially important to guarantee the trustworthiness of provenance data. With data security, we mean the protection of data against unauthorized access or corruption. Security is necessary to ensure data integrity. In the context of provenance, security traditionally refers to aspects such as: (1) confidentiality, which is related to the conditions under which provenance data is shared and/or distributed in a controlled fashion, (2) integrity, since provenance data must be immutable, (3) unforgeability, since provenance data could not be forged without being detected, (4) non-repudiation, which ensures that the user cannot deny if she/he has taken any action, and (5) availability, which means that data must be available at any time from everywhere. *Privacy*, although related to security, has always concerned more than just confidentiality. It refers to the way in which sensitive information is managed. Applied to provenance, while security focus on the protection of data, privacy rather deals with permission and the desire of limiting the distribution of such existing provenance data. It is worth noting that most of the works published on provenance systems hardly mention security or privacy; they assume that both privacy and security rely on the host system or the file-system model (e.g., [S3] Kepler provenance and [S4] PASS).

Verification In this category we want to show whether the provenance proposals consider any way of formal verification. We uncovered that none of the surveyed systems uses formal verification methods neither to verify provenance data nor to check business processes. In fact, there is limited literature within the provenance context addressing formal verification [156].

Repeatability/Reproducibility/Replayability *Repeatability* relies on sufficient information for the original researcher or others to be able to repeat the study. *Reproducibility*, which is seen as a special case of *repeatability*, constitutes an interesting element within provenance since it allows users both to replicate a result starting with the same source elements and methods, and to see that a prior result can be confirmed. Finally, *replayability* allows the investigator to "go back and see what happened" [157]. Concretely, *reproducibility* is one of the characteristics that appears in almost all the surveyed provenance systems (e.g., [S13] Wings-Pegasus and [S22] REDUX). In fact, it is not considered a cause but rather a consequence of applying provenance capabilities within a provenance system.

3.4 Open problems

So far, we have given an overview of our systematic review, which plots the landscape of the different approaches to address a variety of provenance concerns. Thanks to this work, we have determined the following four open problems.

3.4.1 Integration

The integration of provenance capture with applications is a problem to face with when dealing with provenance. In fact, the effort required to perform this integration gives an idea about how much the provenance solution will intrude on users' modus operandis.

In this line, our systematic review uncovered two main approaches for capturing data in the least intrusive manner: *workflow-level* and *os-level* approaches in the *level* category of the *data capture* dimension (Section 3.3.4). The most notable strategy used by the surveyed systems is the *workflow-level* approach through the integration of provenance capabilities into a WfMS, bridging the gap between experimentation and provenance management [158]. These provenance systems advocate declaring the entire workflow in advance using a WfMS (e.g., [S1] VisTrails and [S8] ZOOM), which leads to little effort from the users. Concretely, it requires to move to a new environment, or wrapping external applications for their use in the WfMS. However, its dependency on the WfMS makes it difficult to use the same mechanism in other scenarios, being applicable only in non-heterogeneous scenarios where only one WfMS is used. Alternatively, some systems capture provenance data at the *os-level*.

This kind of systems leads to the lowest intrusiveness in addition to avoiding the modification of the target applications; however, they do not deal with distributed applications and they do not capture the high level meaning of application executions [159]. What is worse, since they capture provenance for all the executions, such provenance may provide too much irrelevant information and no particular nuances about the application workflow. In addition to these two approaches, although it is scarcely used, another approach is to capture provenance at *process-level*, but it has limitations that have hampered a broader adoption. In fact, none of the surveyed systems capture provenance at *process-level*. This approach has the advantage of being independent from the WfMS (as opposed to the *workflow-level*), and additionally, it is able to capture high level meaning of the process (unlike the *os-level*). The issue at this type of level of capture is the need of adapting pre-existing activities of the process to incorporate provenance collection functionalities [155].

Aiming at integrating provenance capture with applications in order to make them provenance-aware, the *Provenance Incorporation Methodology* (PrIME [11]) has been developed. It consists of eliciting provenance requirements, identifying the parts of the application that contain relevant information, and iteratively designing provenance, while modifying the application design so that it can generate the expected provenance. However, PrIME is standalone, and it does not integrate with existing software engineering methodologies, which makes it challenging to use in practice. In fact, Groth in [159] states as future work that “this methodology [11] needs to be improved through integration with common development methods such as those based on the Unified Modeling Language [1]. To assist in the usage of the methodology, tools for integrated development environments need to be developed. Taken together both improvements could engender an environment where making applications provenance-aware is a central part of any application development effort.”

3.4.2 Interoperability

It is worth noting the enormous effort made by the provenance community to develop a standard model for provenance data exchange. Existing attempts, such as the Provenance Challenges series [12], shown the importance of providing such a means of exchange. Concretely, the *second provenance challenge* was focused on understanding the interoperability of the approaches. Thirteen teams responded to the challenge, and this resulted in discussions about a common data model, which led to the proposal of the Open Provenance Model (OPM) [13]. Subsequently, the *third provenance challenge* aimed at identifying the weaknesses and strengths of OPM. Fifteen teams were involved, and their proposals resulted in a new version revisiting OPM. Finally, the *fourth provenance challenge* finished early as

a consequence of the creation of the PROV standard [2] by the World Wide Consortium Incubator on Provenance.

As we showed in the *interoperability* category of the *subject* dimension (Section 3.3.2), some systems such as [S2] myGrid/Taverna and [S18] ES3 have made an effort to add PROV capabilities by implementing a plugin, whereas other systems such as [S3] Kepler provenance and [S24] PLUS have developed export/import provenance functionalities. In this line, toolkits supporting PROV have been facilitating the software engineer's tasks. For instance, the *ProvToolbox* [16] is a Java toolbox for handling PROV; *ProvPy* [15] is a Python implementation of the PROV data model; *ProvExtract* [160] is a tool to extract PROV from web-pages; and *ProvVis* [161], which allows visualising PROV through different charts. However, such toolkits do not help decide what information should be included in provenance, and how software should be designed to allow for its capture. Therefore, the ability to consider the intended use of provenance, specially during the software engineering design phase, has become critically important to support the software designer in making provenance-aware systems.

Against this background, a remarkable solution developed on top of PROV is the PROV-Template approach [3]², which allows the design of the provenance to be generated by means of PROV *templates*. Although the PROV-Template approach reduces the development and maintenance effort, separating responsibilities between software and provenance designers, it still requires designers to be knowledgeable in provenance. Thus, an approach that automatically links the design of the provenance with the actual design of the application is desirable. This would lead to the design of applications as usual, but such applications would be automatically provided with provenance capabilities.

3.4.3 Computational overhead

Another remarkable aspect when choosing a provenance system is the computational overhead. Classical provenance data techniques and methods are usually data-intensive, resource and time-consuming [45, 60]. This imposes the need for providing techniques that introduce a minimum computational overhead [60]. Among the categories stated in our taxonomy, those that are closely related to temporal or spatial overhead are *granularity* from the *subject* dimension (Section 3.3.2), as well as *tracing*, *level* and *technique* from the *data capture* dimension (Section 3.3.4).

As far as the *granularity* category is concerned, the amount and cost of provenance information can be inversely proportional to the *granularity* [50]. More specifically, the

²Background information about the PROV-Template approach is provided in Section 2.1.2.

provenance information can grow to be larger than the data it describes if the data is *fine-grained* and the provenance information rich [50, 59]. Thus, how the provenance metadata is captured is important for its scalability.

In the particular case of *tracing*, it may incur in both temporal and spatial overhead. To follow an *eager* approach for computing provenance not only may cause run-time overhead for the transformation to compute the output, but also it requires additional storage space to store the produced provenance information. In contrast, a *lazy* computation does not result in additional storage space and run-time overhead. However, this approach is not applicable for all types of transformations and it can slowdown provenance retrieval [46, 55, 155].

Similarly, the *level* of capture is directly related to the spatial overhead. The capture of provenance at the *os-level* provides the possibility of recording a low level of the metadata completeness, which would be difficult or impossible to achieve with the *workflow-level* or even with the *process-level* solution. However, this approach collects provenance information for all the executions, leading to capture more provenance than required to satisfy specific provenance requirements, and resulting in unnecessary overheads. Conversely, the storage overhead for systems capturing provenance at the *workflow-level* and the *process-level* depends on the amount of data per operation and the number of recorded operations [162]; thus, it is believed that they incur in less spatial overhead than the *os-level* approach. As for time overhead of *workflow-level* approaches, they usually provoke a minuscule overhead because of their limited approach to collecting running process information [162]. Finally, for systems that record data at the *process-level*, provenance capture costs are related to the cost of intercepting and recording observable operations [162].

The *technique* in *data capture* may also incur in temporal and spatial overhead. Concretely, using the *annotation* approach is useful when the source data are unavailable after transformation, but it takes more time and space for executing and storing the information than the *inversion* approach. Indeed, the *inversion* approach does not have any performance or storage overhead during data transformation. However, it cannot compute provenance when the source is unavailable after its transformation.

Carata et al. in [162] claim that it is often essential to grasp if the overhead costs imposed by the execution of provenance systems are acceptable. In fact, it is useful to predetermine what provenance information will be required to answer queries, and at what granularity this information will be sufficient. All in all, it is still considered an open question to reach the appropriate mechanisms which find a balance between both the capture of provenance and the computational overheads.

3.4.4 Querying

As stated in [162], using a provenance system is as useful as the questions that someone can answer based on the collected provenance. In fact, how to access such a provenance is a well-known challenging problem [57, 163]. As we stated in the *querying* category of the *data access* dimension (Section 3.3.5), there are two main approaches to address it: *exploratory* and *directed*.

The *exploratory* approach is used when users do not have an exact idea of what metadata they might want to retrieve. Thus, these systems usually provide both a visual representation of the provenance graph, and tools to explore it without succumbing to information overload. Conversely, when users know precisely what information they might want to query, the *directed* approach is the most effective. Nevertheless, they must be knowledgeable about the specific query language of the persistence system. Concretely, in the *persistence* category of the *storage* dimension (Section 3.3.3) we gave some remarks about such an aspect. Basically, the analysis we have performed on the provenance systems showed that there is a wide variety of persistence systems, which has been motivated by the development of new approaches which rely their data storage on new persistence systems based on, for example, XML or noSQL, against older systems which usually rely on relational databases. This diversity leads users seeking to exploit provenance data to have to learn the query language of the persistence system, which is not desirable.

Against this background, we advocate for provenance solutions agnostic about any storage system, so that users can decide which storage system best suits their needs, and consequently, the query language. In this way, users do not need to learn a new query language, and what is more, they leverage their previous experience in any query language for exploiting the provenance data.

3.5 Conclusions

In this chapter, we gave an overview of a systematic literature review of studies, identifying a comprehensive set of 105 relevant published works in the provenance field. The results showed that there are common aspects or characteristics of provenance systems widely renowned throughout the literature on the topic. Based on these results, we defined a six-dimensional taxonomy of provenance characteristics attending to: general aspects, subject, storage, data capture, data access, and non-functional aspects. Additionally, the study has found that there are 25 most referenced provenance systems within the provenance context.

It is our belief that this work can provide several benefits for potential researchers, in general, and for this thesis, in particular.

Our systematic review provides a comprehensive and consistent background of provenance concepts and techniques that will allow potential readers to grasp this memoir. Moreover, does not only it report about the available approaches and technologies in the field of provenance, but it also justifies the set of uncovered open problems that serves as foundations for this thesis.

Chapter 4

Conceptual definition of UML2PROV

Taking into account the open problems uncovered in the previous chapter, here we provide the reader with an explanation about the conceptual definition of our approach: UML2PROV. To do this, first we motivate the need for UML2PROV by referring to the aforementioned stated open problems. Later, we move on to describe the UML2PROV architecture. Subsequently, we will explain its foundations and finally, we will finish this chapter with the conclusions.

The work explained in this chapter has been partially presented in:

- Carlos Sáenz-Adán, Beatriz Pérez, Trung Dong Huynh, Luc Moreau: *UML2PROV: Automating Provenance Capture in Software Engineering*. In the International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM) 2018. [164]
- Carlos Sáenz-Adán, Luc Moreau, Beatriz Pérez, Simon Miles, Francisco J. García-Izquierdo: *Automating Provenance Capture in Software Engineering with UML2PROV*. In the International Provenance and Annotation Workshop (IPAW) 2018. [165]

Additionally, an extended and enriched version of the previous works has been submitted for publication in:

- Carlos Sáenz-Adán, Beatriz Pérez, Francisco J. García-Izquierdo, Luc Moreau: *Integrating Provenance Capture and UML with UML2PROV: Principles and Experience*. Submitted for publication in IEEE Transactions on Software Engineering.

4.1 Motivation for UML2PROV

As a result of the previous systematic review, we identified that one of the significant hurdles in the provenance field is how to *integrate* the capture of provenance with applications. Concretely, the ability to consider the intended use of provenance during the software engineering lifecycle, and particularly in the design phase, has become critically important to support the software designer in making provenance-aware systems. As we mentioned previously, in the realm of provenance, PrIme [11] can be considered as the first attempt to develop a methodology for making applications provenance-aware. However, it is standalone, since it is not integrated with existing software engineering methodologies. Similarly, the PROV-Template approach [3]¹ allows the design of the provenance to be generated by means of PROV templates; however, it requires a PROV expert during the design phase of the application. Conversely, in the Software Engineering context, *UML* [1]², a modeling language that has become the software engineers' *lingua franca*, offers no specific support with regard to provenance. In fact, the inclusion of provenance within the UML design can entail significant changes to the original application design, which can make it unreadable and difficult to maintain.

Against this background, UML2PROV is intended to bridge the gap between application design, specified by *UML* diagrams, and provenance design, based on of PROV templates. Concretely, with UML2PROV a software engineer is able to design an application as usual, by following the UML specification, and once the design is ready, UML2PROV comes into play for (1) generating the design of the provenance to be generated (in the form of PROV templates), and (2) creating artefacts that need to be linked with the application to collect provenance. In short, UML2PROV avoids the designers and developers not only to be knowledgeable about provenance, but also to having to deal with complex diagrams, and to maintain an application's provenance-specific source code. The generation of provenance is transparent to them. These benefits, which will appeal to designers in early stages of the development process, are mainly related to the following uncovered open problems.

- *Integration*. With UML2PROV, we provide a way to include provenance capabilities during the design phase without changing the way in which software designers use UML. During the design phase, software designers only have to deal with the design of the system UML diagrams without needing to be knowledgeable about provenance aspects, since provenance generation is handled automatically from such diagrams. Finally, it is worth mentioning that UML2PROV complements PrIme by integrating the design of

¹Background information about the PROV-Template approach is provided in Section 2.1.2.

²Background information about UML is provided in Section 2.1.1.

provenance, by means of PROV templates, with the design of applications, using well-known software engineering language, namely UML. This prevents software designers from being knowledgeable about provenance during the design phase of the application.

- *Interoperability*. UML2PROV advocates representing the provenance design following the PROV-Template approach. This incurs in several benefits associated with the PROV-Template approach: separation of responsibilities, provenance maintenance, potential runtime checks and static analysis, and provenance consumption [3]. Additionally, since the PROV-Template approach is intended to generate provenance information following the PROV standard, it also leads to several benefits in terms of interoperability. For instance, the provenance generated by UML2PROV could be shared and used by other systems supporting PROV. Some of these systems could be *ProvToolbox* [16] and *ProvPy* [15] for managing the provenance by means of Java or Python, respectively; or *ProvVis* [161] for visualising the provenance information using different types of charts, among other systems. Additionally, it is worth noting that the generated provenance could be used for Linked Data, since the PROV standard provides an OWL2 ontology allowing the mapping to RDF [22].

4.2 UML2PROV architecture

We now overview the architecture we propose for UML2PROV relying on Figure 4.1. To do this, we start by presenting the stakeholders, focusing on their tasks and roles in UML2PROV. Later, we explain the key facets of the architecture. Finally, we move on to describe the overall process of UML2PROV for making applications provenance-aware.

4.2.1 Stakeholders

As Figure 4.1 illustrates, the stakeholders involved in UML2PROV are: the *software designer* and the *software developer*, at the beginning, and the *provenance consumer*, at the end. Since providing a complete description of the usual tasks and duties of each stakeholder is beyond the scope of this document, we only expose those aspects of interest from the UML2PROV's architecture perspective. Concretely, *software designers* are responsible for designing the system, or an excerpt of it. *Software developers* write the application's source code relying on the design of the *software designers*. Finally, *provenance consumers* are the people who exploit the collected provenance.

At this point, it is worth noting that both *software designers* and *developers* may not be familiar with provenance, they just perform their usual tasks. Similarly, *provenance*

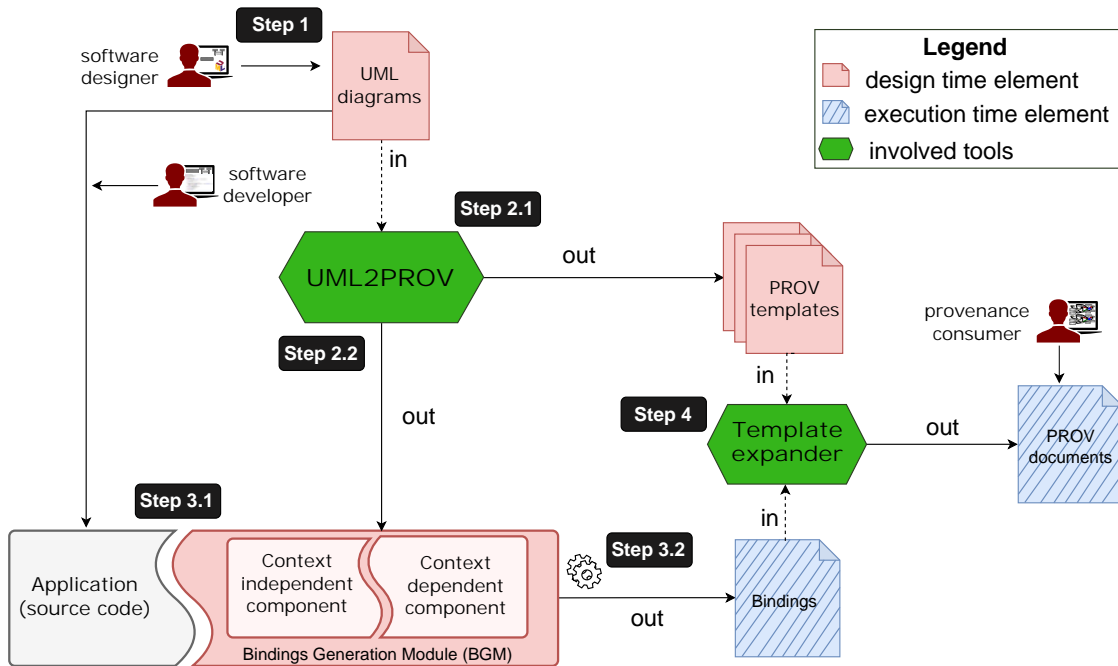


Fig. 4.1 The architecture of UML2PROV

consumers do not need to know how to design systems to exploit the provenance. Finally, we remark that, although having identified three different stakeholders, the same person may play several roles at the same time. For instance, a person may be in charge of designing the application (*software designer*), and also of exploiting the collected provenance (*provenance consumer*).

4.2.2 Key facets

As for the key facets making up the architecture of UML2PROV, we distinguish those that are created at *design time* (red plain background in Figure 4.1) from those generated at *runtime* elements (blue background with striped texture).

On the one hand, the *design time* facets encompass three key elements:

- *UML diagrams*, which constitute the design of the application conforming to the UML specification [1] (see background information in Section 2.1.1). Among the wide range of types of UML diagrams, UML2PROV takes as source those that not only have a strong relation with provenance, but that also are mostly used by *software designers* [17]: (i) Sequence Diagrams (SqDs) and State Machine Diagrams (SMDs), because they are widely used to represent the behaviour of a system (one of the main purposes of capturing provenance information); and (ii) Class Diagrams (CDs), since they are the most widely

adopted formalism for modeling the intentional structure of a software system (that is, low level aspects from objects' internal status, information not given by the considered behavioural diagrams). All in all, taking these UML diagrams as source provides significant benefits. First, we are able to address a wide range of applications, since we have considered the most commonly used UML diagrams for designing systems. Secondly, we obtain provenance from different perspectives, due to the fact that UML2PROV supports several types of UML diagrams.

At this point, we note that more than a nuance in terminology, the distinction between the *state* and the *status* of an object is crucial to understand this thesis. Concretely, in accordance to UML terminology, in UML State Machine diagrams the *state* of an object denotes a situation during which some invariant conditions holds [1]. In UML Class diagrams, to avoid confusion, we use the term object's *status* with a broad scope, referring to the values of the object's attributes at some moment, which particularly could correspond to a concrete *state*.

- *PROV templates*, which express the shape of the provenance to be generated. Briefly speaking, they are PROV documents containing variables that will be replaced by values³.
- *Bindings Generation Module (BGM)*, which includes the code for capturing provenance data in the form of *bindings* (explained below). It consists of two components: the *context-dependent component*, which includes specific code for a concrete application, and the *context-independent component*, which contains code that is common to all applications.

On the other hand, the *runtime execution* facets are made up of two key elements:

- *Bindings*, which associate concrete values collected from the execution (provenance data) with variables from a PROV template³.
- *PROV documents*, which contain the provenance information suitable for consumption. These documents conform to the PROV standard³.

4.2.3 How to use UML2PROV

The overall process of UML2PROV for making applications provenance-aware, in which the aforementioned stakeholders and facets are involved, consists of four main steps, which have been identified with a black label in Figure 4.1:

Step 1 *Software designers* model the application by means of structural and/or behavioural *UML diagrams*, conforming to the UML specification.

³Background information about the PROV-Template approach is provided in Section 2.1.2.

- Step 2** Taking the UML diagrams as source, UML2PROV automatically produces both the *PROV templates* with the design of the provenance to be generated for the concrete application (**Step 2.1**), and the specific *BGM* responsible for generating the set of *bindings* during the application execution (**Step 2.2**).
- Step 3** The *BGM* is integrated into the existing application in a non-intrusive manner, so that no changes to the application are required (**Step 3.1**). Subsequently, while the application is running, the *BGM* comes into play collecting provenance data in the form of *bindings* (**Step 3.2**).
- Step 4** The *template expander* [3] takes both the *PROV templates* and the set of *bindings*, replaces the *templates*' variables by the concrete values from *bindings*, and generates a set of *PROV documents* for the executed application that are ready to be exploited by the *provenance consumers*.

In conclusion, UML2PROV takes as source a subset of *UML diagrams* modeling a system (Sequence, State Machine, and Class diagrams) and automatically generates (1) the *PROV templates* with the design of the provenance to be generated, as well as (2) the module (i.e., the *BGM*) for collecting *bindings* during the execution of the application. Therefore, UML2PROV is intended to be used with applications modeled by UML diagrams. These UML diagrams could have been developed from scratch during the design phase of the software development lifecycle; or alternatively, they could have been generated once the system is developed by applying *reverse-engineering* (we will discuss this aspect further on). Finally, we note that one of the main advantages of taking as source UML Sequence, State Machine and Class diagrams, which provide different perspectives of a system, is the generation of provenance with different viewpoints. Nevertheless, this fact does not mean that UML2PROV must be used taking as source all types of diagrams from those supported, but a potential user of UML2PROV may choose the perspective (or perspectives) of provenance to be generated by selecting the type(s) of diagram to be processed.

4.3 From UML to PROV: the transformation patterns

Our conceptual proposal to generate PROV templates from UML diagrams (Step 2.1 in the architectural overview) has been rigorously defined by an extensive set of 17 *transformation patterns* (or *patterns* for short) that ultimately associates UML elements with PROV elements. Below, we begin by discussing why we have selected some UML elements to be translated, and why we have ruled out others. Then, we state the set of principles we have followed for

the definition of these patterns. Subsequently, we explain the common structure we have used for describing them. Later, we move on to presenting a generic overview of the patterns. At the end, an illustrative example is presented to explain how the patterns are applied.

4.3.1 Considered UML elements

One of the main reasons for UML2PROV supporting three different types of UML diagrams is the common requirement in provenance systems for providing different viewpoints of a situation [14]. This fact forces us to choose a cornerstone situation common to all the supported types of UML diagrams. Our proposal advocates designing provenance for different perspectives of *operations executions* for two main reasons: (1) all the supported types of diagrams provide elements for modeling different perspectives of them (later, we will see what these elements are), and (2) the execution of concrete behaviour (execution of operations) drives the provenance of a piece of data. Concretely, those templates generated from Sequence diagrams will reflect how collaborating objects interact for executing operations, and the exchange of information between them. The templates generated from Class diagrams will contain specific information about (1) the objects' characteristics at some point, i.e. the object's *status*, and (2) the operations that have led the objects' *status* to be as they are. Finally, the templates obtained from State machine diagrams will show information about the evolution of the objects' state as a consequence of operations executions taken place.

UML Sequence diagrams

Aiming at obtaining PROV templates with information about how collaborating objects interact for executing operations, and the exchange of information between them, we have selected the following elements: *lifelines*, *messages*, *arguments*, and *execution specifications*. Concretely, we have selected *lifelines* because they expose information about the participants involved in the interactions. This information will allow us to reflect in PROV not only the information about the participants but also their responsibilities in the execution of operations. We have taken into account *messages* due to the fact that they may model operation calls. *Messages* provide us with information about the object that executed an operation, and the object that called such an operation. Likewise, we translate *arguments*, which are contained in *messages*, because they represent the data that are exchanged between the collaborating objects. Finally, we have selected *execution specifications* because, among other things, they represent concrete features of operations executions, which are the cornerstone of the transformations into PROV (later, we will provide more information about this aspect).

Conversely, we have ruled out those UML elements that are not of interest from our point of view, or those whose semantics could be inferred from the provenance exposed by the addressed UML elements. For example, we have not taken into account UML *combined fragments*, which are commonly used in SqDs to explicitly model various control structures. For example, the *combined fragment* with operator *alt* shows an alternative interaction; the operator *opt* models an optional interaction; or *loop* specifies an iterative interaction. The provenance information that could be exposed from these elements (and the remaining operators) is indirectly collected during the execution. For instance, in case of alternative (or optional) interactions, a provenance consumer will be able to know which alternative (or optional) interaction has been executed because the collected provenance only has information about the executed interactions. In case of loops, the provenance would contain the information about how many times the loop has been executed. The same applies to chronological aspects represented in the SqDs, the provenance collected from the execution will have information about the order in which the operations were called, so that we do not need to reflect such aspects in the templates.

UML Class diagrams

From CDs, we are interested in obtaining PROV templates about (1) the objects' characteristics at some point, i.e. the object's *status*, and (2) the operations that have led the objects' *status* to be as they are. To do this, we have taken into account the following elements: *classes*, *attributes*, *operations*, and *parameters*. We have identified *classes* because they are the main element of CDs, which are a means of classifying objects attending to their characteristics, represented by *attributes*, and behaviour, modeled by *operations*. This set of elements, together with the *parameters* inside the *operations*, will allow us to obtain PROV templates with information about the objects' *status* and how the execution of operations affect such a *status*.

Whilst the aforementioned elements are widely used in CDs, there are also commonly used UML elements that have not been taken into account. For instance, *abstract classes* and *interfaces* because they are not instantiated. As for *associations*, we do not consider the concrete semantics of *shared* and *composite aggregations*, they are translated as usual *binary associations*. We consider that their semantics are not of interest from the provenance point of view. Similarly, we do not directly address *generalization relationships* in the transformations, although we represent their semantics. Concretely, when we address the translation of a UML *class*, we identify its characteristics (*attributes* and *operations*) as well the characteristics of its superclasses, if any.

UML State Machine diagrams

In order to produce PROV templates with information about the evolution of the objects' state as a consequence of operations executions taken place, we have selected *states*, *initial pseudostates*, and *events* to be translated into PROV. Firstly, among the different types of *states*, we have chosen *simple states*, later we will explain why we have ruled out *composite* and *submachine states*. *Simple states* will allow us to generate PROV templates with information about an object's situation in which some invariant conditions hold [1]. Secondly, we have selected *initial pseudostates* because, among other things, they may refer to the execution of the operation that creates an object. We consider that this specification is crucial from the provenance point of view, since we are interested in determining when an object is created and what is its first *state*. Finally, *events* may be used to specify that an operation execution that triggers a change in an object's state has taken place. This information will allow us to obtain PROV templates with information about an operation execution from a State Machine diagram perspective.

Transitions can have a *guard* condition that indicates if the *transition* can even be considered (enabled), a *trigger* specified by an *event* that causes the *transition* to execute if it is enabled, and the *effect* that will be executed when the *transition* occurs [19]. As we mentioned previously, we only have considered *events*, and therefore, we have ruled out *guards* and *effects*. Concretely, we have not considered *guards* because the collected provenance only has information about the executed *transitions*, and such *transitions* are executed because the *guards* were evaluated as true. Thus, a provenance consumer could infer that the condition inside a *guard* was true when the corresponding *transition* was executed. We made the decision of considering neither *effects* nor *internal activities*, whereby an activity may consist of multiple actions [18]; however, we note that future research considering these elements might provide finer-grained provenance. Finally, we note that in our transformations we have mainly focused on *simple states*, instead of on *composite* or *submachine states*. This is because SMDs with *composite* or *submachine states* may be transformed into SMDs with only *simple states* by resorting to a flattening process, which is a very common approach in the realms of model checking and code generation [166]. However, since the users might be interested in representing *composite states* directly into the PROV templates, the description of our patterns also provide an overview of how *composite states* can be mapped to PROV.

4.3.2 Principles for the definition of UML to PROV patterns

The patterns have been defined taking into account a set of principles that are intended to facilitate their understandability and maintenance, in addition to enabling other researchers to replicate and implement their functionality. These principles are the following:

Consistency. Since UML diagrams model a system (or an excerpt of it) from different perspectives (depending on the diagram used), the resulting PROV templates will represent the provenance aspects associated with the perspective at hand. Although modeling different perspectives of a system, UML diagrams can have a rich semantic overlap. Similarly, the PROV templates generated from the UML diagrams of a system must have common elements which allow the merging of the expanded PROV documents.

Level. The PROV templates obtained from the patterns contain the design of the provenance to be generated and therefore, they state the *level* of capture of provenance (see Section 3.3.4); that is, the particular point in the software stack where the provenance must be collected. Due to the fact that the resulting PROV templates contain information about operations executions, the provenance must be collected at *process-level*. At this level, it is possible to obtain information regarding the operations executions.

Understandable. Each pattern identifies a UML design specifying a concrete situation of the system that is related to an operation execution (hereinafter we refer to this situation as *context*), and whose translation to PROV is provided. Aiming at facilitating the reader the understandability of the situation modeled by the UML design, each pattern must provide the reader with a description of the *context*, by using the natural language and software engineering terminology. From our point of view, this has two advantages. First, it gives an insight into the design of the provenance obtained from the UML design. Second, it may help users barely familiar with UML find the pattern that best suits their purposes.

Self-explanatory. Each pattern must be easily understood from the information given and not needing further explanation. As a consequence, one who reads all the patterns from beginning to end will find that some explanations bear a strong resemblance, even that there are repeated ones. We have made this decision to avoid that a potential user loses part of the explanations that are discussed elsewhere in other patterns.

Systematic. The explanation of the patterns must be uniformly structured. This will allow users to easily navigate the document and compare the patterns.

Finally, we note that potential designers (or developers) may extend the stated set of patterns, under their responsibility, by following the aforementioned principles.

4.3.3 Structure of the patterns

Firstly, we have assigned to each pattern a unique identifier. This identifier is an acronym that refers to the type of UML diagram together with a numeric identifier. The UML *Sequence diagram* Patterns are referred to as *SeqP<N>*, where *N* is the numeric identifier. Likewise, *CIP<N>* corresponds to the UML *Class Diagram* Patterns, and *StP<N>* to the UML *State Machine* Patterns. Subsequently, following the aforementioned principles, we have defined a common structure for all the patterns. This structure is made up of four blocks: *Context*, *UML diagram*, *Mapping to PROV*, and *Discussion*. For illustrating the above explanation of these blocks, we will use excerpts of the definition of the *SeqP1*, which are located inside a frame to avoid confusion with the explanation. The complete explanation of *SeqP1*, gathering all the following blocks together, is given in Appendix B.

Context The explanation of the situation addressed by the UML representation identified in the pattern. With the aim of improving the understandability of the context, we use the natural language by including well-known software engineering terminology (e.g., *object*, *operation*...). Additionally, each pattern context block includes a detailed description of its identified *key elements*, which are modeled in the UML representation.

Context block in *SeqP1*

A participant (the sender) interacts with another participant (the recipient) by calling an operation in the recipient, and then, it continues immediately. The call causes the recipient to execute the operation.

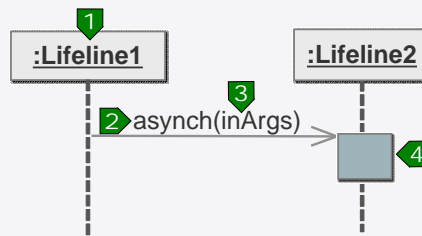
Key elements

<i>Sender</i>	The participant that makes the operation call.
<i>Operation call</i>	The call that starts the execution of the operation.
<i>Input data</i>	The information (if any) passed to the operation through the <i>Operation call</i> .
<i>Operation execution</i>	The execution of the operation.

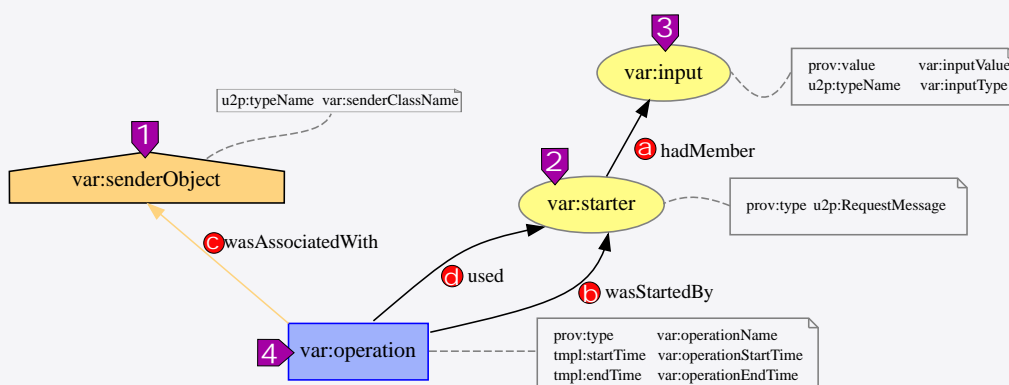
UML Diagram The excerpt of the UML diagram whose translation into PROV is ruled by the pattern. We provide a table (see the frame below) that explains the reasons for using each UML element that models each *key element* (column *rationale*). Additionally, we assign a green label containing a numeric identifier (e.g., 1) to each UML element, which makes it easier its location in the UML diagram. We remark that not all the elements in the UML diagram are mandatory. Some patterns identify specific UML elements to provide a more comprehensive translation (e.g., input or output data of operations or messages in some patterns).

UML diagram block in *SeqPI*

Key Element	UML	Rationale
<i>Sender</i>	Lifeline 1	It models the <i>Sender</i> participant involved in the interaction.
<i>Operation call</i>	Asynchronous Message	It models the <i>Operation call</i> when the <i>Sender</i> does not wait for a response, but instead continues immediately after sending the message.
<i>Input data</i>	Input Arguments	They specify the information passed to the operation through the <i>Operation call</i> .
<i>Operation execution</i>	ExecutionSpecification	It shows the period of time that the recipient's participant devotes to the <i>Operation execution</i> .



Mapping to PROV The PROV template proposed as translation for the previous excerpt of *UML Diagram*. This template is accompanied by an explanation about that transformation; that is, how the *PROV elements*, *attributes*, and *PROV relations* are generated from the elements in the *UML Diagram*. Each generated PROV element has a numeric identifier inside a purple label (e.g., 1) that corresponds to the identifier of the *UML element* from which it comes from (e.g., 1).

Mapping to PROV block in *SeqPI*: PROV elements (I)





Mapping to PROV block in *SeqPI: PROV elements* (II)


UML	PROV / id	Rationale
Lifeline 1▶	prov:Agent 1▶ / var:senderObject	The sender Lifeline 1▶ is mapped to a prov:Agent identified by var:senderObject. It assumes the responsibility for starting the ExecutionSpecification 4▶.
Asynchronous Message 2▶	prov:Entity 2▶ / var:starter	The Asynchronous Message 2▶ that initiates the ExecutionSpecification 4▶ of the recipient is a prov:Entity with identifier var:starter.
Input Arguments 3▶	prov:Entity 3▶ / var:input	Each argument of Input Arguments 3▶ is a separate prov:Entity identified as var:input.
ExecutionSpecification 4▶	prov:Activity 4▶ / var:operation	The ExecutionSpecification 4▶ is a prov:Activity with identifier var:operation.

Mapping to PROV block in *SeqPI: attributes* (I)





PROV Element	Attribute / Value	Description
var:senderObject 1▶	u2p:typeName / var:senderClassName	The value var:senderClassName is the string with the name of the class to which the var:senderObject 1▶ belongs.
var:starter 2▶	prov:type / u2p:RequestMessage	The value u2p:RequestMessage shows that var:starter 2▶ is a request message.
var:input 3▶	prov:value / var:inputValue	The value var:inputValue is the direct representation of var:input 3▶.
	u2p:typeName / var:inputType	The value var:inputType is the string with the name of the class to which var:input 3▶ belongs.

Mapping to PROV block in *SeqP1: attributes* (II)

PROV Element	Attribute / Value	Description
var:operation 	prov:type / var:operationName	The value var:operationName is the name of the operation var:operation  .
	tmpl:startTime / var:operationStartTime	var:operationStartTime is an xsd:dateTime value for the start of var:operation  .
	tmpl:endTime / var:operationEndTime	var:operationEndTime is an xsd:dateTime value for the end of var:operation  .

Additionally, each *relation* among PROV elements appearing in the PROV template is labelled with a letter (e.g., ) that helps link such a relation with its description. At this point, we note that in PROV two relationships of the form (B, prov:used, A) and (C, prov:wasGeneratedBy, B) may be enriched with (C, prov:wasDerivedFrom, A) to explicitly express that C was derived from A. This is a well-known construction called *use-generate-derive triangle* [167], which explicitly associates a *generated* prov:Entity with a *used* prov:Entity. In the definitions of the patterns, this construction may be applied in those templates in which a prov:Entity is *used* by a prov:Activity, and such a prov:Activity *generates* another prov:Entity. Nevertheless, in order to avoid the overburden of the PROV template explanations by including information that could be inferred, we only include the relation prov:wasDerivedFrom only when we have considered that such a relation enriches the provenance to be generated.

Mapping to PROV block in *SeqP1: PROV relations*

PROV Relation	Description
 prov:hadMember	It states that var:input is one of the elements in var:starter.
 prov:wasStartedBy	var:operation is deemed to have been started by var:starter.
 prov:wasAssociatedWith	It is the assignment of responsibility to var:senderObject for var:operation.
 prov:used	It is the beginning of utilizing var:starter by var:operation.

Discussion Issues related to the transformation of UML to PROV. Basically, we explain and justify the decisions we have made in the transformations addressed by the pattern.

Additionally, we discuss alternative solutions (if any) and some questions that are likely to come up to the reader.

Discussion block in *SeqP1*

- The UML diagram in this pattern depicts the responsibility of the *Sender* lifeline (`var:senderObject`) for the recipient lifeline to execute the operation (`var:operation`). However, the recipient lifeline is not modeled in this PROV template, even though it is the participant that executes the operation. This decision is based on other patterns' better ability to both (1) identify the participant responsible for executing that operation, and (2) give a more detailed information about the implications that the execution of that operation has in the recipient participant. More specifically, these patterns are: *StP1-StP3*, which mainly focus on representing possible changes in an object's state caused by an *Operation execution*; and patterns *CIP1-CIP10*, which put more stress on how the execution affects the *status* of the object responsible for performing such an execution.
- [...]

4.3.4 Overview of the defined transformation patterns

Our approach identifies a total of 17 patterns, 4 of them addressing SqDs, 10 addressing CDs, and 3 addressing SMDs. These patterns have been defined in a systematic and a thoroughly way by following the principles stated in Section 4.3.2, and according to the structure described in Section 4.3.3. Since each pattern has been defined in a *self-explanatory* and *systematic* way, their sequential reading from the first to the last could be considered a tedious task. This fact prevents us from including all the patterns' explanation in this section. Instead, in Appendix B we present five patterns used throughout this memoir (*SeqP1*, *SeqP2*, *CIP10*, *StP1*, and *StP3*), and in [168], we provide the explanation of all the patterns together.

In this section, we will introduce a set of charts built on top of the patterns' contexts for helping users find the pattern of their interest, and to give an insight into the wide range of situations they cover. Concretely, we have designed a chart for each type of UML diagram supported by UML2PROV (figures from 4.2 to 4.4). These charts provide several benefits. Firstly, they guide users to find the pattern that best suits their needs, preventing them from performing the tedious task of reading all of the patterns sequentially. Secondly, all the patterns provided for a concrete type of UML diagram are presented in the same figure. In this way, users are not only able to easily compare the patterns, but also they can easily get an idea of the wide range of situations that are addressed by the stated patterns. Finally, we remark that to facilitate the access to the explanations of the patterns included in Appendix B, the nodes in those charts representing such patterns include the number of the page in this document that contains the pattern's explanation, and a hyperlink to it (the latter only available in digital format).

UML Sequence Diagrams Patterns

Sequence diagrams are used to model the interactions among collaborating objects and the exchange of information between them; thus, we have focused on the interaction between components by means of operations' calls, and the consequences of such calls. We have defined four patterns identified from *SeqP1* to *SeqP4*. As shown in Figure 4.2, there are two cornerstone patterns when a sender participant calls an operation: (1) *SeqP1* when the sender does not wait for a response and it continues immediately, and (2) *SeqP2* when the sender waits for a response. Besides, both patterns may be complemented only by *SeqP3* when, during the execution of the called operation, a nested operation is called and the execution continues immediately, or may be complemented by *SeqP3* as well as *SeqP4* when a nested operation is called and additionally the execution waits for a response.

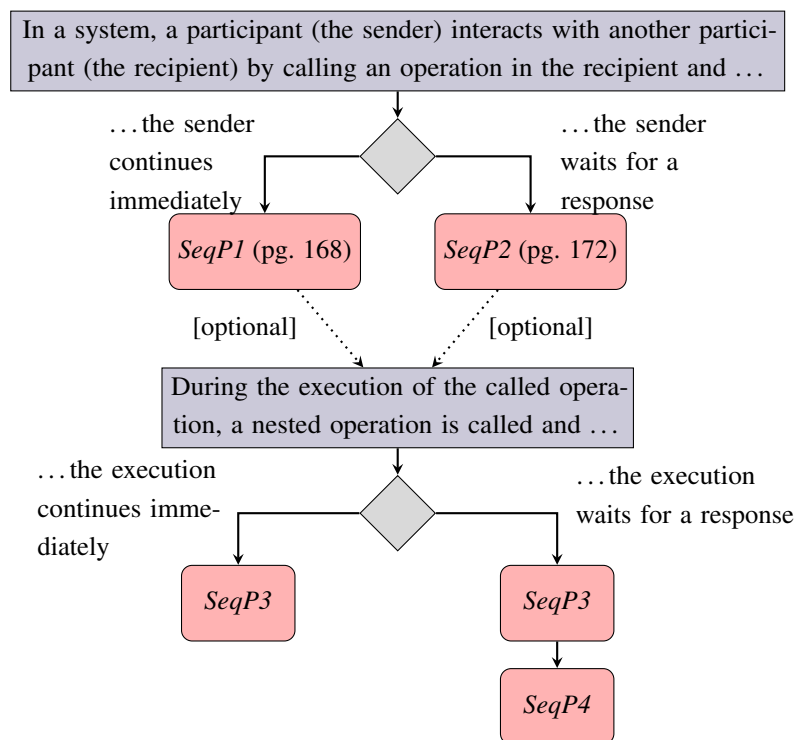


Fig. 4.2 This chart is focused on those patterns that rule the transformation of SeqDs

UML Class Diagrams Patterns

Class diagrams model the static structure of a system by means of UML *classes*, and their relationships, which classify objects and specify the features that characterise the structure and behaviour of those objects [1]. Concretely, aiming at generating PROV templates with detailed information about the changes performed within the objects modeled by *classes*, we

are interested in information about (1) the object's *status* (i.e., values of the *attributes*) before and after the execution of an operation, and (2) the object's internal changes (e.g., setting a new value of an *attribute*, or adding/removing an element in a collection *attribute*). However, the different nature of the operations prevents us from defining a generic transformation for all the UML *operations*. Depending on their nature, operations implicitly have specific semantics that can also provide information of interest for provenance capture. For instance, the key factors involved in the execution of an operation such as `getID` (which would return information about an *attribute*) are different from the ones related to a `setID` operation (which would set the value of an *attribute*).

Taking this into account, we made the decision of considering different types of operations aiming at providing meaningful provenance which explains the behaviour of each UML *operation*. To do this, we performed a literature review of studies identifying categorisations of operations in general (not necessary UML *operations*) based on their behaviours. The identified approaches showed that the behaviour of operations can be seen from different levels of detail. For instance, the classification presented in [1] differentiates operations that leave the state of the system unchanged from those that have side effects in the system. Conversely, the taxonomy provided in [169] gives a finer-grained classification based on aspects such as concurrency, synchronization, polymorphism, and so on. However, we remark the taxonomy given in [6] for being the most complete one, in addition of being agnostic about any specific context of application. Concretely, the taxonomy that we have used (which we explain below) is based on such a taxonomy.

A taxonomy of class' operations As we have mentioned, our taxonomy was built upon the taxonomy presented in [6]. Such a taxonomy defines five categories, each one containing UML *stereotypes* for identifying each type of behaviour. The five stated categories are the following:

- The *creational* category refers to operations responsible for creating or destroying objects of a class.
- The *structural accessor* category refers to operations that return information regarding the attributes of the object to which the operation belongs, without changing the *status* of the object.
- The *structural mutator* category corresponds to operations that change the *status* of the object to which the operation belongs.
- The *collaborational* category which helps define the communication between objects and how objects are controlled in the system.
- The *degenerate* category corresponds to operations which provide little information.

Based on these categories, we defined our taxonomy (see Table 4.1). Concretely, our taxonomy (1) discards the *collaborational* and *degenerate* categories, and (2) has been enriched with additional *stereotypes* aimed at identifying extra/further *operations*' semantics not considered in the original taxonomy (marked with an asterisk in Table 4.1). On the one hand, we have not taken into account the *collaborational* category since the collaboration between objects is already given by the SqDs. Likewise, the *degenerate* category has been discarded because it reflects aspects that cannot be faced without checking the source code. On the other hand, we have included the *search*, *add* and *remove stereotypes* for covering *operations* that manage collection attributes (such as search, addition or removal, respectively); the *process stereotype*, for *operations* computing and returning information based on the object's *status* as a whole, without focusing on specific attributes; and the *modify stereotype*, for *operations* that modify concrete attributes of an object.

Table 4.1 Extension of the taxonomy of UML *class*' *operations* given in [6], showing the categories considered in our proposal. An asterisk in a *stereotype* means that such a *stereotype* does not appear in the original taxonomy, but it has been added by our proposal.

Category	Stereotype name	Description
Creational	«create»	The operation creates an object.
	«destroy»	The operation destroys an object.
Structural Accessor	«get»	The operation returns values of concrete attributes of an object.
	«search» *	The operation returns elements belonging to a concrete collection attribute of an object.
	«process» *	The operation returns values that are computed based on the object's <i>status</i> as a whole.
	«predicate»	The operation returns boolean values that are computed based on concrete attributes of an object.
	«property»	The operation returns values (of any type) that are computed based on concrete attributes of an object.
	«void-accessor»	The operation returns values (of any type) that are computed based on concrete attributes of an object. These values are returned by means of parameters.
Structural Mutator	«command»	The operation changes the <i>status</i> of an object as a whole (the modified attributes are unknown or irrelevant). It does not return information.
	«non-void-command»	The operation changes the <i>status</i> of an object as a whole (the modified attributes are unknown or irrelevant). It does return information.
	«set»	The operation directly sets the information passed to the operation as values of concrete attributes of an object.
	«modify» *	The operation modifies concrete attributes of an object.
	«remove» *	The operation removes an element from a concrete collection attribute of an object.
	«add» *	The operation adds an element on a concrete collection attribute of an object.

It is worth noting that when we say “the object’s *status* as a whole” in the explanations given in Table 4.1 for the *process*, *command* and *non-void-command stereotypes*, we refer to the object’s characteristics at some point, without delving into the concrete values of the object’s attributes. For instance, *operations* with the *process stereotype* compute and return information based on the object’s characteristics at some point, without relying on concrete attributes for computing the returned information. Thus, we can say that the involved attributes are *unknown* or *irrelevant*.

Finally, we note that all the stereotypes whose definition involves one or various attributes have been defined with a property (or *tag definition*) with the name *attributes*. This property is used to keep the list of all the attributes involved in the behaviour of the stereotyped operation. For example, the stereotype `<<get>>` linked with an operation (e.g., `getName`) that returns the value of the `name` attribute, has the *attributes* property containing the string "name".

Based on the categories and *stereotypes* shown in Table 4.1, we have defined the set of patterns included in the chart of Figure 4.3. Below, we explain such a chart relying on the categories from the taxonomy.

Creational *CIP1* is applied when an operation creates a new object (`<<create>>`), whereas *CIP2* is focused on the destruction of an object (`<<destroy>>`).

Structural Accessor In case of an operation execution not changing the object’s *status*, we have focused on the returned information. More specifically, *CIP3* addresses operations that directly return values of concrete object’s attributes (`<<get>>` and `<<search>>`), and *CIP4* and *CIP5* focus on operations that return computed values. Concretely, *CIP4* considers that the returned values are computed relying on the object as a whole (`<<process>>`), without taking into account the concrete attributes involved in the computation. Conversely, *CIP5* identifies the concrete attributes used to compute such information (`<<predicate>>`, `<<property>>`, and `<<void-accessor>>`). Therefore, it gives a *finer-grained* provenance than *CIP4*. As we can see in Table 4.1, the stereotypes `<<predicate>>`, `<<property>>`, and `<<void-accessor>>` denote behaviours with specific nuances. Nevertheless, we have not included such nuances in the PROV template because we consider that such information is irrelevant from the provenance point of view and therefore, its inclusion would overburden the resulting PROV templates. In our opinion, the most relevant feature of these stereotypes is that they compute the returned values based on concrete object’s attributes without modifying the object’s *status*.

Structural Mutator In case of an operation execution changing the object’s *status*, we distinguish two levels of granularity. Firstly, *CIP6* is used when the operation modifies the object as a whole (`<<command>>` and `<<non-void-command>>`), in such a way there is no information

about the attributes involved in such a change. Secondly, as we are interested in a *finer-grained* provenance when the attributes are relevant, we have distinguished when the execution of an operation:

- (CIP7) directly sets the information passed to the operation as values of attributes of the object («set»);

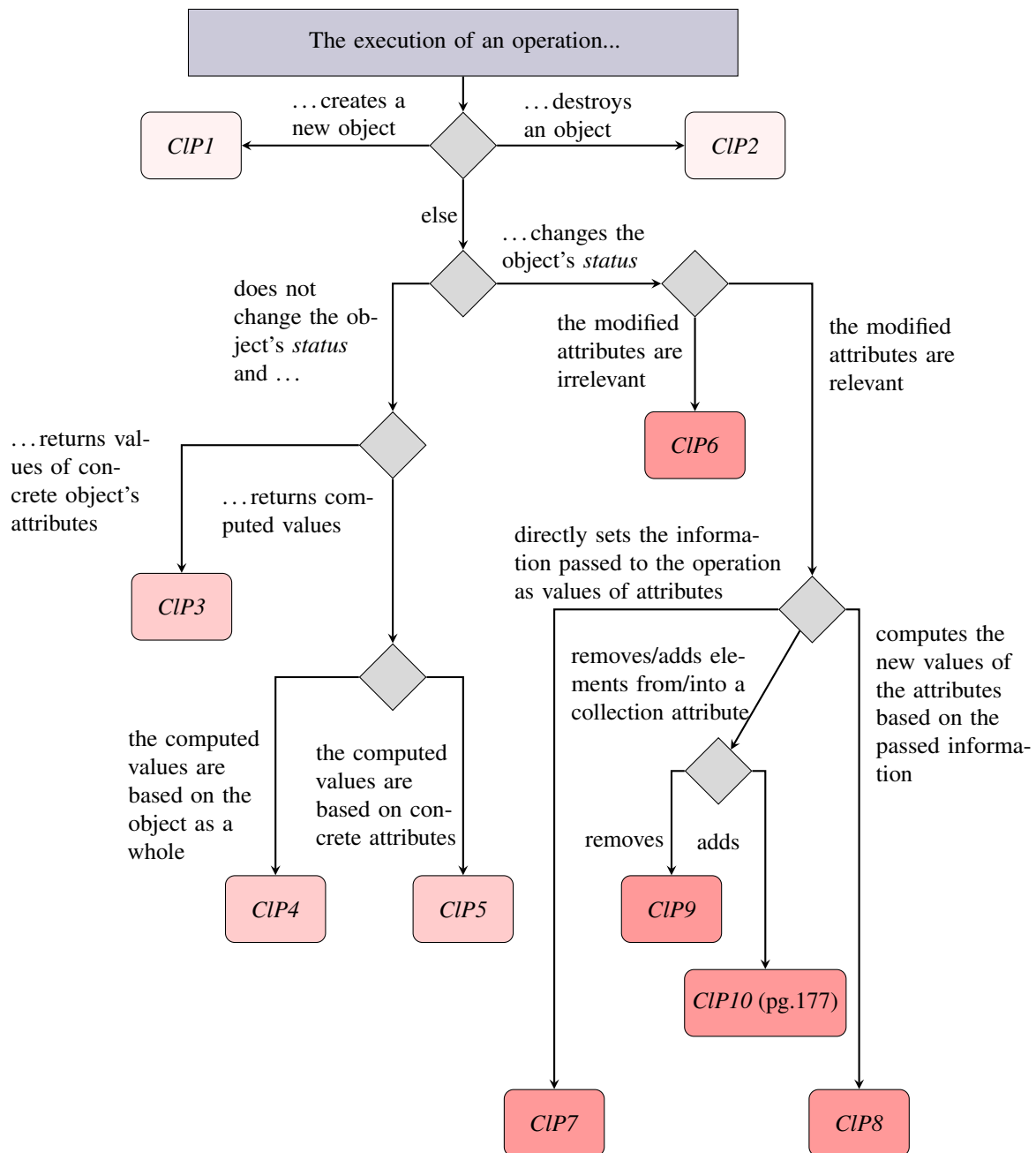


Fig. 4.3 This chart is focused on those patterns that rule the transformation of CDs

- (*CIP8*) computes the new values of the object's attributes based on the information passed to the operation (`<<modify>>`);
- removes (*CIP9*) or adds (*CIP10*) elements from/into a collection attribute of the object (`<<remove>>/<<add>>`).

UML State Machine Diagrams Patterns

State Machine diagrams specify the behaviour of individual objects of a system by means of the states they can go through during their lifetimes in response to events; hence, we focus on the information regarding the evolution of the objects' state caused by the execution of operations. The patterns addressing SMDs, which are included in Figure 4.4, address three different situations of an object, and concretely its state, caused by the execution of an operation. First, when an object is created (*StP1*); second, when the behaviour of an object is completed (*StP2*), and finally, when an object changes its state to another state (*StP3*).

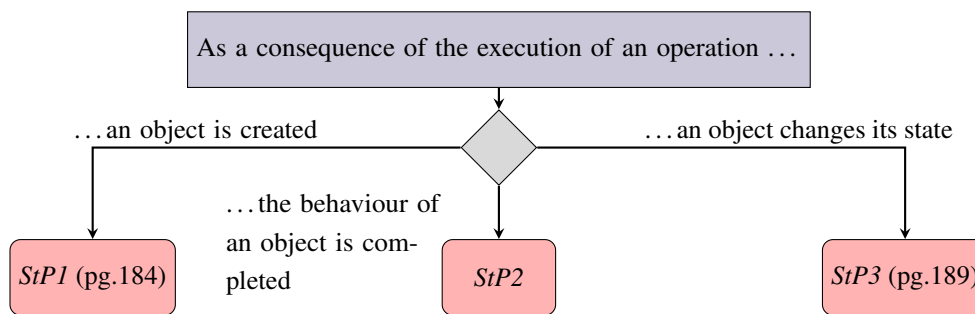


Fig. 4.4 This chart is focused on those patterns that rule the transformation of SMDs

4.3.5 Three patterns as an example

As an illustrative example, we have selected a situation from the University example (introduced in Chapter 2). This situation refers to the enrolment of students into seminars that, depending on the considered point of view, is modeled by a different type of UML diagram (see the SqD depicted in Figure 4.5, the CD in Figure 4.7, and the SMD in Figure 4.9). Below, we will explain how the patterns *SeqP2*, *CIP10* and *StP3* translate such diagrams into PROV. Here we note that the generic and complete description of these patterns is provided in Appendix B.

UML Sequence Diagrams

The enrolment of a student into a seminar could be modeled by the SqD depicted in Figure 4.5. It shows a context in which a sender participant (i.e., student), which is modeled

by the *lifeline* `st:Student` 1, interacts with a recipient participant (i.e., seminar) by calling the `enrolStudent` operation (specified by the *synchronous message* `enrolStudent` 2), and passing information (represented by the *input argument* `st` 3) into the operation execution. The call causes the recipient participant to execute the operation (modeled by the *ExecutionSpecification* 4), which results in a response (specified by the *reply message* `enrolStudent` 5) to the sender participant, containing output information (represented by the *output argument* `boolResult` 6).

The UML elements in the excerpt (those identified from 1 to 6) match the UML elements identified in *SeqP2*. This matching is illustrated in Table 4.2, where the first column shows the UML elements generically identified in *SeqP2*, and the second column presents the corresponding UML element in Figure 4.5. Hence, the transformation defined in *SeqP2* can be applied.

Table 4.2 Matching between the UML elements identified in *SeqP2* and the UML elements from Figure 4.5

UML element in <i>SeqP2</i>	UML element in Figure 4.5
<i>Lifeline</i>	<code>st:Student</code> 1
<i>Synchronous message</i>	(request) <code>enrolStudent</code> 2
<i>Input arguments</i>	<code>st</code> 3
<i>Execution specification</i>	<i>ExecutionSpecification</i> 4
<i>Reply message</i>	(response) <code>enrolStudent</code> 5
<i>Output arguments</i>	<code>booleanResult</code> 6

Following the transformation from UML to PROV defined by *SeqP2*, below we will explain the translation of the UML elements in Figure 4.5. Each step in the translation is accompanied by a replica of Figure 4.5 highlighting the UML element to be translated, together with the PROV element generated from it.

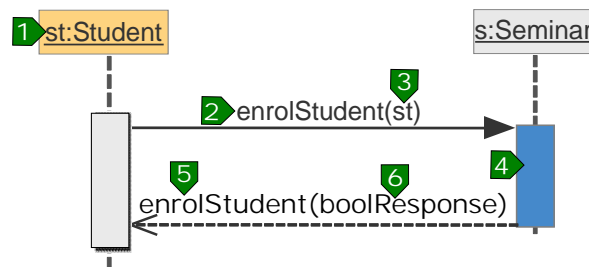
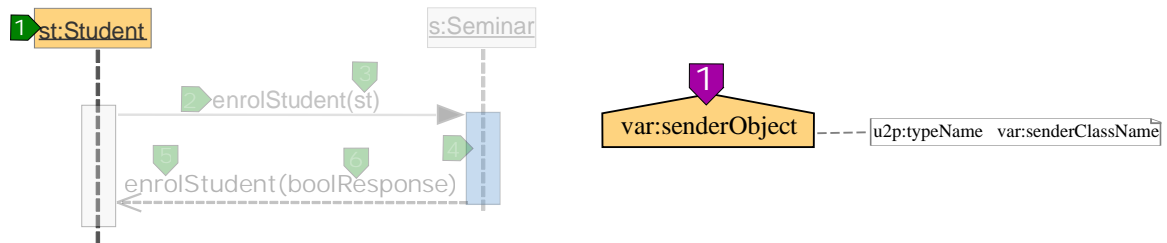
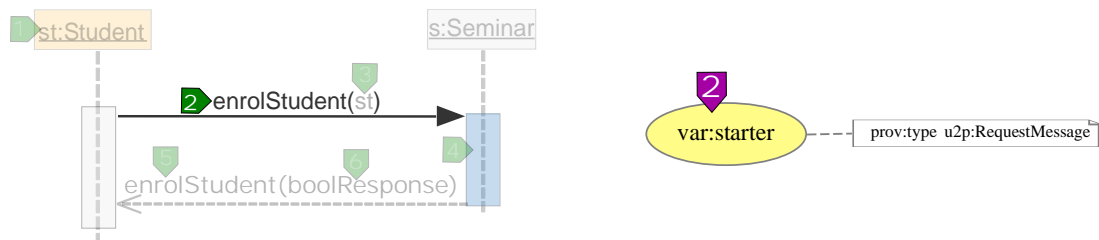


Fig. 4.5 An excerpt of the SqD depicted in Figure 2.2 showing the interaction between Student and Seminar when a student enrolls in a seminar

- The *lifeline* `st:Student` 1 models the Student object that calls the `enrolStudent` operation in a Seminar object. Since the Student object bears some form of responsibility for the execution of the `enrolStudent` operation, the *lifeline* `st:Student` 1 is mapped to a `prov:Agent` with the identifier `var:senderObject` 1. As with all the PROV elements in the template, this `prov:Agent` has a variable as identifier (`var:senderObject` in this case). When the template expands, this variable will be replaced by a unique qualified name such as `ex:Student3`. Additionally, this `prov:Agent` will contain the attribute `u2p:typeName` associated with `var:senderClassName` to denote the name of the class to which `var:senderObject` 1 belongs. In this case, when the resulting template expands, `var:senderClassName` will be replaced by the string "Student", since it is the name of the class to which the object represented by the *lifeline* `st:Student` 1 belongs.

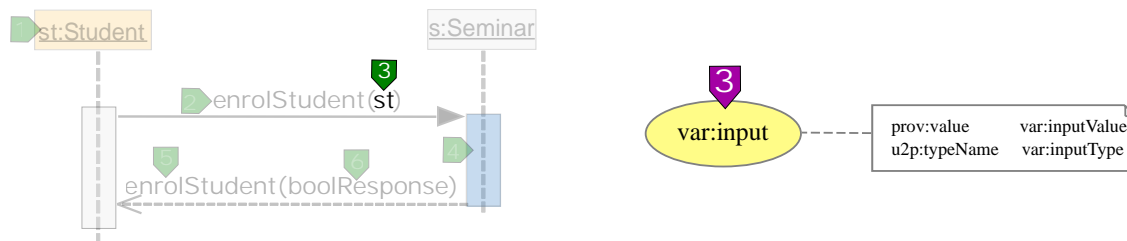


- The *synchronous message* `enrolStudent` 2 models the call that starts the execution of the `enrolStudent` operation. It is translated into a `prov:Entity` identified as `var:starter` 2. Furthermore, in order to show that `var:starter` 2 is a request message, `var:starter` 2 has the attribute `prov:type` with the value `u2p:RequestMessage`.

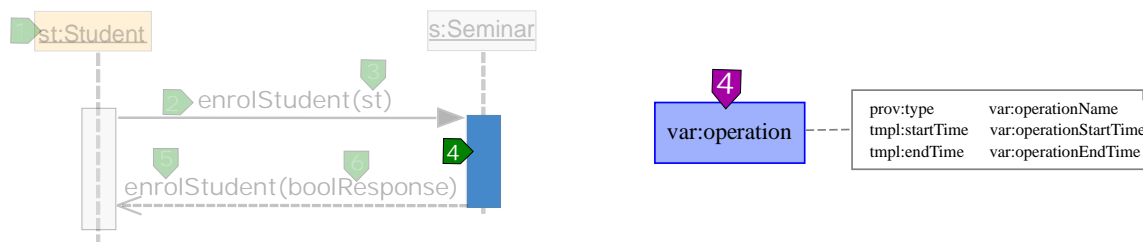


- The *input argument* `st` 3 specifies the Student object that is passed to the `enrolStudent` operation. It is mapped to a `prov:Entity` with the identifier `var:input` 3. This `prov:Entity` has the attributes `prov:value` and `u2p:typeName` associated with the variables `var:inputValue` and `var:inputType`, respectively. When PROV template expands, the former variable will be replaced by the direct representation of the *input argument* `st` 3 (e.g., "Carlos Sáenz"),

and the latter variable will be replaced by a string with the name of the class to which the *input argument* `st` **3** belongs (i.e., "Student").



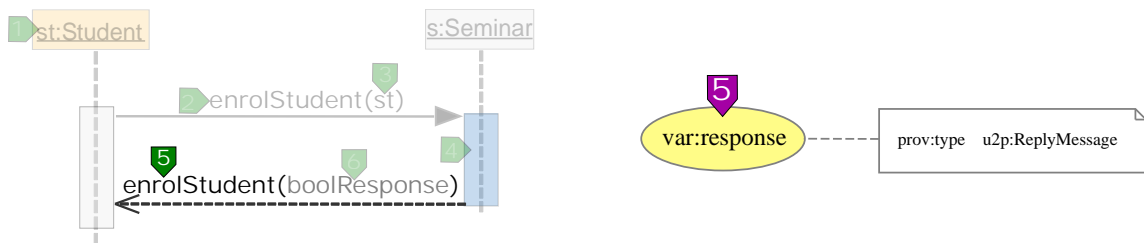
- The ExecutionSpecification **4** shows the period of time that a Seminar object devotes to the execution of the `enrolStudent` operation. It is translated into a `prov:Activity` identified as `var:operation` **4**. This `prov:Activity` has three attributes with information about the execution of the `enrolStudent` operation. The attribute `prov:type` is associated with the variable `var:operationName`, whose value will be the name of the executed operation (i.e., `ex:enrolStudent`). Likewise, the attributes `tmpl:startTime` and `tmpl:endTime` are linked with the variables called `var:operationStartTime` and `var:operationEndTime`, respectively. For example, when the template expands, the former will be replaced by an `xsd:dateTime` value for the start of the execution of the `enrolStudent` operation (e.g., 2019-07-17T14:21:12.085844), whereas the latter could be replaced by an `xsd:dateTime` for the end of the `enrolStudent` operation (e.g., 2019-07-17T14:22:12.085844).



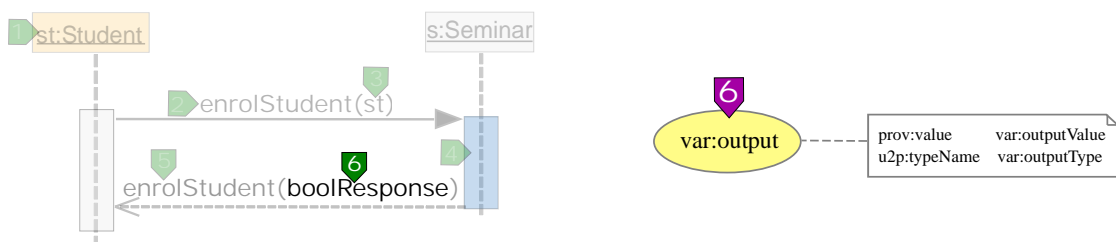
Note that the proposed translation into PROV takes into account the *lifeline* `st:Student` **1**; however, the *lifeline* `s:Seminar` is not addressed, even though it is the participant that executes the operation `enrolStudent`. This decision was based on other patterns' better ability to both (1) identify the participant responsible for executing that operation `enrolStudent`, and (2) give a more detailed information about the implications that the execution of `enrolStudent` has in the object that executes it. As we will see in the following sections, these patterns are: *CIP10*, which shows how the execution of `enrolStudent` affects the

status (i.e., values of the attributes) of `Seminar` objects; and the pattern *StP3*, which represents the changes in the `Seminar` objects' state caused by the execution of the operation `enrolStudent`. Both *StP3* and *CIP10* will be described below. Later, we will explain how the PROV templates resulting from these patterns are consistent with each other, and therefore, the PROV documents generated after their expansion can be merged.

- The *reply message* `enrolStudent` 5 models the response to the call that starts the execution of the `enrolStudent` operation. It is mapped to a `prov:Entity` with identifier `var:response` 5. Additionally, so as to note that `var:response` 5 is a reply message, `var:response` 5 has the attribute `prov:type` with the value `u2p:ReplyMessage`.



- The *output argument* `boolResult` 6 represents the `Boolean` object contained in the response of the `enrolStudent` operation. It is translated into a `prov:Entity` identified as `var:output` 6. Furthermore, it has two attributes, `prov:value` and `u2p:typeName` that are linked with the variables `var:outputValue` and `var:outputType`, respectively. When the template expands, `var:outputValue` will be replaced by the direct representation of `boolResult` 6 (i.e., `true` or `false`) and `var:outputType` by a string with the name of the class to which `boolResult` 6 belongs (i.e., `"Boolean"`).



In addition to defining the translation from concrete UML elements (identified from 1 to 6) to PROV elements (identified from 1 to 6), *SeqP2* also states how the generated PROV elements must be related between each other in order to generate a PROV template containing the full semantics of the translated UML SqD. Below, we explain the PROV

relations defined by *SeqP2* which, together with the aforementioned PROV elements, define the PROV template shown in Figure 4.6.

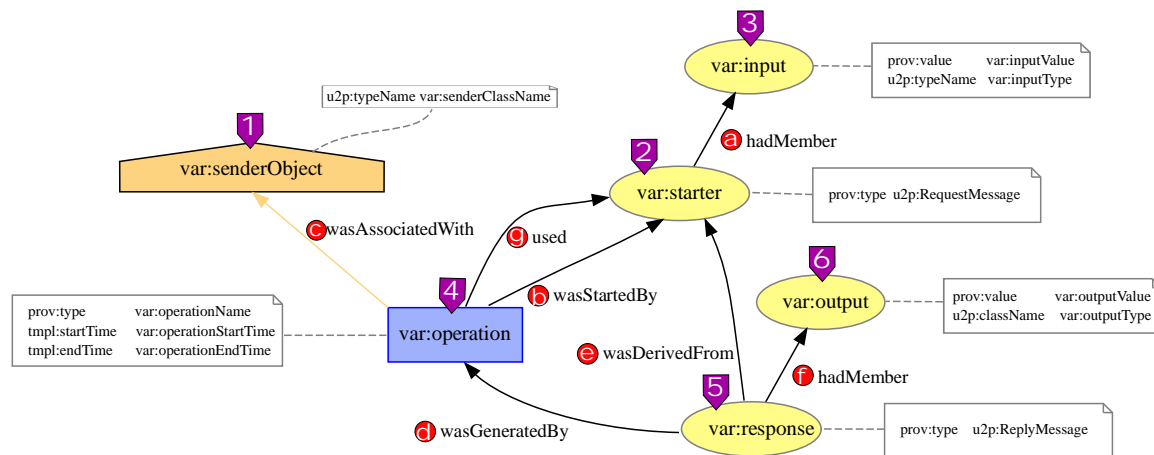


Fig. 4.6 PROV template obtained by applying *SeqP2* to the SqD depicted in Figure 4.5

- **(a)** `prov:hadMember`. Since the input argument `st` **(3)** (`var:input` **(3)**) is contained in the synchronous message `enrolStudent` **(2)** (`var:starter` **(2)**), the relation **(a)** `prov:hadMember` links `var:starter` **(2)** with `var:input` **(3)**.
- **(b)** `prov:wasStartedBy`. The ExecutionSpecification **(4)** (`var:operation` **(4)**), referring to the execution of the `enrolStudent` operation, is deemed to have been started by the *synchronous message* `enrolStudent` **(2)** (`var:starter` **(2)**). Thus, the relation **(b)** `prov:wasStartedBy` associates `var:operation` **(4)** with `var:starter` **(2)**.
- **(c)** `prov:wasAssociatedWith`. The *lifeline* `st:Student` **(1)** (`var:senderObject` **(1)**) is responsible for the execution of the `enrolStudent` operation. This is because the *lifeline* `st:Student` **(1)** (`var:senderObject` **(1)**) sends the *synchronous message* `enrolStudent` **(2)** (`var:starter` **(2)**) that starts the ExecutionSpecification **(4)** (`var:operation` **(4)**). This responsibility is included in the template using the relation `prov:wasAssociatedWith` from `var:senderObject` **(1)** to `var:operation` **(4)**.
- **(d)** `prov:wasGeneratedBy`. The *reply message* `enrolStudent` **(5)** (`var:response` **(5)**) is the response obtained from the execution of `enrolStudent` (which is represented in UML by the ExecutionSpecification **(4)** that is translated into `var:operation` **(4)**). This fact is shown by means of the relation **(d)** `prov:wasGeneratedBy` between `var:response` **(5)** and `var:operation` **(4)**.

- **e** `prov:wasDerivedFrom`. The *reply message* `enrolStudent` **5** (`var:response` **5**) is the response to the *synchronous message* `enrolStudent` **2** (`var:starter` **2**). To reflect this fact, the relation **e** `prov:wasDerivedFrom` associates `var:response` **5** with `var:starter` **2**.
- **f** `prov:hadMember`. Since the *output argument* `boolResult` **6** (`var:output` **6**) is contained in the *reply message* `enrolStudent` **5** (`var:response` **5**), the relation **f** `prov:hadMember` associates `var:response` **5** with `var:output` **6**.
- **g** `prov:used`. Due to the fact that the execution of the operation `enrolStudent` (specified by the *ExecutionSpecification* **4** that is translated into `var:operation` **4**) uses the *synchronous message* `enrolStudent` **2** (`var:starter` **2**) to perform its behaviour, the relation **g** `prov:used` is included for linking `var:operation` **4** with `var:starter` **2**.

Finally, we remark that, contrary to what may be expected, the PROV template generated by *SeqP2* does not reflect the *usage* of the *input argument* `st` **3** (`var:input` **3**) by the operation execution (modeled by the *ExecutionSpecification* **4** that is translated into `var:operation` **4**). This is because SqDs focus on the flow of information, not its usage. As we will see below, the pattern *CLP10* is better suited for this purpose, since it will include detailed information regarding the role of `var:input` **3** in the execution of the operation `enrolStudent`.

UML Class Diagrams

The situation of enrolling a student into a seminar involves the UML elements inside the CD depicted on the left-hand side of Figure 4.7. It shows a context in which the execution of the `enrolStudent` operation (modeled by the *operation* `enrolStudent` **2** with the stereotype `add`) on a `Seminar` object (specified by the *class* `Seminar` **1**) directly adds the information passed to the operation (the *input parameter* `st:Student` **3**) as new element(s) of a concrete object's

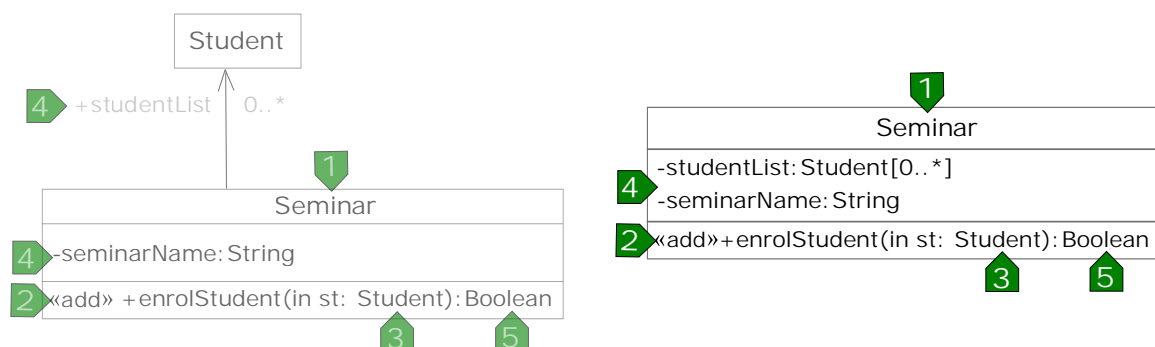


Fig. 4.7 On the left-hand side, there is an excerpt of the CD depicted in Figure 2.1.1 showing those elements that are involved in the enrolment of a student in a seminar. On the right-hand side, there is an equivalent representation of it.

collection attribute (specified by the *association* with the role `studentList` 4), thus provoking a change in the `Seminar` object's status. Additionally, the execution of the `enrolStudent` operation returns a boolean value (represented by the *output parameter* `Boolean` 5).

On the basis of the UML specification (page 206 in [1]), we note that the *association* with the role `studentList` 4 can be represented as an *attribute* inside the *class* `Seminar` 1 with the appropriate multiplicity, whereby the type of the *attribute* is the class of the partner objects (i.e., `Student`), and the name of the *attribute* is the role of the *association* (i.e., `studentList`). Aiming at facilitating the understanding of the following explanation, we will use this equivalent representation, which is depicted on the right-hand side of Figure 4.7.



The elements in the excerpt (identified from 1 to 5) match with those UML elements identified in *CIP10*, and consequently, the transformation defined in *CIP10* can be applied. The match between the elements identified in the pattern, and the elements in the excerpt is shown in Table 4.3. The first column depicts the UML elements from *CIP10*, and in the second column there are the matched UML elements from the right-hand side of Figure 4.7.

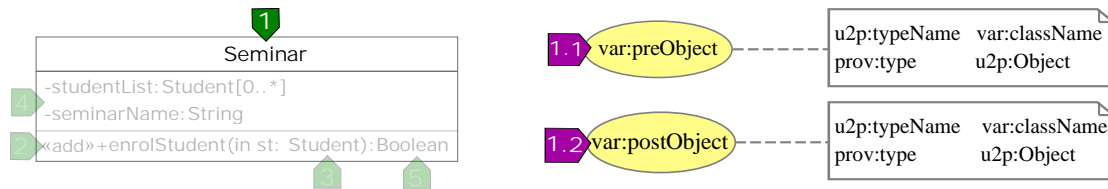
Table 4.3 Matching between the UML elements identified in *CIP10* and the highlighted UML elements from Figure 4.7




UML element in <i>CIP10</i>	UML element in Figure 4.7
<i>Class</i>	<code>Seminar</code> 1
<i>«add» Operation</i>	<code>enrolStudent</code> 2
<i>Input parameters</i>	<code>st:Student</code> 3
<i>Attributes</i>	<code>studentList</code> and <code>seminarName</code> 4
<i>Output parameters (optional)</i>	<code>Boolean</code> 5

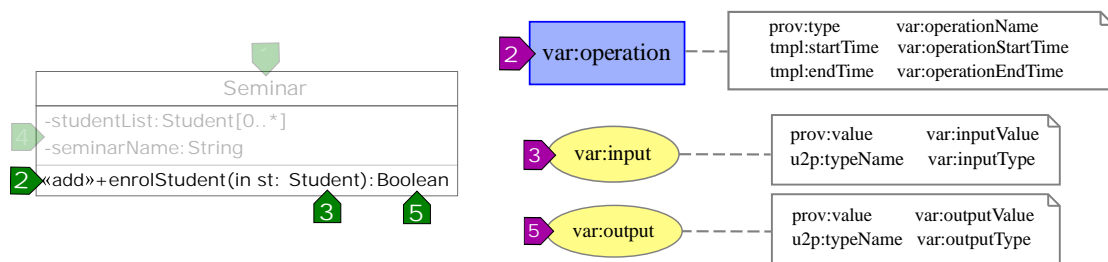
Guided by the transformation defined in *CIP10*, we will explain how the UML elements from the right-hand side of Figure 4.7 are translated into PROV. Each step in this transformation is illustrated by a copy of the right-hand side of Figure 4.7 highlighting the UML element(s) to be translated, together with the PROV element(s) generated from it(them).








- The *class* `Seminar` 1 classifies `Seminar` objects attending to their static characteristics and behaviour. Thus, it represents `Seminar` objects both before and after the execution of an operation. Since the execution of the `enrolStudent` operation changes the object's status, the *class* `Seminar` 1 is translated into two PROV elements: (1) the `prov:Entity var:preObject` 1.1, which is the `Seminar` object with the status (pre) before the execution; and (2) the `prov:Entity var:postObject` 1.2, which is the `Seminar` object with the status (post) after the execution. Additionally, both `var:preObject` 1.1 and `var:postObject` 1.2 have the attributes `u2p:typeName` and `prov:type`. On the one hand, `u2p:typeName` is associated with the variable `var:className`, which will be replaced by the string "Seminar" when the


template expands. On the other hand, `prov:type` has the value `u2p:Object` to denote that both `var:preObject`  and `var:postObject`  are objects.



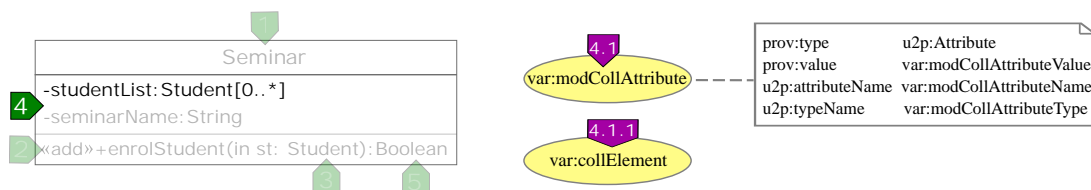
- The *operation* `enrolStudent`  with the stereotype `«add»` shows that the execution of the operation directly adds a new element into a concrete collection attribute (in this case, the *attribute* `studentList` ). It is translated into a `prov:Activity` identified as `var:operation` . Such a `prov:Activity` has three attributes: `prov:type`, `tpl:startTime`, and `tpl:endTime`. These attributes, when the template expands, will be replaced by the values such as `ex:enrolStudent`, `"2019-07-17T14:21:12.085844"` and `"2019-07-17T14:22:12.085844"`, respectively.



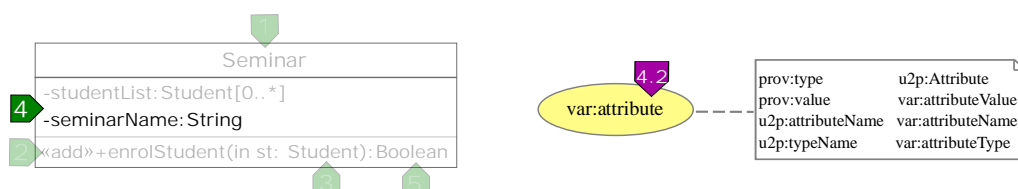
Additionally, the *operation* `enrolStudent`  shows two parameters: the *input parameter* `st:Student` , and the *output parameter* `Boolean` . On the one hand, the *input parameter* `st:Student`  is mapped to a `prov:Entity` with the identifier `var:input` , which has the attributes `prov:value` and `u2p:typeName` associated with the variables `var:inputValue` and `var:inputType`, respectively. On the other hand, the *output parameter* `Boolean`  is translated into a `prov:Entity` identified by `var:output` . This `prov:Entity` contains the attributes `prov:value` and `u2p:typeName` linked with the variables `var:outputValue` and `var:outputType`, respectively. When the template expands, the variables `var:inputValue` and `var:outputValue` will be replaced by the direct representation of their corresponding `prov:Entity` (e.g., `var:inputValue` by `"Carlos Sáenz"` and `var:outputValue` by `true`), whereas `var:inputType` and `var:outputType` will be replaced by the name of the type of the parameter (i.e., `var:inputType` by `"Student"` and `var:outputType` by `"Boolean"`).

- The *attributes* `studentList` and `seminarName`  represent the static characteristics of `Seminar` objects. They are translated as follows:

- `studentList` is the collection attribute that the operation modifies by adding a new element. It is translated into the `prov:Entity` identified by `var:modCollAttribute` 4.1. Additionally, each element in this collection preceding the operation execution is a separate `prov:Entity` identified by `var:collElement` 4.1.1. The `prov:Entity` with the identifier `var:modCollAttribute` 4.1 has four attributes: `prov:type`, `prov:value`, `u2p:attributeName`, and `u2p:typeName`. As for the attribute `prov:type`, it is related to the value `u2p:Attribute` to show that `var:modCollAttribute` 4.1 is an attribute. The attribute `prov:value` is associated with the variable `var:modCollAttributeValue` which will be replaced by the direct representation of `var:modCollAttribute` 4.1 when the template expands (e.g., the string "[Carlos Sáenz, Beatriz Pérez, Francisco García]"). Regarding the attribute `prov:attributeName`, it is linked with the variable `var:modCollAttributeName`. This variable will be replaced by the string value "studentList" when the template expands. Finally, the attribute `u2p:typeName` is related to the variable `var:modCollAttributeType`, which, when the template expands, will be replaced by a string such as "List", which denotes the name of the class to which the *attribute* `studentList` 4 belongs.



- `seminarName`, which is an attribute not modified by the execution of the operation, is mapped to a `prov:Entity` identified with `var:attribute` 4.2. As before, this `prov:Entity` has the attributes `prov:type`, `prov:value`, `u2p:attributeName`, and `u2p:typeName`, which are respectively associated with `u2p:Attribute`, `var:attributeValue`, `var:attributeName`, and `var:attributeType`. As previously, when the template expands, the variables `var:attributeValue`, `var:attributeName`, and `var:attributeType` will be replaced with values such as "Provenance_seminar", "seminarName", and "String".



The aforementioned generated PROV elements are not isolated, *CIP10* also defines how they must be related to obtaining a PROV template with the full semantics of the translated

UML CD. Next, we will explain the PROV relations stated in *CIP10*, which together with the previous PROV elements, define the resulting PROV template shown in Figure 4.8.

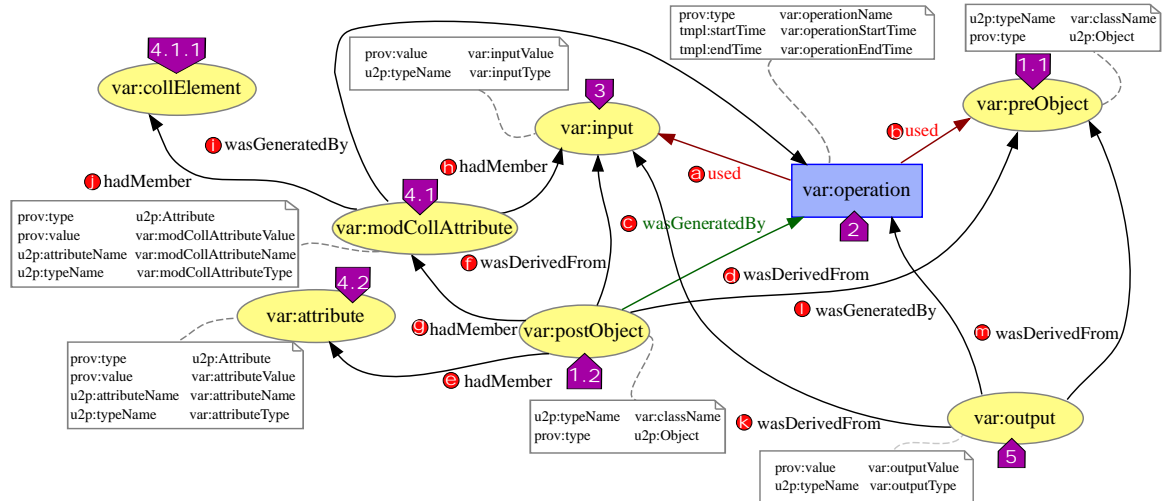


Fig. 4.8 PROV template obtained by applying *CIP10* to the CD highlighted in Figure 4.7

- The *operation* `enrolStudent` (2) (`var:operation` 2) uses the *input parameter* `st:Student` (3) (`var:input` 3) to perform its behaviour, and generates the *output parameter* `Boolean` (5) (`var:output` 5). These facts are shown in the PROV template through the relations (a) `prov:used` between `var:operation` 2 and `var:input` 3, and (1) `prov:wasGeneratedBy` between `var:output` 5 and `var:operation` 2. Additionally, according to the *use-generate-derive triangle* [167], we associate the *generated* `prov:Entity` `var:output` 5 with the *used* `prov:Entity` `var:input` 3 by means of the relation (k) `prov:wasDerivedFrom`.
- The *operation* `enrolStudent` (2) (`var:operation` 2) provokes the change in the object's status from a pre-operation (`var:preObject` 1.1) to a post-operation status (`var:postObject` 1.2). This fact is denoted by (b) `prov:used` between `var:operation` 2 and `var:preObject` 1.1, and by (c) `prov:wasGeneratedBy` associating `var:postObject` 1.2 with `var:operation` 2. In this case, we have applied the *use-generate-derive triangle* three times. First, associating `var:postObject` 1.2 with `var:preObject` 1.1 by the relation (d) `prov:wasDerivedFrom`. Second, linking `var:postObject` 1.2 with `var:input` 3 by the relation (f) `prov:wasDerivedFrom`. Third, associating `var:output` 5 and `var:preObject` 1.1 by (m) `prov:wasDerivedFrom`.
- The object of the *class* `Seminar` (3) with the status after the execution (`var:postObject` 1.2) has the *attributes* `seminarName` (4) (`var:attribute` 4.2) and `studentList` (4) (`var:modCollAttribute` 4.1). Thus, `var:postObject` 1.2 is associated with `var:attribute` 4.2 through the relation (e) `prov:hadMember`, and with `var:modCollAttribute` 4.1 using (g) `prov:hadMember`.

A question likely to come up is why `var:attribute` 4.2 is linked with `var:postObject` 1.2 (which represents the object with the status after the execution of the operation), but it is not associated with `var:preObject` 1.1 (the object with the status before the execution). We have taken this decision because the object of the *class* `Seminar` 1 that acts as a `var:preObject` here, was a `var:postObject` in a previous operation execution. Therefore, those attributes associated with this object in a `var:preObject` were registered when it previously played the role of `var:postObject`.

- This pattern states that the *operation* `enrolStudent` 2 *stereotyped* with `add` (`var:operation` 2) directly adds into the collection *attribute* `studentList` 4 (`var:modCollAttribute` 4.1) the *input parameter* `st:Student` 3 (`var:input` 3). We show this fact by using two relations: **i** `prov:wasGeneratedBy` for associating `var:modCollAttribute` 4.1 with `var:operation` 2; and **h** `prov:hadMember` for linking `var:modCollAttribute` 4.1 with `var:input` 3.
- The *operation* `enrolStudent` 2 adds a new element into the *attribute* `studentList` 4 (`var:modCollAttribute` 4.1), but such an *attribute* already had elements before the execution (`var:collElement` 4.1.1). Thus, **i** `prov:hadMember` associates `var:modCollAttribute` 4.1 with `var:collElement` 4.1.1.

UML State Machine Diagrams

Figure 4.9 shows an excerpt of a *state machine* (modeled by the `StateMachine` 2) that models the behaviour of objects of the `Seminar` class. Concretely, it depicts a context in which as a consequence of the execution of the `enrolStudent` operation (represented by the *event* `enrolStudent` 5), an *object* of the `Seminar` class 1 *changes* its state (specified by the source *state* `enroling` 3 and the target *state* `enroling` 4). Two remarks are necessary in this case. First, the objects of the `Seminar` class, whose behaviour is modeled by the `StateMachine` 2, lack a graphical representation in UML SMDs. Thus, we will refer to the object that changes

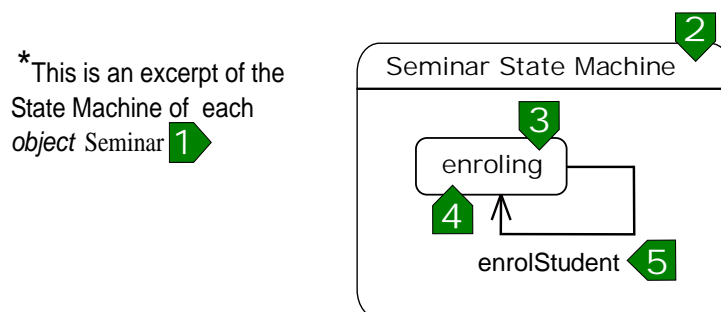


Fig. 4.9 An excerpt of the SMD depicted in Figure 2.4 showing how the enrolment of a student into a seminar affects the state of the objects of the `Seminar` class

its state as “*object* Seminar 1.” Second, so as to simplify the following explanation, Figure 4.9 lacks the performing *composite state*, which appears in the original State Machine diagram (Figure 2.4). Nevertheless, readers interested in how to include both *simple composite states* (e.g., the performing *state*) and *orthogonal composite states* are referred to the definition of *StP3*.

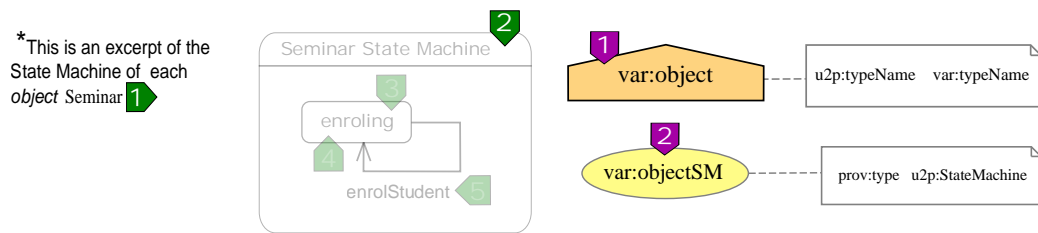
The *object* Seminar 1 as well as the UML elements in this excerpt (those identified from 2 to 5) match with the UML elements identified in *StP3*. We can see this matching in Table 4.4, in which the first column depicts the UML elements identified in *StP3*, and the second column shows the matched UML elements from Figure 4.9.

Table 4.4 Matching between the UML elements identified in *StP3* and the UML elements from Figure 4.9

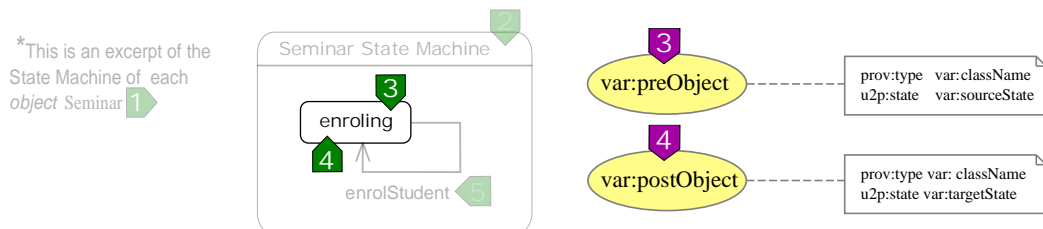
UML element in <i>StP3</i>	UML element in Figure 4.9
<i>Object</i>	Seminar 1
<i>State machine</i>	StateMachine 2
(source) <i>State</i>	(source) enrolling 3
(target) <i>State</i>	(target) enrolling 4
<i>Event</i>	enrolStudent 5

Based on the transformation from UML to PROV defined in *StP3*, we will explain how the UML elements from Figure 4.9 are mapped. Each step in this translation is attended by a copy of Figure 4.9 highlighting the UML element(s) to be translated, together with the PROV element(s) generated from it.

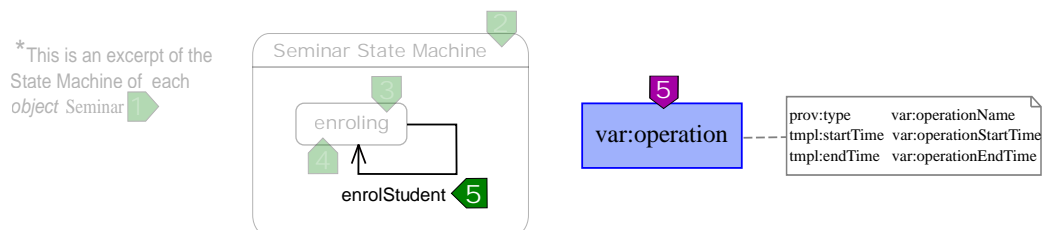
- The StateMachine 2 is used to express the set of states through which each *object* Seminar 1 goes during its lifetime in response to events; for instance, the execution of the enrolStudent operation could be an event. On the one hand, the StateMachine 2 is translated into a prov:Entity identified as var:objectSM 2. We use this prov:Entity to represent the abstraction of the states of the *object* Seminar 1, which will be specialized by each state of the *object* Seminar 1. Aiming at showing that var:objectSM 2 represents a state machine, it has the attribute prov:type with the value u2p:StateMachine. On the other hand, each *object* of the class Seminar 1 is mapped to a prov:Agent identified by var:object 1. We have decided to use a prov:Agent because later we want to show that the *object* Seminar 1 bears responsibility for its StateMachine 2 (var:objectSM 2). This prov:Agent has the attribute u2p:typeName associated with the variable var:typeName to show the name of the class to which var:object 1 belongs. In this case, when the template expands, var:typeName will be replaced by the string "Seminar".



- The *object* Seminar 1 changes from the (source) *state* enroling 3 to the (target) *state* enroling 4. The *state* enroling 3, which is the state before (pre) the operation, is translated into a `prov:Entity` identified by `var:preObject` 3, whereas the *state* enroling 4, which is the state after (post) the operation, is mapped to a `prov:Entity` with the identifier `var:postObject` 4. On the one hand, the `prov:Entity` `var:preObject` 3 contains the attributes `prov:type` and `u2p:state` associated with the variables `var:className` and `var:sourceState`, respectively. When template expands, the former variable will be replaced by the name of the class to which the *object* Seminar 1 belongs (i.e., `ex:Seminar`), and the latter variable will be replaced by the string with the name of the *state* enroling 3 (i.e., "enroling"). On the other hand, the `prov:Entity` `var:preObject` 3 also contains the attributes `prov:type` and `u2p:state` in order to include information about the class of the object, and the name of the target state.



- The *event* enrolStudent 5 specifies that the execution of the `enrolStudent` operation has taken place. It is translated into a `prov:Activity` with the identifier `var:operation` 5. As the previous patterns, this `prov:Activity` contains three attributes (`prov:type`, `tmpl:startTime` and `tmpl:endTime`) with information regarding the execution of the `enrolStudent` operation.



In addition to the definition of the translation from UML elements (identified from ➊ to ➋) to PROV elements (from ➌ to ➍), *StP3* also states relations between the aforementioned PROV elements. These relations together with the PROV elements define the resulting template depicted in Figure 4.10. Next, we will describe these relations.

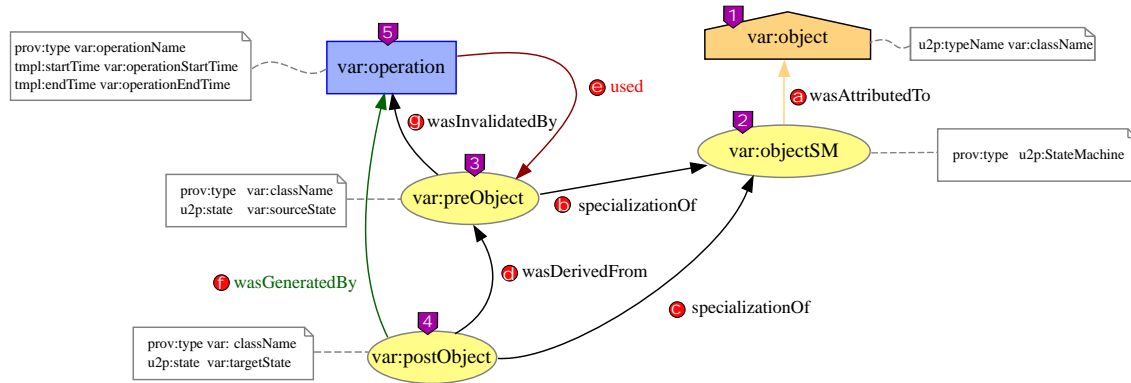


Fig. 4.10 PROV template obtained by applying *StP3* to the SMD depicted in Figure 4.9

- As we stated previously, the decision of using the `prov:Agent var:object` ➊ for representing each *object* Seminar ➊ comes with the need for showing its responsibility for its StateMachine ➋ (`var:objectSM` ➋). Concretely, we use ➋ `prov:wasAttributedTo` to link `var:objectSM` ➋ with `var:object` ➊ in order to show this fact in the PROV template.
- The `prov:Entity var:objectSM` ➋ obtained from the StateMachine ➋ represents the abstraction of the states of the *object* Seminar ➊. Thus, both the (source) *state* enrolling ➌ (`var:preObject` ➌) and the (target) *state* enrolling ➍ (`var:postObject` ➍) specialise the StateMachine ➋ (`var:objectSM` ➋). This is shown by two relations. First, the relation ➌ `prov:specializationOf` associates `var:preObject` ➌ with `var:objectSM` ➋. Second, ➍ `prov:specializationOf` links `var:postObject` ➍ with `var:objectSM` ➋.
- The *event* `enrolStudent` ➎ (`var:operation` ➎) pokes the change in the object's state from a pre-event state (`var:preObject` ➌) to a post-event state (`var:postObject` ➍). This fact is shown by the relation ➎ `prov:used` between `var:operation` ➎ and `var:preObject` ➌, and the relation ➏ `prov:wasGeneratedBy` linking `var:postObject` ➍ with `var:operation` ➎. Additionally, we have applied the *use-generate-derive triangle* resulting in the relation ➐ `prov:wasDerivedFrom` between `var:preObject` ➌ and `var:postObject` ➍.

Consistency among the resulting PROV templates

Throughout the aforementioned explanations, the reader may notice that there are common PROV elements in the resulting templates, e.g., the `prov:Activity` with the identifier

`var:operation` appears in all of them. This is because one of the principles for the definition of the patterns that we stated in Section 4.3.2 is *consistency*. Concretely, this principle states that, although some generated PROV templates design the provenance of the same situation from different perspectives, these PROV templates must have common elements which allow the merging of the expanded PROV documents. We note that we have decided to postpone the explanation of the consistency between templates until we have already explained the generation of bindings (in Section 4.5). This will allow us to illustrate the explanation of the consistency between patterns with the final provenance that is generated from such patterns. This will facilitate the reader to grasp how we have reached the *consistency* not only between patterns but also in the final provenance.

4.4 Towards the generation of *bindings*. BGM features and requirements

Having presented the conceptual approach of UML2PROV for obtaining PROV templates, with the design of the provenance to be generated from UML, in this section, we give our proposal for creating the bindings to be used for instantiating the aforementioned PROV templates, and thus generating the provenance ready for consumption. Concretely, our approach advocates providing each application with a Bindings Generation Module (BGM, see Figure 4.1) for automatically obtaining the bindings as the application is executed. However, since the generation of bindings is driven by UML designs of applications with different nature, to establish an only generic module is not a suitable option. Instead, we have decided to describe the features that, from our point of view, the BGMs should have in order to minimise the intrusion on software designers' and developers' *modus operandi*, as well as to facilitate the maintenance of the provenance-aware applications. To define these features, we will leverage the analysis of the approaches for capturing data that we performed in our systematic review, concretely in the *data capture dimension* in Section 3.3.4. Based on these features, we have defined a set of requirements that the BGMs for an application must satisfy, independently of the UML designs of such an application and the technologies used for its development.

As we stated in Section 3.3.4, the *data capture dimension* refers to the way in which provenance data can be captured, distinguishing four categories: *tracing*, *level*, *mechanism*, and *technique*.

- As far as the *tracing* category is concerned, it refers to the proposal taken for computing provenance information. In our context, the provenance information is computed when the

PROV templates are instantiated with bindings. The BGM should provide developers with a mechanism to select when to compute provenance: when required (*lazy*) or immediately (*eager*). Regardless of when and how bindings are generated, one strategy could be to delay the computation of provenance (i.e., the expansion of templates obtaining the final PROV documents) until the application finishes (*lazy* approach). This strategy could be possible by storing the sets of bindings, or each binding independently. Later, these bindings may be used to instantiate the PROV templates on demand. Alternatively, another strategy could be to expand the templates as the application is being executed (*eager* approach). This strategy does not need to store the bindings since it directly stores the final PROV documents.

Feature 1 The BGM must be agnostic about when to compute provenance.

- Regarding the *level* of capture, it refers to the level (i.e., point in the software stack) where the distinct forms of provenance can be gathered. As we mentioned previously in Section 4.3.2, our approach advocates designing provenance for operations executions since the three supported types of UML diagrams have elements for representing aspects of operations executions. Taking this into account, the BGM must generate bindings based on provenance data obtained from the operations executions of a process (i.e., at *process-level*).

Feature 2 The BGM must capture provenance data from operations executions.

- The *mechanism* used to capture provenance data could rely upon *internal structures* or *external services*. The BGM is meant to be integrated into the existing application, relying on *internal structures* for generating the bindings. Whilst to use *external services* could have been a valid alternative, we have decided not to deal with distributed and heterogeneous systems. Nevertheless, we remark that this could be a further line of work.

Feature 3 The BGM must rely upon *internal structures* for generating bindings.

- Finally, the *technique* category refers to the techniques used by existing provenance systems to capture provenance. In Section 3.3.4 we identified two approaches: *annotation* and *inversion*. Whereas the *annotation* method refers to the process of adding to or “making up” existing data, the *inversion* approach uses the property by means of which some derivations can be inverted to find the input data used to derive the output data. Concretely, an annotation is a *name-value* pair [92], in which the `value` may be `text`, `int` values, and so on. Thus, bindings can be seen as a kind of annotation. These bindings must be generated automatically.

Feature 4 The BGM must automatically annotate the existing data when generating bindings.

Once we have enumerated the desirable features that, to the best of our knowledge, any BGM should have, we have stated a set of requirements that the BGM created for an application must follow so as to satisfy such features.

1. To manually adapt the applications' source code to generate bindings could be a valid option for satisfying the aforementioned features. In that way, the *developer* is free to select any of the approaches in the above categories: e.g. she/he could choose when to compute provenance (Feature 1) as well as to capture provenance from operations executions (Feature 2). Additionally, she/he could use *internal structures* for the generation of bindings (Feature 3), and implement mechanisms for automatically *annotating* the existing data items (Feature 4). However, to manually adapt the applications' source code is a tedious, time-consuming and error-prone task. Concretely, *software developers* would have to transverse the whole applications' source code and add suitable instructions to generate the bindings. Thus, we have identified the following requirement:

Requirement 1 (R1) The instrumentation of the application to add the instructions for generating bindings must be carried out automatically.

2. To rely upon *internal structures* (Feature 3) for generating bindings would result in provenance capture code spread out over the application's code (scattered code), which makes the maintenance of the application a cumbersome task. This is because the bindings must be generated in concrete points distributed over the application's code. Another disadvantage is that if software designers modify the design of the application by including those elements that model the generation of bindings, the resulting diagrams become complex and difficult to manage. Otherwise, if designers do not perform such changes in the design, there will be an inconsistency between the application's design and the application's source code. In order to instrument applications without affecting their design and maintenance, we have stated the following two requirements regarding the *internal structures* to be used for generating bindings:

Requirement 2 (R2) The instructions for bindings generation must be located apart from the application's source code, in an independent module, avoiding the generation of repetitive and obfuscated code.

Requirement 3 (R3) The BGM has to be able to identify the specific moments within the application's source code where it is required to execute such instructions.

3. The BGM should be agnostic about when to compute provenance (Feature 1). That is, *software developers* must be able to follow a *lazy* approach by computing the provenance (that is, to expand the PROV templates) when required, or alternatively, to follow an *eager* approach by computing the provenance as the application is running. Taking this into account, we state that:

Requirement 4 (R4) The BGM must provide the *software developer* with mechanisms for selecting the configuration that best suits her/his needs, allowing the *developer* to decide when to compute the provenance.

4. The generated bindings will be used to instantiate the PROV templates obtained from the UML design in order to obtain the provenance to be exploited. Thus, these bindings must be consistent with such PROV templates aiming at obtaining provenance that describes what the application actually does. This consistency involves two features. First, since the generated templates contain the design of the provenance to be generated from operations executions, the BGM must collect provenance data from operations executions (Feature 2). Second, the bindings must be generated automatically (Feature 4). Each PROV element in a template has a variable that unequivocally identifies the element at hand; thus, the values of such variables must be included in the bindings. Taking these issues into account, we have identified two requirements:

Requirement 5 (R5) Each binding obtained from an application's execution must be associated with at least one PROV template automatically generated from the UML diagrams.

Requirement 6 (R6) The variables included in a set of bindings must correspond with the variables in their associated PROV templates.

With all these requirements, we have defined a conceptual background over which the BGMs should be implemented in order not only to generate bindings accordingly with the designed provenance, but also to automatically generate bindings without affecting the maintenance of the application. Concretely, a possible list of bindings for the PROV templates depicted in Figures 4.6, 4.8, and 4.10 is depicted in Figure 4.11. In addition to the list of bindings (in the right-hand side), on the left of each binding it is the patterns that generated a template containing the name of the variable in the binding. This set of bindings satisfies requirements *R5* and *R6* since each binding is associated with at least one of the PROV templates obtained from *SeqP2*, *CIPI0* and *StP3*, and the variables correspond with the variables in the PROV templates. To optimise space, the serialisation of these bindings was inspired from the JSON-LD serialisation [30]: "var" key contains the bindings represented

```

SeqP2      {"var":{
SeqP2      "senderObject":      [ { "@id": "ex:Student3" } ],
SeqP2      "senderClassName":   [ { "@type": "xsd:string", "@value": "Student" } ],
SeqP2      "starter":          [ { "@id": "ex:Message_1" } ],
SeqP2 CIP10 "input":           [ { "@id": "ex:Student3_1" } ],
SeqP2 CIP10 "inputValue":      [ { "@type": "xsd:string", "@value": "Carlos Sáenz" } ],
SeqP2 CIP10 "inputType":       [ { "@type": "xsd:string", "@value": "Student" } ],
SeqP2 CIP10 StP3 "operation":      [ { "@id": "ex:enrolStudent1" } ],
SeqP2 CIP10 StP3 "operationName": [ { "@id": "ex:enrolStudent" } ],
SeqP2 CIP10 StP3 "operationStartTime": [ { "@type": "xsd:dateTime", "@value": "2019-07-17T14:21:12.085844" } ],
SeqP2 CIP10 StP3 "operationEndTime": [ { "@type": "xsd:dateTime", "@value": "2019-07-17T14:22:12.085844" } ],
SeqP2      "response":         [ { "@id": "ex:Message_2" } ],
SeqP2 CIP10 "output":         [ { "@id": "ex:Boolean_1_1" } ],
SeqP2 CIP10 "outputValue":     [ { "@type": "xsd:boolean", "@value": "true" } ],
SeqP2 CIP10 "outputType":      [ { "@type": "xsd:string", "@value": "Boolean" } ],
SeqP2      "object":          [ { "@id": "ex:Seminar1" } ],
CIP10 StP3 "className":       [ { "@type": "xsd:string", "@value": "Seminar" } ],
CIP10 StP3 "objectSM":        [ { "@id": "ex:Seminar1_0" } ],
CIP10 StP3 "preObject":       [ { "@id": "ex:Seminar1_5" } ],
CIP10 StP3 "sourceState":     [ { "@type": "xsd:string", "@value": "enrolling" } ],
CIP10 StP3 "postObject":      [ { "@id": "ex:Seminar1_6" } ],
CIP10 StP3 "targetState":    [ { "@type": "xsd:string", "@value": "enrolling" } ],
CIP10      "modCollAttribute": [ { "@id": "ex:List_5" } ],
CIP10      "modCollAttributeValue": [ { "@type": "xsd:string", "@value": "[Carlos Sáenz, Beatriz Pérez, Francisco García]" } ],
CIP10      "modCollAttributeName": [ { "@type": "xsd:string", "@value": "studentList" } ],
CIP10      "modCollAttributeType": [ { "@type": "xsd:string", "@value": "List" } ],
CIP10      "attribute":       [ { "@id": "ex:String1_1" } ],
CIP10      "attributeValue":  [ { "@type": "xsd:string", "@value": "Provenance_seminar" } ],
CIP10      "attributeName":   [ { "@type": "xsd:string", "@value": "seminarName" } ],
CIP10      "attributeType":   [ { "@type": "xsd:string", "@value": "String" } ],
CIP10      "collElement":    [ { "@id": "ex:Student1_1" },
                               { "@id": "ex:Student2_1" } ]
    },
    "context":{
      "xsd": "http://www.w3.org/2001/XMLSchema#",
      "u2p": "http://uml2prov.unirioja.es/ns/u2p#",
      "ex": "http://example.com"
    }
  }}

```

Fig. 4.11 Possible set of bindings obtained from the execution of the `enrolStudent` operation. On the left-hand side, those patterns that generated a template including the variable next to it.

as a JSON dictionary; `@id` is the identifier of the current value; `@type` is the data type of the current value; `@value` is the string serialisation of the value; and `"context"` key provides the information to interpret the prefixes used in values.

4.5 The consistency between templates and bindings for obtaining consistent provenance

Throughout Section 4.3.5, we have seen how different types of UML diagrams, which model the enrolment of a student into a seminar, are translated into PROV templates. The consequence of using different types of UML diagrams is the generation of PROV templates with different information, covering different perspectives of the same situation. As we noted in Section 4.3.2, one of the stated principles for the definition of those patterns is the *consistency*, which refers to the semantic overlap that the PROV templates resulting from different types of UML diagrams should have. To see at a glance the overlap between the aforementioned generated templates, we have included the Figure 4.12. This figure contains

colored circles that enclose the variables that identify the PROV elements in the templates obtained by applying *SeqP2* (template in Figure 4.6), *CIP10* (template in Figure 4.8), and *StP3* (template in Figure 4.10). Later, we will base on this figure for explaining the consistency of the generated provenance.

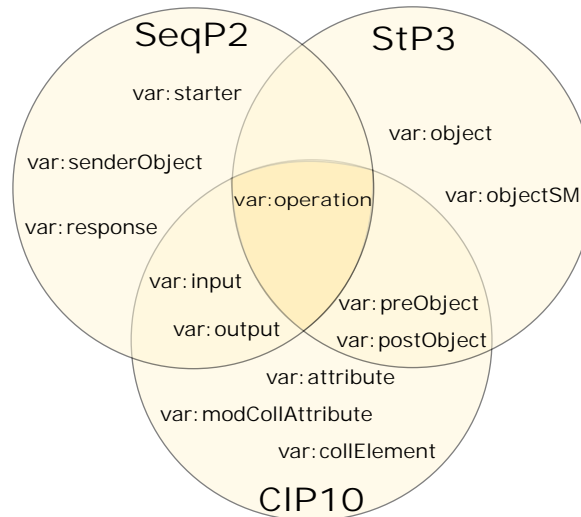


Fig. 4.12 Three circles enclosing the variables that identify the elements in the PROV templates generated by applying *SeqP2*, *CIP10*, and *StP3*

Similarly, to ensure the generation of consistent PROV documents after expansion, in Section 4.4 we stated the requirements *R5* and *R6* for ensuring the consistency between bindings and templates. For instance, the set of bindings in Figure 4.11 fulfils these requirements. Concretely, the expansion of the templates shown in Figures 4.6, 4.8 and 4.10 with the set of bindings in Figure 4.11 generates three final PROV documents with the provenance information ready to be exploited. Each PROV document (depicted in Figure 4.13) contains provenance information about the perspective of the template from which it comes. More specifically, the provenance information in Figure 4.13(a) is focused on the flow of information, since its source template comes from a Sequence diagram. Figure 4.13(b) shows provenance information related to a change in an object's *status* because its template was obtained from a Class diagram. Finally, the provenance information in Figure 4.13(c) is related to a change in an object's state due to the used template resulting from a State Machine diagram.

The generated PROV templates design the provenance for the execution of `enrolStudent` operations from three viewpoints. This execution is shown in all the templates by means of the `prov:Activity` with the identifier `var:operation` (see Figure 4.12). This `prov:Activity` has been obtained from three different UML elements appearing in the three types of UML

diagrams that are addressed. Concretely, it has been generated from an *execution specification* shown in a Sequence diagram, an *operation* from a Class diagram, and an *event* depicted in a State Machine diagram. When an `enrolStudent` operation is executed, a concrete binding is generated for linking a unique identifier value with `var:operation`. As we can see in the set of bindings from Figure 4.11, `var:operation` is linked with the value `ex:enrolStudent1`. As a consequence of this, when the templates expand, we obtain three final PROV documents with the same `prov:Activity` (see Figure 4.13), but with different elements and relations that define the specific perspective of the provenance in such a document.

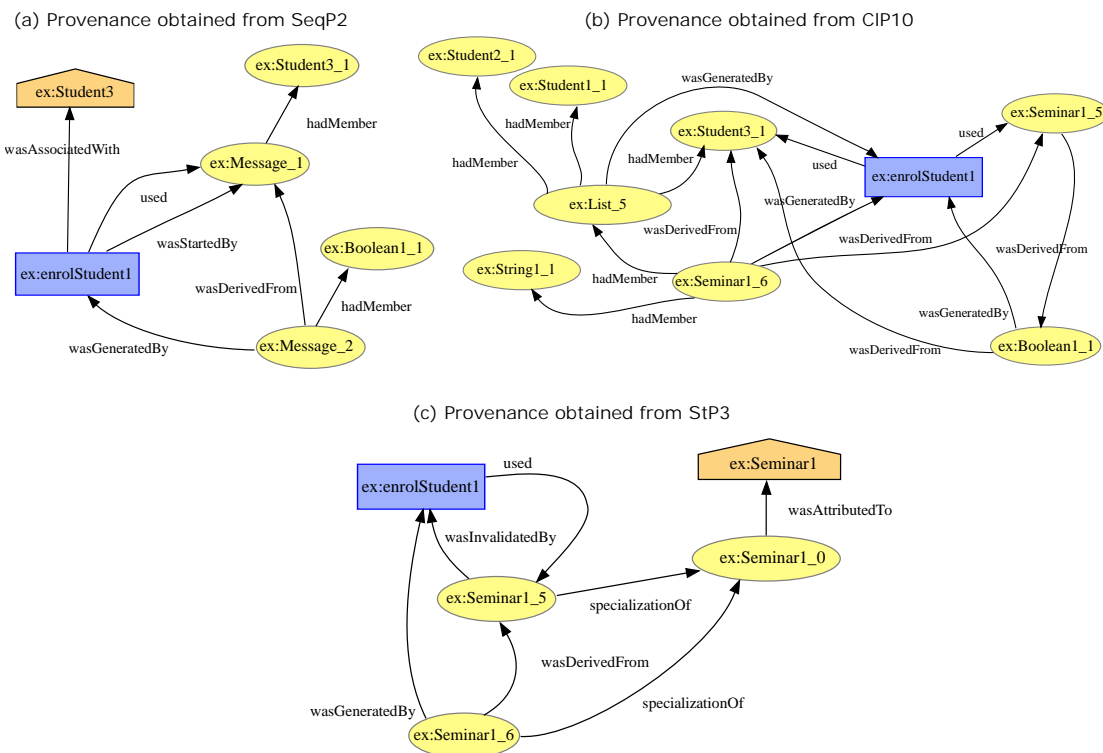


Fig. 4.13 PROV documents obtained by expanding the PROV templates shown in Figures 4.6, 4.8 and 4.10 with the set of bindings in Figure 4.11. We do not show the PROV attributes to avoid overburdening the figure.

Another remarkable aspect is the different perspectives of *StP3* and *CIP10* to reflect the change in an object of the *Seminar* class, which is triggered as a consequence of the execution of the `enrolStudent` operation (`var:operation`). Both patterns generate the `prov:Entities` `var:preObject` and `var:postObject` (see Figure 4.12) to show this change in the object, but from two different perspectives. On the one hand, in *CIP10*, `var:preObject` and `var:postObject` are the object with the *status* (i.e., values of its attributes) before and after the execution of the operation, respectively. Hence, this perspective gives a finer-grained provenance about the change in the object by showing how the values of its attributes change.

On the other hand, in *StP3* these variables represent the source and target *state* of the object before and after the execution of `enrolStudent`, respectively. Thus, this pattern will reflect how a `Seminar` object changes its *state* as a consequence of the execution of `enrolStudent`. Finally, we note that, despite the fact that `var:preObject` and `var:postObject` appear in two different templates, both templates are consistent. This is because, when the operation `enrolStudent` is executed, it is created one binding associating `var:preObject` with a unique value, and another binding associating `var:postObject` with another unique value. For instance, the set of bindings from Figure 4.11 shows that `var:preObject` is linked with the value `ex:Seminar1_5`, and `var:postObject` with the value `ex:Seminar1_6`. As a consequence of this, the final PROV documents (see Figures 4.13(b)(c)) have the same `prov:Entities` with the identifiers `ex:Seminar1_5` and `ex:Seminar1_6`, but they are related to different elements that show the change of *status* (in Figure 4.13(b)), and the change of state (in Figure 4.13(c)).

Similarly, both *SeqP2* and *CIP10* translate the input and output data into two different `prov:Entities` identified by `var:input` and `var:output` (see Figure 4.12). The pattern *SeqP2* translates the input and output data that is modeled using UML *input* and *output arguments* of UML *messages*, whereas *CIP10* translates the input and output data specified by UML *input* and *output parameters* of UML *operations*. When the `enrolStudent` operation is executed, a concrete value is associated with `var:input` and `var:output`. Concretely, the set of bindings in

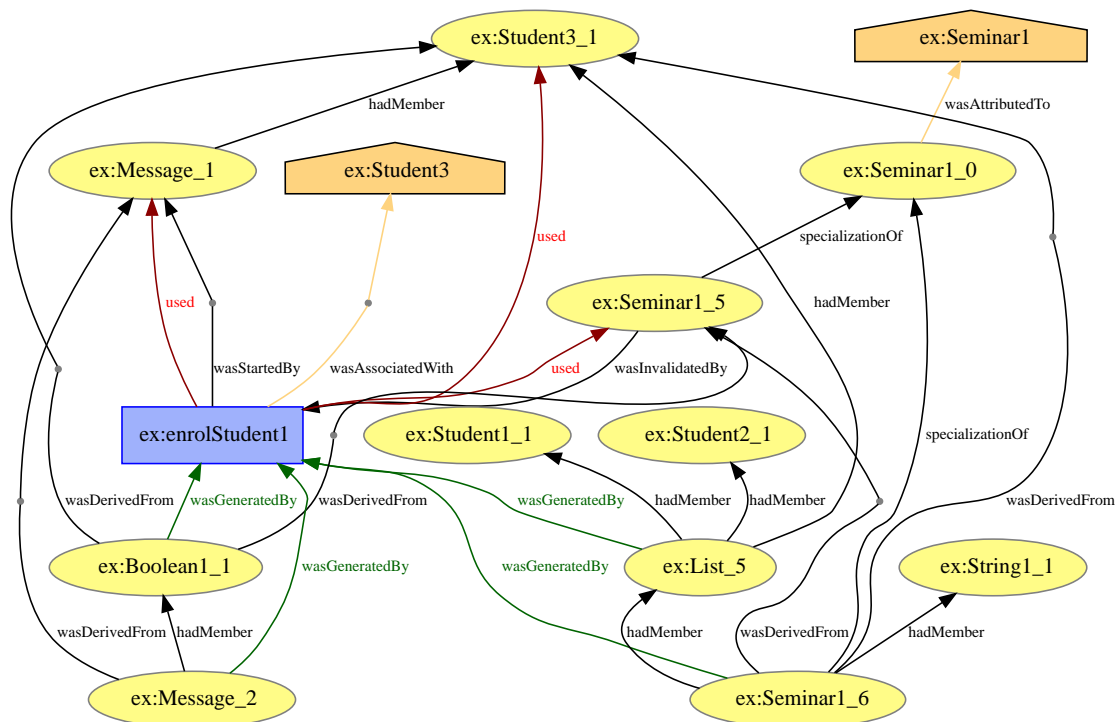


Fig. 4.14 PROV document obtained by merging the documents shown in Figure 4.13

Figure 4.11 shows that `var:input` is associated with `ex:Student3_1` and `var:output` is linked with `ex:Boolean1_1`. These associations lead to the generation of two different final PROV documents (in Figures 4.13(a)(b)) with the same `prov:Entities` identified by `ex:Student3_1` and `ex:Boolean1_1`. However, as before, the remaining relations and elements in that PROV documents define the perspective of the provenance in such documents. Whereas the provenance in Figure 4.13(a) shows the flow of the input and output data because its design is given by a UML Sequence diagram, the provenance in Figure 4.13(b) gives information about the usage and the generation of such data.

The remainder variables are specific of a concrete template and therefore, when the templates expand, they will be replaced only in the template that contains them.

Finally, we would like to note that although the expansion of each PROV template resulted in a PROV document with provenance regarding a concrete perspective (Figures 4.13(a)(b)(c)), these final PROV documents can be merged in one PROV document containing the three perspectives together. Figure 4.14 shows the result of merging the three PROV documents depicted in Figure 4.13.

4.6 Conclusions

This chapter introduced the conceptual definition of UML2PROV, a framework for making applications provenance-aware. Unlike other approaches, UML2PROV addresses the design and the capture of provenance from the design of the application. To do this, UML2PROV encompasses two main elements (PROV templates and BGM) that are obtained from the UML diagrams of an application. While PROV templates contain the design of the provenance to be generated, BGM is the module deployed into the application to generate bindings. Once we have the templates and the bindings, the template expander takes as input both of them and generates the final PROV documents with the provenance ready to be exploited.

On the one hand, we defined the generation of PROV templates by an extensive set of 17 transformation patterns, which were stated following a set of principles that facilitates their understandability and maintenance. As far as these patterns are concerned, we also provided readers with a set of charts that aims to (1) help users find the pattern that best suits their interests and (2) see at a glance information about all the patterns together, allowing users to compare them easily. Additionally, in order to illustrate the application of these patterns, we showed how the provenance of a situation from the University example, which is modeled in UML, can be designed using PROV templates generated by UML2PROV.

On the other hand, UML2PROV advocates providing each addressed application with a BGM obtained from the application's UML design, so as to minimise the intrusion on

software designers' and developers' modus operandi, as well as to facilitate the maintenance of the provenance-aware applications. In order to define a conceptual background over which the BGMs will be implemented, we stated a set of requirements that all the generated BGMs must fulfil. Finally, we shown how the consistency between the templates, which are produced by our transformation patterns, and the bindings, which are generated by BGMs fulfilling our set of requirements, results in consistent provenance.

We note that the transformation patterns together with the BGM's requirements define a conceptual background over which an implementation of UML2PROV should rely on. In the following chapter, we will introduce our reference implementation of UML2PROV.

Chapter 5

Implementation of UML2PROV

This chapter aims at providing a *reference implementation* of UML2PROV, that, following the conceptual considerations presented in Chapter 4, automatically generates PROV templates and a BGM from the UML design of an application. Concretely, the process followed towards the development of such a reference implementation has consisted of two main stages. First, we implemented a *proof of concept* that addressed part of the conceptual definition of UML2PROV. More specifically, this proof of concept was developed in order to verify the feasibility of such a conceptual definition, as well as to validate if a potential implementation of it was possible. Once we validated both aspects, we developed our reference implementation, which provides a complete development of the conceptual definition of UML2PROV and therefore, it automates the full process for making an application provenance-aware.

The poof of concept of UML2PROV has been presented in:

- Carlos Sáenz-Adán, Beatriz Pérez, Trung Dong Huynh, Luc Moreau: *UML2PROV: Automating Provenance Capture in Software Engineering*. In the International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM) 2018. [164]
- Carlos Sáenz-Adán, Luc Moreau, Beatriz Pérez, Simon Miles, Francisco J. García-Izquierdo: *Automating Provenance Capture in Software Engineering with UML2PROV*. In the International Provenance and Annotation Workshop (IPAW) 2018. [165]

The reference implementation, which implements the full conceptual definition of UML2PROV, has been submitted for publication in:

- Carlos Sáenz-Adán, Beatriz Pérez, Francisco J. García-Izquierdo, Luc Moreau: *Integrating Provenance Capture and UML with UML2PROV: Principles and Experience*. Submitted for publication in IEEE Transactions on Software Engineering.

5.1 Introduction

Our conceptual definition of UML2PROV establishes the background and considerations to create, starting from UML diagrams, the two main elements of the UML2PROV architecture: the PROV templates and the BGM. The disparate nature of applications, with their own nuances and complexity, and thus the UML design of such applications, prevent us from performing a manual generation of the two main elements of the UML2PROV architecture, since it would constitute a time-consuming and error-prone task. Thus, the automatization of this process is desirable. Since the starting point of UML2PROV is a set of diagram models of a system, the usage of MDD¹ tools to implement UML2PROV is natural.

Relying upon MDD, we started developing a proof of concept [164, 165] in order to verify that the conceptual definition of UML2PROV was feasible, as well as to validate if a potential implementation of it was possible. Concretely, this first approach focused on providing a pragmatic solution that did not necessarily fully implement UML2PROV. Thus, we based on technologies we were familiar with, instead of learning new techniques. Whilst this proof of concept allowed us to verify the feasibility of the conceptual definition of UML2PROV and showed us that an implementation of it was possible, we realised that a more complete and accurate implementation of UML2PROV could be provided (later, we will explain the reasons).

Subsequently, we based on the experience acquired during that prior implementation to develop a more complete solution that would constitute the *reference implementation* of our proposal. This reference implementation strictly implements the full conceptual definition of UML2PROV (see Chapter 4) for the automatic generation of (1) the templates and (2) the BGM for an application.

It is remarkable that our reference implementation for generating templates could be seen as a generic solution suitable for being used by any final user of UML2PROV, who has the UML design of its application. Such a reference implementation provides a complete automatic translation from any UML design to PROV templates as established by our patterns. However, unlike the generation of templates, our reference implementation for generating the BGM of an application is limited to applications developed in Java. Nevertheless, we have defined a generic structure for the BGM aimed at allowing potential developers to implement BGMs in other programming language. An example of this fact is the work presented by Johnson et al. in [170], where they modeled their system by means of UML Sequence diagrams, and later, they used our reference implementation of UML2PROV to generate *templates*. Since their system is developed in a programming language other than Java, they

¹Background information about MDD and tools is provided in Section 2.2.1

did not generate the BGM by using our reference implementation, but they implemented their own mechanisms to generate *bindings* during the execution of the system.

That the BGM generated by our reference implementation satisfactorily addresses two of the open problems uncovered in Section 3.4: *computational overhead* and *querying*. On the one hand, we mentioned that the *computational overhead* is intrinsically related to the capture of provenance, and therefore, it is desirable to provide users with mechanisms to find a balance between both the capture of provenance and the computational overheads attributable to it. On the other hand, *querying* the generated provenance was also identified as an open problem in the provenance field. Concretely, when a provenance consumer is interested in *directly* retrieving provenance information, she/he must know the specific query language of the persistence layer of the used provenance system. This requirement usually forces provenance consumers to learn new query languages, which is an undesirable task. Thus, the BGM has been implemented in such a way that not only it satisfies all the requirements stated for BGMs, but it also allows developers to follow different approaches (called *configurations*) to manage bindings, keeping in mind both *computational overhead* and *querying* aspects. Concretely, although we will explain three different configurations, which are considered by us as the most relevant for reasons that we will explain later, we will give developers guidelines to implement their own configurations.

5.2 A Proof of concept for implementing UML2PROV


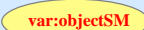


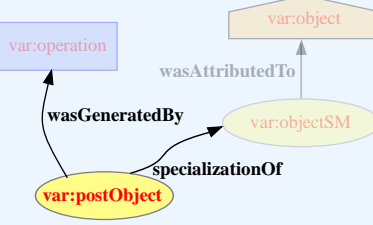
In this preliminary stage, we considered only two types of UML diagrams (Sequence and State Machine Diagrams) used to design an application. The purpose of this implementation was to validate if both the conceptual definition of the transformation patterns and the BGM were feasible. Thus, we prioritised our experience with XSLT [34] and the *Proxy-pattern* [171] over other available solutions that were unknown for us. Concretely, we used (1) XSLT to automate the transformation patterns and the generation of the BGM, and (2) the *Proxy-pattern* approach to implement the *BGM*. Next, we will explain the implementation of this proof of concept together with some of the issues raised by this solution.

5.2.1 Automatization of the transformation patterns: an XSLT-based approach

As advanced previously, following an MDD approach, we first implemented a preliminary version of the SMD and SqD patterns described in Section 4.3 by using XSLT [34]. Besides our previous experience with other developments, another reason for choosing XSLT as the

technology to be used for our first implementation of UML2PROV was that UML diagrams models are serialised in XMI, and XSLT was specifically developed for transforming XML-based documents into another text format. Taking this into account, the use of XSLT seemed natural. Concretely, following this approach, the UML diagram models, encoded in XMI, are automatically translated into PROV templates files, serialised in PROV-N (a text notation

Table 5.1 An excerpt of the XSLT template focusing mainly on the implementation of the *StP1* pattern. For each fragment in the excerpt (“XSLT source code” column), the description of the transformation (“Description” column) as well as the graphical notation of the template model (last column).

XSLT source code	Description	Graphical representation of the PROV template file
<pre><xsl:template match= "/uml:Model/packagedElement/region/transition"></pre>	<p>It states that the template is applied to all the UML Transitions in the source model.</p>	
	<p>This block creates those elements shared by all the patterns addressing State Machine patterns</p>	
<pre>document [...]</pre>	<p>It generates the document</p>	
<pre>agent(var:object, ...)</pre>	<p>It creates an agent with identifier <code>var:object</code></p>	
<pre>entity(var:objectSM, ...)</pre>	<p>It creates an entity identified by <code>var:objectSM</code></p>	
<pre>activity(var:operation, ...)</pre>	<p>It creates an activity with the identifier <code>var:operation</code></p>	
	<p>Next, it is the code that generates the concrete elements of each pattern, in particular, StP1.</p>	
<pre><xsl:when test=" (//subvertex[@xmi:type='uml:Pseudostate' and @xmi:id=\$idSource] and //subvertex[@xmi:type='uml:State' and @xmi:id=\$idTarget]) and not(//subvertex[@xmi:type='uml:State' and @xmi:id=\$idTarget]/region) and not(//subvertex[@xmi:type='uml:Pseudostate' and @xmi:id=\$idTarget]/region)"></pre>	<p>This excerpt checks if the transition corresponds with UML elements identified in StP1. That is, if the tackled transition has an initial pseudostate as source, and a state as target.</p>	
<pre>wasAttributedTo(var:objectSM, var:object, ...)</pre>	<p>This excerpt is responsible for linking <code>var:objectSM</code> with <code>var:object</code> by means of the PROV relation <code>wasAttributedTo</code>.</p>	
<pre>entity(var:postObject, ...) wasGeneratedBy(var:postObject, var:operation, ...) wasAttributedTo(var:postObject, var:objectSM, ...) [...]</pre>	<p>First, it generates an entity identified by <code>var:postObject</code>. Subsequently, it creates two PROV relations. (1) The PROV relation <code>wasGeneratedBy</code> between <code>var:postObject</code> and <code>var:operation</code>. (2) the PROV relation <code>specializationOf</code> between <code>var:postObject</code> and <code>var:objectSM</code>.</p>	
<pre>[...] </xsl:when> </xsl:template></pre>	<p>Below it is the code implementing the remaining SMD patterns.</p>	

format). As a way of illustrating how the transformation patterns were implemented, Table 5.1 depicts an excerpt of the XSLT template focusing mainly on the implementation of the pattern *StPI*². Such a pattern deals with UML *transitions* that lead objects to their first states; that is, a *transition* that goes from an *initial pseudostate* to a simple *state*. In particular, this table shows, for each fragment of the XSLT template (in the first column), a description of the transformation (second column) as well as the graphical notation of the PROV template file being generated by such a fragment (third column). Additionally, these fragments are organised around horizontal blocks of three colours with different intensity. The first block shows the XSLT code that defines which UML element is addressed by the XSLT template. In this case, as it is an XSLT template addressing UML State Machine diagrams, the XSLT template considers UML transitions due to the fact that transitions are triggered by events, which can reflect occurrences of operation executions (from which we want provenance data). The second block contains fragments of code that create PROV elements shared by all the State Machine diagram patterns. The third block shows the code responsible for generating the specific PROV elements of *StPI*. More specifically, at the beginning of this block, we can see the excerpt (inside a frame) in charge of checking whether *StPI* has to be applied or not; to do this, we have used XPATH expressions in order to check inside the XMI model if the source of the *transition* is an *initial pseudostate*, and if the target is a simple *state*.

Although this XSLT-based implementation was functional, during its development we came across several hurdles. More specifically, we found that the implementation of the patterns in XSLT implies to have a great knowledge of XMI specification, because when using XPATH in XSLT, we must take into account the deep structure of models that depends on metamodels which are themselves highly dependent on XMI [172]. Another disadvantage is that, in addition to the fact that the resulting XSLT code is not very understandable, it highly depends on the XMI structure [172], which leads to a high cost of maintenance as the UML diagrams' complexity increases. This is a crucial drawback since our proposal is also intended to support Class diagrams in addition to State Machine and Sequence diagrams, which would make us to define new XSLT templates taking into account the concrete XMI structural aspects concerning also Class diagrams. For instance, in Table 5.1, the excerpt located inside the frame serves to give an idea regarding the deep knowledge about the XMI structure that it is required. This excerpt aims at checking whether *StPI* has to be applied or not; the implementation of this condition in XSLT requires a great knowledge about not only how the involved UML elements are serialised in XMI, but also how we could access these elements by using XPATH.

²The complete definition of this pattern can be seen in Appendix B

Finally, we remark that this first attempt for implementing our patterns followed an M2T approach. That is, it directly translated UML diagrams into PROV template files in PROV-N format. The fact of implementing a direct transformation makes this approach highly dependent on the format of the PROV templates files. Thus, if we want to serialise the PROV templates in other format, it is required to modify the whole XSLT code to meet such a requirement. Considering this, we believe that a chain of M2M and M2T transformations is more suitable, since it contributes to the distinction between the transformation of UML diagrams into PROV templates, and the subsequent serialisation of these PROV templates.

In conclusion, although this proof of concept implementation for our patterns was a feasible solution at the beginning, we realised that using XSLT for defining our mapping rules was no longer the best suitable option. Specially, taking into account the existing mapping and transformation languages created by the MDD community that have better properties in terms of maintenance, reusability, and support to software development processes [173]. Later, in the reference implementation, we will see how alternative solutions to XSLT have allowed us to address the implementation of the transformation patterns in a more suitable way.

5.2.2 Automatization of the generation of the BGM: a *Proxy-pattern* and XSLT-based approach

Next, we explain our proof of concept implementation for automatically generating the BGM of an application, taking as input the UML design of such an application. We will start by explaining the implementation of the BGM we proposed as part of this first attempt and which relied upon the *Proxy-pattern* approach. Later, we will describe the XSLT template that we defined to automatically generate BGMs according to such an implementation.

An implementation of the BGM: a *Proxy-pattern* approach

Given our knowledge about the *Proxy-pattern* [171], we first decided to implement a proof of concept of the BGM based on such a pattern. The *Proxy-pattern* [171] is mainly intended to manage the access to objects' operations, allowing us to modify their behaviour dynamically. Briefly speaking, the *Proxy-pattern* provides a surrogate (called *proxy*) for another object (called *real subject*) to control its behaviour. In our case, we leverage the *Proxy-pattern* in order to extend the behaviour of the object's operations with extra functionality to generate *bindings*. Figure 5.1 depicts a UML Class diagram showing the structure of the *Proxy-pattern*. Next, we explain the four elements of this pattern together with the way we have used them for our implementation of the BGM:

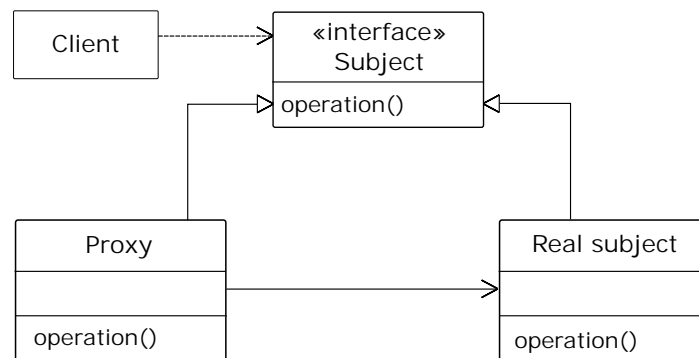


Fig. 5.1 Proxy-pattern architecture

1. The *Real subject* corresponds to the object whose behaviour we want to modify. It must implement the *Subject* interface. Concretely, we are interested in modifying its behaviour in order to obtain *bindings* during the execution of the object's operations while an application is running.
2. The *Subject* interface specifies all the operations implemented by the *Real Subject*.
3. The *Proxy* element also implements the *Subject* interface so that it can be used in any location where the *Real Subject* can be used. The *Proxy* element further maintains a reference to the *Real subject* and may execute code before and after the execution of the *Real subject's* operations. In our case, this element contains all the instructions to generate *bindings*.
4. The *Client* element is in charge of invoking the *Subject*, which allows the *Client* to interact with the *Proxy* as though it was the *Real Subject*. Thus, the *Proxy* constitutes the intermediary between the *Client* and the *Real Subject*. Each time an operation of the *Real Subject* is called, the *Proxy* intercepts the operation invocation, gathers *bindings*, and delegates the execution of the operation to the *Real Subject*.

Whilst to leverage the *Proxy-pattern* to generate *bindings* is a valid option, this solution requires to manually modify the application's source code (later, we will see how). Thus, this solution does not meet the BGM requirements stated in Section 4.4.

Automatization of the implementation of the BGM: an XSLT-based approach

To automatically obtain a BGM from the UML design for an application, we originally followed an XSLT-based proposal which took the *UML diagram models*, encoded in XMI, and generated the Java implementation of the BGM based on the *Proxy-pattern*. Concretely, the defined XSLT template generates a BGM encompassing two main elements: a *set of*

Subject interfaces or *interfaces* for short (one per each class in the SqD and SMD diagrams) and a Java class called *ProxyProvGenerator* (which would correspond to the *Proxy* element in the *Proxy-pattern* architecture).

Interfaces It is important to note that in the *Proxy-pattern* the *Real subject* must implement a *Subject* interface, and this interface should not necessarily appear in the original design of the application. Aiming at facilitating the creation of the *Subject* interface per each *Real subject* by making it in an automatic way, the defined XSLT transformation takes each UML class involved in the *UML* diagrams (SqD and SMD) and returns a Java interface containing all the class's operations. These interfaces will allow us to wrap a *Real subject* with the *Proxy* and consequently, to execute the additional behaviour for generating bindings. However, to do this, developers have to manually add a clause in the definition of each *Real subject*'s class in order to state that such a class implements the generated interface. To easily identify the interface of the *Real subject*, we have followed a naming convention of these interfaces so that each generated interface is named with the character "I" preceding the name of the *Real subject* class (e.g., the *IStudent* interface is generated for the *Student* class).

Finally, we note that these interfaces are considered *context-dependent components* (see the UML2PROV architecture in Figure 4.1) since they directly depend on the classes involved in the UML Sequence and State Machine diagrams.

ProxyProvGenerator Each *Real subject* needs a concrete *Proxy* that wraps it. In order to automatically generate these *Proxies* for concrete *Real subjects*, the XSLT transformation directly generates an only Java class called *ProxyProvGenerator*.

This class defines a static method called `generateProxy`, which receives a *Real subject* implementing its corresponding *Subject* interface and then, the method returns a specific *Proxy* for such a *Real subject*, which also implements the *Subject* interface. This *Proxy* is created with all the bindings generation instructions within. When a developer wants to capture provenance from the execution of the operations of a concrete object (i.e., a *Real subject*), she/he has to transverse the source code of the application identifying those objects of interest, and then to insert a call to the `generateProxy` operation with the identified object as input.

ProxyProvGenerator class uses the `Proxy` [174] class together with the interface named `InvocationHandler` [175] provided by the `java.lang.reflect` package. Using these Java elements, we are able to create a dynamic proxy object. More specifically, the `Proxy` class creates the dynamic proxy element itself, whereas the interface `InvocationHandler` invokes the methods of the dynamic proxy element.

without provenance capture	<code>Student student = new Student();</code>
with provenance capture	<code>IStudent student = ProxyProvGenerator.generateProxy(new Student());</code>

Fig. 5.2 Excerpts of Java code depicting the construction of an object as usual (without provenance capture), and the construction of an object by following the *Proxy-pattern* approach (with provenance capture)

In Figure 5.2, we give an example of construction of an object of the class `Student` (from the University example) *without/with provenance capture*. When the capture of provenance is required, the variable `student` must be typed as `IStudent`. Additionally, the operation `generateProxy` of the `ProxyProvGenerator` class has to be invoked, sending to it the instance of the class `Student` to be wrapped in the generated *Proxy*, which also implements the interface `IStudent`. This *Proxy* contains the behaviour to collect bindings regarding the invocation of `Student` objects' operations. The new variable `student` can be used as usual, but its behaviour has been modified to capture bindings.

Finally, we note that the `ProxyProvGenerator` is considered a *context-independent component* since it is agnostic about the given *Real subject*.

As it can be inferred from the above explanation, using *the Proxy-pattern* requires a manual instrumentation of the application's code. More specifically, this instrumentation task consists of (1) adding an *implements* clause in each *Real subject* class denoting that such a class implements a *Subject* interface, and (2) using the `ProxyProvGenerator` class to construct a concrete *Proxy* for each instance of a *Real subject* object. This latter task is especially remarkable for being time-consuming, since users have to transverse the source code of the application identifying those *Real subjects* of interest, and instantiating the `generateProxy` operation of the `ProxyProvGenerator` class to obtain the *Proxy* which generates bindings.

All in all, with this proof of concept implementation, we can conclude that, although considering only UML Sequence and State Machine diagrams, the conceptual definition of UML2PROV stated in Chapter 4 was feasible and, what is more, its implementation was possible. However, the technologies used in this implementation incurred in some drawbacks that led us to search for other alternatives. First, XSLT is not the best option to implement transformations from UML diagram models since it implies a deep understanding of the XMI specification. Additionally, the resulting XSLT code is not very understandable, which hinders its maintenance and scalability. Second, although the *Proxy-pattern* is a suitable solution for generating bindings during the execution, it still requires a manual instrumentation of the code in addition to including provenance-related instructions within the code. Thus, it does not meet the Requirement 1 for the BGM (defined in Section 4.4)

which states that the instrumentation must be carried out automatically. Similarly, it also does not fulfil Requirements 2 and 3, which state that the instructions for binding generation must be located apart from the application's source code (Requirement 2), and that the BGM must be able to identify those moments within the source code where it is required to execute the instructions (Requirement 3).

5.3 Reference implementation

As opposed to the previous implementation, which only supports UML Sequence and State Machine diagrams, this implementation supports all types of the UML diagrams considered in this work.

In order to avoid the disadvantages arose in the proof of concept concerning the automatic generation of both the templates and the BGMs, we decided to look for a MDD alternative solution. Concretely, we have chosen ATL and XPand as languages to implement the transformation patterns, which have better properties than XSLT in terms of maintenance, reusability and support to software development processes [173]. As for an implementation of BGMs which fulfils the BGM requirements as stated in Section 4.4, in this second approach we have advocated an *event-based* proposal developed on top of the Aspect Oriented Programming paradigm (AOP) [40]³ for generating bindings without user intervention. As a result, we have achieved a reference implementation which provides a complete implementation of the conceptual definition of UML2PROV given in Chapter 4.

5.3.1 Automatization of the transformation patterns: an ATL and XPand-based approach

Our first implementation approach directly generated PROV templates from the UML design by means of XSLT transformations; thus, the transformation from UML to PROV templates is closely related to the concrete PROV format chosen to serialise such PROV template files (PROV-N in that case). Now, our reference implementation automatically generates the *PROV template files* relying upon a refinement MDD-based process (see Figure 5.3) which, instead of directly generating the templates from the UML design, includes an intermediate step. We have decided to define an intermediate step in the transformation process in order to draw a distinction between the transformation from UML to PROV templates, and the concrete serialisation format of the final PROV template files. In this way, if a potential user is interested in a different serialisation format, it is not necessary to modify the transformation

³Background information about AOP is provided in Section 2.2.2

from UML diagrams to PROV templates. As we can see in Figure 5.3, our reference implementation follows an MDD-based tool chain that comprises two transformations: the first one corresponds to an M2M transformation (T_1), and has been implemented in ATL [33], and the second one is an M2T transformation (T_2), and has been defined in XPand [35]. First, the process starts with T_1 by means of which the *UML diagram models* are translated into transitional models (*template models*), which are basically the PROV templates serialised in XMI (a serialisation format independent of any PROV-compatible format). Subsequently, the transformation T_2 takes place, by means of which these *template models* are translated into the *PROV template files*, which are written in a PROV compatible format (PROV-N in this case).

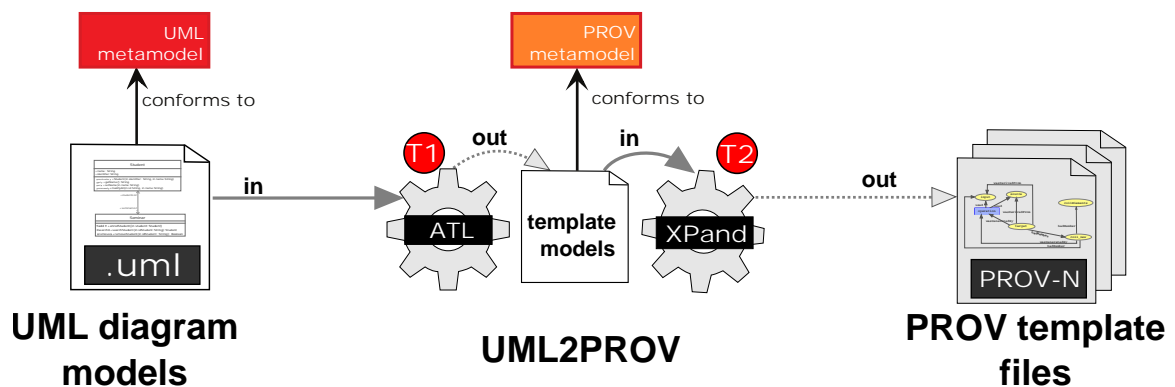


Fig. 5.3 Detailed MDD-based implementation of the PROV templates generation process

Transformation T_1 : from UML diagram models to template models

This M2M transformation takes as source the *UML diagram models*, conforming to the UML metamodel [1], and generates the corresponding *template models*, conforming to the PROV metamodel [21]. To that end, our transformation patterns serve as the basis for the definition of an ATL module that consists of a set of ATL rules. These rules describe how the UML elements identified by the patterns are translated into the specific PROV elements, and their relations, constituting the corresponding *template model*.

In order to show how these ATL rules look like, Table 5.2 depicts an excerpt of the ATL rule focusing mainly on the implementation of the pattern *StP1*⁴. This rule is equivalent to the previous one in Table 5.1 defined in XSLT. Concretely, Table 5.2 shows, for each fragment of the rule (in the first column), the PROV elements/relations in the template model that are generated by such a fragment (see column “Template model”). Additionally, this

⁴The complete definition of this pattern can be seen in Appendix B

Table 5.2 An excerpt of the ATL rule focusing mainly on the implementation of *StP1*. For each fragment in the excerpt (“ATL source code” column), the PROV elements it generates are provided (“Template model” column) together with a description of the transformation (“Description” column) as well as the graphical notation of the template model (last column).



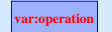

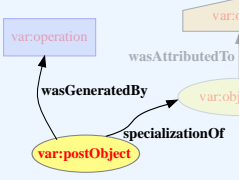
ATL source code	Template model	Description	Graphical representation of the template model
rule transition2Document{ from trans: UMLTransition		It states that the rule is applied to all the UML Transitions in the source model.	
to		This block creates those elements shared by all the patterns addressing State Machine diagrams.	
doc: PROV!Document(),	<document>	It generates the <document>	
ob: PROV!Agent (id<- 'var:object', ...),	<agent id="var:object" />	It creates an <agent> with identifier <i>var:object</i> .	
objSM: PROV!Entity (id<- 'var:objectSM', ...),	<entity id="var:objectSM" />	It creates an <entity> identified by <i>var:objectSM</i> .	
op: PROV!Activity (id<- 'var:operation', ...)	<activity id="var:operation" />	It creates an <activity> with the identifier <i>var:operation</i> .	
do {		Next, it is the code that generates the concrete elements of each pattern, in particular, StP1.	
if(trans.source.ocllsTypeOf(UMLIPseudostate) and trans.source.kind=#initial and trans.target.ocllsTypeOf(UMLIState)){		This excerpt checks if the transition corresponds with the UML elements identified in StP1. That is, if it has an initial pseudostate as source, and a state as target.	
thisModule.genWATAss('var:object', 'var:objectSM');	<wasAttributedTo> <entity ref="var:objectSM" /> <agent ref="var:object" /> </wasAttributedTo>	This excerpt is responsible for linking <i>var:objectSM</i> with <i>var:object</i> by means of the PROV relation <wasAttributedTo>.	
thisModule.genEntity('var:postObject', ...); thisModule.genWGBAss('var:postObject', 'var:operation'); thisModule.genSpeAss('var:postObject', 'var:objectSM');	<entity id="var:postObject" /> <wasGeneratedBy> <entity ref="var:postObject" /> <activity ref="var:operation" /> </wasGeneratedBy> <specializationOf> <specificEntity ref="var:postObject" /> <generalEntity ref="var:objectSM" /> </specializationOf>	First, it generates an <entity> identified by <i>var:postObject</i> . Subsequently, it creates two PROV relations. (1) The PROV relation <wasGeneratedBy> between <i>var:postObject</i> and <i>var:operation</i> . (2) the PROV relation <specializationOf> between <i>var:postObject</i> and <i>var:objectSM</i> .	
[...]		Below it is the code implementing the remaining SMD patterns	
}			

table also depicts how PROV elements such as *document*, *entity*, and *activity*, in addition to PROV relations such as *wasAttributedTo*, appear in the column as <document>, <entity>, <activity>, <wasAttributedTo>, and <wasGeneratedBy>. Finally, it also shows in the two right-hand columns the description of the fragment together with the graphical notation of the template model being generated. This table has been organised in the same way as Table 5.1. Thus, it is easy to compare both implementations. We remark two main differences. First, in the XSLT implementation, the resulting text of the PROV template files is directly defined inside the XSLT template (e.g., `agent(var:object, ...)`), which implies that a change in the serialisation requires to update the XSLT template. As opposed to this, in the ATL implementation (Table 5.2), we can see that the elements to be generated in the *template model* are generated by calling elements from the PROV metamodel (e.g., `PROV!Agent`, `PROV!Entity`, and so on). With this implementation, we draw a distinction between the

transformation from *UML diagram models* to *template models* and the serialisation format of the *PROV template models*. The second main difference is about the fragment that checks if the transition corresponds with *StPI* (located inside a frame in both tables). In XSLT, the excerpt relies on how the UML models are serialised in XMI, which implied a great knowledge in the XMI specification. Conversely, the ATL implementation relies on the UML metamodel, which leads to more clear code and, additionally, there is no need to be knowledgeable in XMI.

Transformation T₂: from template models to PROV template files

This second transformation, identified as T₂, is an M2T transformation that takes as input the *template models* resulting from T₁, and automatically produces the *PROV template files* in PROV-N format. We have implemented this transformation in an XPand module which consists of XPand templates that associate each PROV element/relation with its PROV-N representation. One potential user could define its own XPand templates in order to obtain the PROV template files in another PROV-compatible format. Among the implemented XPand templates, there is a main template (see Figure 5.4) which translates each PROV <document> from the *template models* (line 1) into a *PROV template file* (line 2), defined as a .provn extension text file. This PROV-N document will include: (1) the fixed text, shown in green in Figure 5.4, and (2) the text resulting from instantiating those XPand templates responsible for translating the PROV elements/relations included in the <document> (lines from 3 to 15).

```

1: «DEFINE documentTemplate FOR Document »
2: «FILE id+ ".provn"»

Fixed text |
  document
  prefix prov<http://www.w3.org/ns/prov#>
  prefix tpl <http://openprovenance.org/tmpl#>
  prefix var <http://openprovenance.org/var#>
  prefix ex <http://example.org/>
  prefix u2p <http://uml2prov.unirioja.es/ns/u2p#>
  bundle ex:bundle1

3:   «EXPAND entityTemplate FOREACH entity »
4:   «EXPAND activityTemplate FOREACH activity »
5:   «EXPAND agentTemplate FOREACH agent »
6:   «EXPAND wgbTemplate FOREACH wasGeneratedBy »
7:   «EXPAND usedTemplate FOREACH used »
8:   «EXPAND wibTemplate FOREACH wasInvalidatedBy »
9:   «EXPAND wdfTemplate FOREACH wasDerivedFrom »
10:  «EXPAND hmTemplate FOREACH hadMember »
11:  «EXPAND spOTemplate FOREACH specializationOf »
12:  «EXPAND watTemplate FOREACH wasAttributedTo »
13:  «EXPAND wawTemplate FOREACH wasAssociatedWith »
14:  «EXPAND wInfByTemplate FOREACH wasInformedBy »
15:  «EXPAND wStartedByTemplate FOREACH wasStartedBy »

Fixed text |
  endBundle
  endDocument
  «ENDFILE»
«ENDDFINE»

```

Fig. 5.4 XPand templates defined for each <document> and <entity> in the *template models*

Figure 5.4 also depicts the XPand template (next to line 3) that translates each `<entity>`, in the source *template model*, into the corresponding `prov:Entity` in PROV-N.

5.3.2 Automatization of the generation of the BGM: an event-based and an XPand-based approach

Here, we explain in detail our reference implementation for the automatic generation of the BGM corresponding to a certain application.

Implementation of the BGM

In order to provide a fully automatic solution to generate bindings, which does not require the manual instrumentation of the application's source code, we have defined a generic *event-based* approach to implement the BGMs. While *events* are notable occurrences that happen while the application is running, *listeners* contain the behaviour for processing the *events*. Due to the fact that BGMs must capture provenance from operations executions (see Feature 2 in Section 4.4), we have identified four notable types of occurrences that may take place during an operation execution, and which correspond to four types of *events*, respectively. Two of these *events* are related to the start and end of an operation. The two remaining *event* types refer to the collection of values associated with two different types of variables (*group variables* and *statement-level variables*), which have been stated in [176] for being useful for the expansion algorithm. Figure 5.5 shows a `prov:Activity` in PROV-N with variables occurring in different positions within the definition of the activity.

While a *group variable* is a type of variable that occurs in a mandatory identifier position (e.g., `var:operation`), a *statement-level variable* is a variable that occurs in an optional identifier position (e.g., `var:operationStartTime`), or that occurs in an attribute-value pair, either in an attribute position or in a value position (e.g., `var:operationName`).

In this context, the *event* types we have defined are the following:

- (1) *operationStart* and (2) *operationEnd*. These types of *events* refer to the start and end of an operation execution, respectively. As we will see later, these types of *events* are of interest when developers are interested in creating and storing sets of bindings associated with a concrete operation execution, instead of storing each binding independently.

<code>activity</code>	<code>(var:operation,</code>	<code>var:operationStartTime,</code>	<code>var:operationEndTime,</code>	<code>[prov:type=var:operationName])</code>
	mandatory identifier position	optional identifier position	optional identifier position	value position

Fig. 5.5 PROV activity in PROV-N [5] with different types of variables

- (3) *newBinding*. This type of *event* refers to the occurrence of the collection of a provenance value associated with a *group variable*. For instance, the collection of a value associated with the variable `var:operation` in Figure 5.5 will trigger an *event* of type *newBinding* since `var:operation` occurs in a mandatory identifier position.
- (4) *newValueBinding*. This type of *event* refers to the occurrence of the collection of a provenance value associated with a *statement-level variable*. For instance, the collection of values associated with `var:operationEndTime` and `var:operationName` in Figure 5.5 fires *newValueBinding* events due to `var:operationEndTime` occurring in an optional identifier position, and `var:operationName` occurring in a value position.

Our reference implementation of BGM consists of four main components written in Java (see Figure 5.6) which are divided into two main groups: *context-independent components* and *context-dependent components*. The *context-independent components* group is made up of those elements that do not depend on the source *UML diagram models*, and therefore, they are the same in all the BGMs. This group encompasses the *BGMEventListener*, *BGMEvent*, and *BGMEventManager* (see components in white background in Figure 5.6). The second group, called *context-dependent components*, is made up of those elements whose implementation depends on the source *UML diagram models*. Concretely, in this reference implementation the only element included in this group is the *BGMEventInstrumenter* (depicted in dark background in Figure 5.6). Next, we describe these components in detail:

- *BGMEventListener*. It is a Java interface that defines four operations for managing each type of *event* (*operationStart*, *operationEnd*, *newBinding*, and *newValueBinding*). Such operations have an input parameter of type *BGMEvent* (see below) that contains the

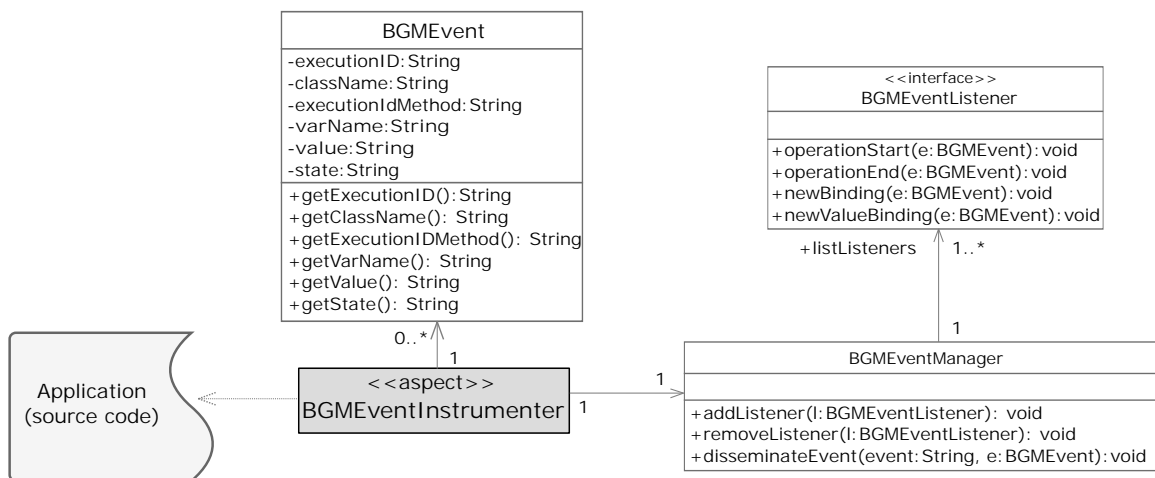


Fig. 5.6 UML Class diagram depicting our reference implementation for a BGM

provenance data to be processed. The implementation of these operations constitutes the mechanism used by a class implementing this interface to generate, manage, and store the bindings. Hence, developers are free to choose the mechanisms that best suits their requirements by developing classes implementing the *BGMEventListener* interface, without coupling this task to a concrete persistence infrastructure as other provenance systems do (see Chapter 3). In this way, this reference implementation of BGM meets the Requirement 4 defined in Section 4.4, which states that developers must be able to select when to compute provenance (i.e., when to expand the PROV templates with the collected bindings). Later, we will give three different implementations of this interface, which manage bindings in three different ways (called *configurations*). At this point, we highlight that with the aim of simplifying the design, we have grouped all the operations for managing the abovementioned *event* types in the same interface (*BGMEventListener*). In case a developer is not interested in handling a concrete *event*, she/he can leave empty the implementation of its corresponding operation.

- *BGMEvent*. This component is used to carry information about the occurrence of an *event*, which corresponds to the provenance data to be processed. We have decided to use the same class *BGMEvent* to contain information about the four event types (*operationStart*, *operationEnd*, *newBinding*, *newValueBinding*) because such information can be stored using the same structure. Concretely, this structure will contain the provenance data necessary for constructing the bindings. Among the attributes of the *BGMEvent*, we remark the attribute `varName` for the name of the variable to which the binding corresponds, and the attribute `value` for the value associated with such a variable (see Figure 5.6). For instance, in case of an *operationStart event*, a *BGMEvent* object could have an attribute `varName` containing the value "var:operationStartTime", and an attribute `value` with the value "2019-07-17T14:21:12.08584". Another example could be a *BGMEvent* object with information about a *newBinding event*, where the attribute `varName` could contain the value "var:operation", and whose attribute `value` could have "ex:enrolStudent1" as value. Readers may notice that the values of these `varName-value` attribute pairs correspond to the set of bindings previously depicted in Figure 4.11 (Chapter 4). Thus, those bindings could have been constructed using this information.
- *BGMEventManager*. In some circumstances, having only one *listener* for managing bindings could not be enough, and the same happens with the mechanisms for generating and storing provenance. For instance, one provenance consumer may be interested in replicating the information by storing in different storage systems both the provenance data, and the bindings generated from them. To address these scenarios, we have decided to

include the *BGMEventManager* component with two main responsibilities: to manage a list of subscribed *listeners* (`addListener` and `removeListener` operations), and to disseminate the objects of type *BGMEvent* among them (`disseminateEvent` operation).

- *BGMEventInstrumenter*. As we mentioned in the Requirement 1 for BGMs of Section 4.4, to manually adapt the source code of an application could be a valid option to capture provenance data. Nevertheless, this strategy would require a developer to traverse the whole code of the concrete application identifying the classes that will be the source of the *events*, and those places within these classes where *events* will be fired. Then, the developer would have to include in those places instructions not only for constructing *BGMEvent* objects with the provenance data, but also for disseminating such events among the *listeners*. This task constitutes a tedious, time-consuming and error-prone process. Additionally, the manual adaptation could incur in such instructions for provenance capture scattered across all the application classes, making their maintenance a cumbersome task. To avoid such inconveniences, we propose to use the AOP paradigm [40] for implementing what we have named *BGMEventInstrumenter*. As we already presented in Chapter 2, AOP aims at improving the modularity of software systems, by capturing inherently scattered functionality, often called *cross-cutting concerns*, (e.g., the capture of provenance), and placing that functionality apart from the actual application’s source code. More specifically, in our reference implementation, we have used the AOP extension for Java, AspectJ [41], to implement the *BGMEventInstrumenter* by means of an *aspect*, which is made up of an *advice* with a *pointcut* (see Figure 5.7). On the one hand, the *pointcut* identifies locations within the application code where we want to fire *events* (i.e., to collect provenance data). In our case, we identify operation calls (`call` pointcut designator) and constructor invocations (`initialization` pointcut designator). More specifically, in the *pointcut* of Figure 5.7 we have denoted the identified operations as “<objectClass>.<operation>”, where <objectClass> would correspond to the full class name of the objects involved in

```

aspect
public aspect BGMEventInstrumenter{
    Pointcut
    Object around(): call(*<objectClass>.<operation>(..)||initialization(<objectClass>.new(..)){
    advice
        behaviourBeforeExecution();
        Object rtn = proceed();
        behaviourAfterExecution();
        return rtn;
    }
}

```

Fig. 5.7 Structure overview of a reference implementation of the *BGMEventInstrumenter* in AspectJ

the UML design, while *<operation>* would refer to the name of each identified operation. Likewise, we have denoted the constructor invocations of each class in the UML design as *<objectClass>.new*. To construct the final *pointcut*, the expression *call(...)* would have to be repeated per each identified operation, whereas *initialization(...)* would have to be repeated per each class in the UML design. On the other hand, the *advice* describes the behaviour executed when the *pointcut* is matched (*before*, *after* or *around* the *pointcut*). Since our identified *events* can occur both before and after operations calls and constructors invocations, we have used an *around advice* for executing custom behaviours before and after the actual behaviour. These custom behaviours consist of constructing objects of type *BGMEvent* and disseminating them to the *listeners* (by invoking the *disseminateEvent* operation from *BGMEventManager*). In the end, the AspectJ *weaver* automatically integrates the behaviour from the *aspects* into the locations specified by the *pointcut* at compilation time. In this way, our AOP approach does not require a manual intervention for adapting the source code, and automatically collects provenance data in a transparent way for software developers. In this way, we fulfil the requirements *R1-R3* stated in Section 4.4.

Several implementation configurations for the *BGMEventListener*

Here, we present three configurations for managing bindings through three different implementations of the *BGMEventListener* interface. We will refer to these configurations by using the name of the class that implements the *BGMEventListener* interface in each configuration (i.e., *BindingsBGMEventListener*, *SetBindingsBGMEventListener*, and *ProvenanceBGMEventListener*). The reason why we present three different configurations is that we will evaluate their time and spatial overheads in Section 6.1 of Chapter 6. Thus, these configurations have been defined taking into account those aspects that incur in overhead. As presented in Section 3.4, those categories from the systematic review that were related to temporal and spatial overheads were *granularity*, *tracing*, *level*, and *technique*. Whereas the *granularity*, the *level* and the *technique* are intrinsic aspects of the UML2PROV proposal, the *tracing* category, which refers to whether provenance is computed on demand (*lazy*) or as the application executes (*eager*), does not. In particular, the *BGMEventListener* could be implemented by several classes, each considering a different strategy for computing provenance (*lazy* or *eager*). In fact, providing *software developers* with mechanisms to decide when to compute the provenance is one of the stated requirements in Section 4.4 (Requirement 4). Additionally, this mechanism also allows developers to use the persistence system more suitable for their needs. For the evaluation of our reference implementation we have used MongoDB [177] as persistence system. We have made the decision of using only

one persistence system due to the fact that we are interested in a comparative analysis between different configurations of the BGM, and we consider that the different characteristics of the persistence systems may add noise to such an analysis. We consider that the evaluation of the performance of UML2PROV with other persistence systems is out of the scope of this thesis, since it depends on the characteristics of such systems.

Concretely, we have considered two *lazy* configurations (*BindingsBGMEventListener* and *SetBindingsBGMEventListener*) that compute provenance (i.e., expands the templates) on demand. The main difference between these configurations is that the *BindingsBGMEventListener* corresponds to the simplest way of implementing the *BGMEventListener* interface, whereas *SetBindingsBGMEventListener* adds more complexity in its implementation (later, we will see more details). Furthermore, we have considered an *eager* configuration (*ProvenanceBGMEventListener*) that computes provenance as the application executes. These configurations are illustrated in Figure 5.8, which include two main blocks. On the left-hand

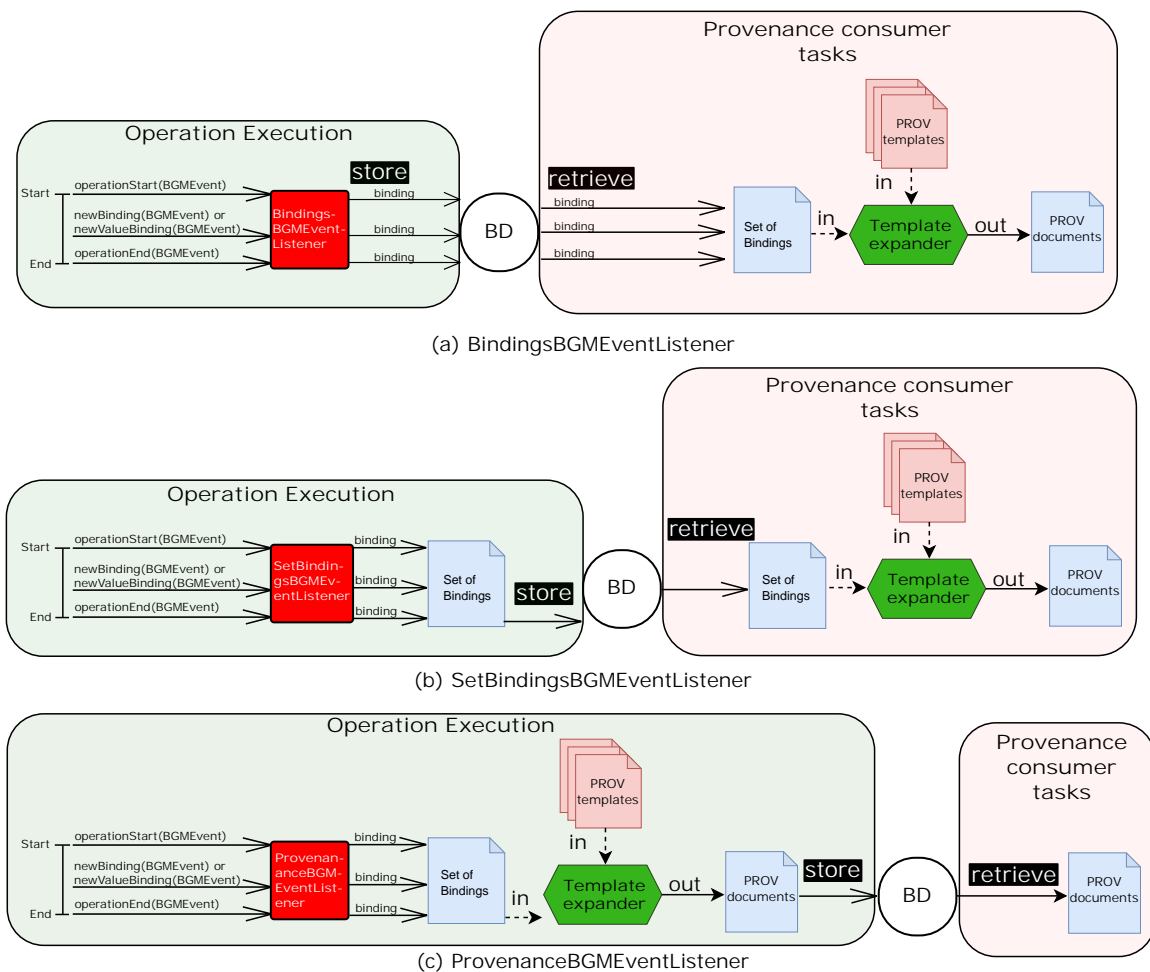


Fig. 5.8 Graphical representation of the three defined configurations

side (block “Operation Execution”), it is the behaviour carried out during the execution of a tracked operation, from the start to the end of the operation execution. On the right-hand side (block “Provenance consumer tasks”), it is the behaviour executed to exploit the provenance information, and which takes place after the execution of the tracked operation. Next, we present these three *configurations* in detail:

- *BindingsBGMEventListener*. In this configuration (see Figure 5.8(a)), PROV templates are expanded on demand (i.e., a *lazy* approach). This class corresponds to the simplest implementation of the *BGMEventListener* interface. Concretely, it implements all the operations from the *BGMEventListener* interface, so that each time they are executed, the provenance data contained in the *BGMEvent* is shipped to the database in the form of *binding*. In this way, when the provenance consumer wants to generate the final *PROV documents*, she/he has to: (1) retrieve each *binding* from the database, (2) create the *set of bindings*, and (3) instantiate the *PROV templates* with the created *set of bindings*.
- *SetBindingsBGMEventListener*. As the *BindingsBGMEventListener*, in this configuration (see Figure 5.8(b)) PROV templates are expanded on demand (i.e., a *lazy* approach). This class implements the four operations defined in the *BGMEventListener*, so that they generate *bindings* as the tracked operation executes, and accumulate them in memory until the execution of the tracked operation finishes. When the execution of the tracked operation ends (i.e., when the `operationEnd` event is fired), the accumulated *set of bindings* is shipped to the database. Thus, this implementation is only in charge of generating and storing *sets of bindings*, delaying the expansion of templates until the tracked operation finishes. In this way, when the provenance consumer wants the *PROV documents*, she/he has to (1) retrieve the *set of bindings* from the database, and (2) instantiate the *PROV templates* with the retrieved *set of bindings*.
- *ProvenanceBGMEventListener*. In this configuration (see Figure 5.8(c)), PROV templates are expanded as the application is running (i.e., *eager* approach). This class implements all the operations of *BGMEventListener*, so that the provenance data in the *BGMEvent* is used to create the *set of bindings* in memory while the tracked operation is running. When the execution of the tracked operation ends (i.e., when the `operationEnd` event is fired), the *PROV templates* are expanded with the generated *set of bindings* and the resulting *PROV documents* are shipped to the database. Thus, when the provenance consumer wants to exploit the provenance, she/he only has to retrieve the PROV documents from the database.

With these three different configurations, a reader can get an idea about the inverse relationship between the number of tasks carried out during the operation execution (defined

by means of a class implementing the *BGMEventListener*) and the effort required from the provenance consumer for exploiting the final provenance. Concretely, the more effort devoted to implementing the *BGMEventListener* interface, the less effort required from the provenance consumer. This is the case of the *ProvenanceBGMEventListener* configuration, whose implementation contains most of the tasks for obtaining the provenance to be exploited (to create the set of bindings, to expand the templates,...), and consequently, the provenance consumer only has to retrieve the PROV documents from the database to exploit the provenance. Conversely, in the *BindingsBGMEventListener* configuration, the implementation effort is limited to store the generated bindings independently, so the effort required from the provenance consumer is very high (to retrieve each binding, to create *set of bindings*, and to instantiate the *PROV templates*). Whilst the *ProvenanceBGMEventListener* is in principle the most suitable implementation, since it requires the lowest effort from provenance consumers, later in Chapter 6, we will see that it incurs in additional run-time overhead and storage cost that developers may not be willing to assume.

Automatization of the implementation of the BGM

As an alternative to XSLT, we decided to follow an MDD proposal similar to that presented for generating the PROV templates in Section 5.2.1. In this case, the BGM for an application is automatically generated by means of an M2T transformation referred to as T_3 in Figure 5.9. Such a transformation has been implemented by means of an XPand module that takes as source the application's *UML diagram models*, conforming the UML metamodel [1], and generates the java code of the BGM.

As we stated previously, the source code of the *context-independent components* (i.e., *BGMEvent*, *BGMEventManager*, and *BGMEventListener*) is the same for all the BGMs; thus, it does not depend on the *UML diagram models* used as input of the transformation.

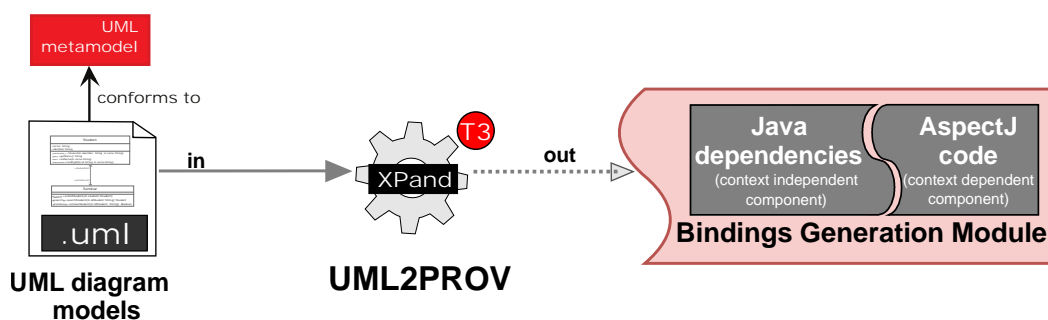


Fig. 5.9 Detailed MDD-based implementation of the BGM for an application

Conversely, the implementation of the *context-dependent component* (i.e., *BGMEventInstrumenter*) depends on the source *UML diagram models*.

Consequently, we have decided to implement the XPand module so that it (1) directly creates all the *context-independent components*, and (2) generates the *BGMEventInstrumenter* based on the source UML design. Whilst we could have provided users with a separate library including all the *context-independent components*, we have made the decision of generating them automatically together with the *BGMEventInstrumenter* in order to reduce the code dependencies.

In particular, the XPand module generates the *BGMEventInstrumenter* so that its *pointcut* identifies the calls to operations and invocations of constructors. Concretely, this *pointcut* matches (1) the invocations of the constructors of classes involved in the UML design, and (2) the calls of operations that are involved in the source diagrams: SqDs (i.e., the operations whose calls are modeled by means of UML Messages), SMDs (i.e., the operations whose occurrences are associated with UML Events), and CDs, (i.e., the operations that are modeled by UML Operations). The remainder source code of the *BGMEventInstrumenter* (that is, the *advise*) is also shared by all the BGMs.

5.3.3 Fulfilment of BGM requirements

We would like to note that the reference implementation of the BGMs given in this document fulfils the six requirements stated in Section 4.4 (identified from *R1* to *R6*). As we previously mentioned, requirements from *R1* to *R3* have been met thanks to the AOP implementation of the *BGMEventInstrumenter*. As for *R4*, we have noted that it is satisfied by the different configurations a *developer* can make by implementing the *BGMEventListener* interface. Regarding the requirements *R5* and *R6*, we remark that they have been satisfied because of the suitable ad hoc implementation of the *BGMEventInstrumenter* for a concrete application (explained in the previous section). On the one hand, the automatically generated *pointcut* inside the *BGMEventInstrumenter* ensures that the collected bindings are associated with at least one PROV template (requirement *R5*). This is because the *pointcut* corresponds to operations calls and constructors invocations that are modeled in the UML design, and therefore they have an associated PROV template. On the other hand, the requirement *R6* is fulfilled since the transformation T3 has been implemented so that it respects the names of the variables appearing in the PROV templates generated by the chain of transformations T1-T2.

5.4 Conclusions

In this chapter we have mainly introduced a reference implementation of the conceptual definition of UML2PROV, which addresses the automatic generation of the two main artefacts of UML2PROV: PROV templates and the BGM. This reference implementation was not our first implementation attempt, but we originally developed a proof of concept in order to validate whether the conceptual definition was feasible and if its implementation was possible. This proof of concept considered only two types of UML diagrams (UML Sequence and State Machine Diagrams) used to design an application. Concretely, we used (1) XSLT to implement the transformation patterns, and (2) the *Proxy-pattern* to develop the BGM for generating bindings. It is remarkable that, although this implementation was a feasible solution at the beginning, we found that both XSLT, for defining the transformation patterns, and the *Proxy-pattern*, for implementing the BGM, were no longer the best suitable options. On the one hand, XSLT incurred in several drawbacks regarding maintenance, reusability and support to software development processes. On the other hand, the *Proxy-pattern* forced the developers to manually instrumentate the application's source code.

After this proof of concept, we developed a reference implementation considering the whole conceptual definition of UML2PROV. In this reference implementation, we used MDD-based approaches, more suitable than XSLT, to perform the transformations from the source models to the corresponding targets. As for the implementation of the transformation patterns, it is worth remarking that it could be used 'as it is', without any modification, by any potential user seeking to obtain the provenance design from the UML design. This is because this implementation is agnostic about the technologies and programming languages used to develop the target application. Regarding the implementation of the BGM, it is automatically generated by an XPand module. The generated BGMs fulfil the six requirements stated in Section 4.4, so that the generation of bindings boils down to deploy the BGM into the target application. As we decided to develop this reference implementation in Java, we note that the usage of the generated BGMs is limited to Java applications. To address this drawback and allow potential developers to implement their own BGM, the reference implementation of the BGM is built on top of a generic structure (depicted in Figure 5.6) that could be implemented in other programming languages. The unique requirement for using this structure is to have an AOP implementation compatible with the programming language of the target application, which we do not consider a hindrance because, since the initial release of the AOP concept, a large number of implementations for various programming languages have emerged (such as Python, C, Ada, JavaScript, C#, and so on).

All in all, with this reference implementation, we provide a feasible and complete implementation of the conceptual definition given for UML2PROV. Concretely, this implementation automatically generates the PROV template files (with the provenance design) and the BGM (to be deployed in the application for collecting bindings) from UML diagrams. In this way, software developers only have to deal with the design of the UML diagrams without needing to be knowledgeable about provenance, since UML2PROV is in charge of taking such UML design and automatically generating the artefacts for obtaining provenance. In the following chapter, we provide an extensive and systematic evaluation of this reference implementation. In this evaluation, we will use both quantitative data and qualitative arguments to show the benefits and trade-offs of our overall proposal.

Chapter 6

Evaluation

In the present chapter, we describe the analysis we have carried out to evaluate the benefits and trade-offs of using UML2PROV, in general, and our reference implementation of UML2PROV, in particular. More specifically, we have applied our reference implementation to a legacy Bioinformatics application called GelJ [178], which has been adapted to be provenance-aware, giving support for 9 different provenance requirements raised by its users. Below, we start by introducing this evaluation. Subsequently, we present GelJ together with the *design strategies* that we have defined to obtain different UML designs of GelJ, which state the basis for benchmarking UML2PROV. Then, we analyse the benefits and trade-offs of using UML2PROV based on five aspects regarding provenance, and finally, we end up with a conclusion block.

The work presented in this chapter has been partially submitted for publication in:

- Carlos Sáenz-Adán, Beatriz Pérez, Francisco J. García-Izquierdo, Luc Moreau:
Integrating Provenance Capture and UML with UML2PROV: Principles and Experience.
Submitted for publication in IEEE Transactions on Software Engineering.

6.1 Introduction

The provenance capture has been shown to have implicit costs [45], as any other computational functionality. One of the main goals of this chapter is to show the UML2PROV costs and benefits to software engineers seeking to make applications provenance-aware.

In UML2PROV, UML diagrams drive both the design and the capture of provenance, so that the intuition is that the systems' diagrams themselves have implications on a number of aspects regarding the generation of provenance. For example, for reasons that will

be clear later on, taking a detailed UML design as source of UML2PROV may incur in the capture of more provenance than required to satisfy certain provenance requirements, resulting in unnecessary overheads. Conversely, if the UML design is limited to tackle certain provenance requirements, only relevant provenance will be generated, thus reducing the overhead. Nevertheless, it is worth noting that adapting the UML design to precisely fit provenance requirements demands additional effort from the designer. Concretely, the more effort is devoted to adapting the UML diagrams to selectively expose the relevant information for certain provenance requirements, the less provenance collected and the less run-time overhead and storage needs.

As we have stated previously, to show both the benefits and trade-offs of using UML2PROV, we have decided to apply UML2PROV to a Bioinformatics application called GelJ [178], which originally lacks a UML design. Aimed at analysing the implications of using UML2PROV with different UML designs, we have considered what we have named *design strategies* (see Section 6.2). More specifically, we have defined three different design strategies that yield three UML designs of GelJ with different levels of detail. On the basis of these resulting UML diagrams, we have analysed the implications of the design in the generation of provenance with UML2PROV. Concretely, this analysis has been made based on five aspects: *generation of the provenance design, instrumentation of the application, maintenance of provenance capabilities, run-time overhead and storage needs, and quality of provenance*. At this point, we note that the *run-time overhead and storage needs* aspect directly depends on the configuration of the BGM. Thus, this aspect not only has been analysed on the basis of different UML designs (produced from the design strategies), but also through the three different configurations of the BGM provided by our reference implementation (described in Section 5.3.2).

6.2 GelJ and the Design Strategies

GelJ [178] is an open-source tool used in the realm of Bioinformatics in order to analyse DNA fingerprint gel-images (hereinafter *gel-images*), and to compare DNA patterns. More specifically, this tool allows the analysis of the genomic relatedness among different samples, as well as their classification. The analysis of DNA patterns has applications in different fields such as in medical diagnosis, forensic science, parentage testing, food industry, agriculture, and many others [179]. This tool has been selected for three main reasons. First, it has a large user community with needs for provenance for their DNA analysis. Second, the tool lacks a UML design, which allows us to show how to deal with this type of applications. More specifically, it will give us the possibility to show the trade-offs between the effort devoted to

obtaining the UML diagrams and the benefits of the generated provenance. Third, we have direct contact both with developers and potential users of GelJ. In particular, the developers can provide us with meaningful information about the implementation of the tool, whereas potential users can assist us to assess the quality of the provenance obtained by UML2PROV against their real needs.

GelJ includes an *experiment wizard* that users must use for analysing DNA fingerprint gel-images. More specifically, as depicted in Figure 6.1, after choosing a gel-image as source, this wizard guides the user through a process that consists of four steps. In Step 1, the user may perform operations to increase the quality of the source gel-image. In Step 2, GelJ automatically detects the lanes of the gel-image. Subsequently, the user might have to perform some adjustments over the lanes such as adding new ones, removing or editing. Finally, among all the detected lanes the user selects one or several as reference. In Step 3, a normalization task is performed by means of which GelJ automatically detects markers in the reference lane. Subsequently, the user might have to handle these markers by adding or removing some of them. In Step 4, GelJ automatically detects bands inside the detected lanes; the user might modify these bands later (e.g., adding, moving, and removing). The result of this workflow is an *experiment* consisting of the source gel-image together with a set of detected bands (see Figure 6.1). At this point, it is worth remarking that GelJ's database only stores concrete characteristics with regard to an experiment (e.g., its name, the user who performed it, and biological information such as genus, species, . . .). It does

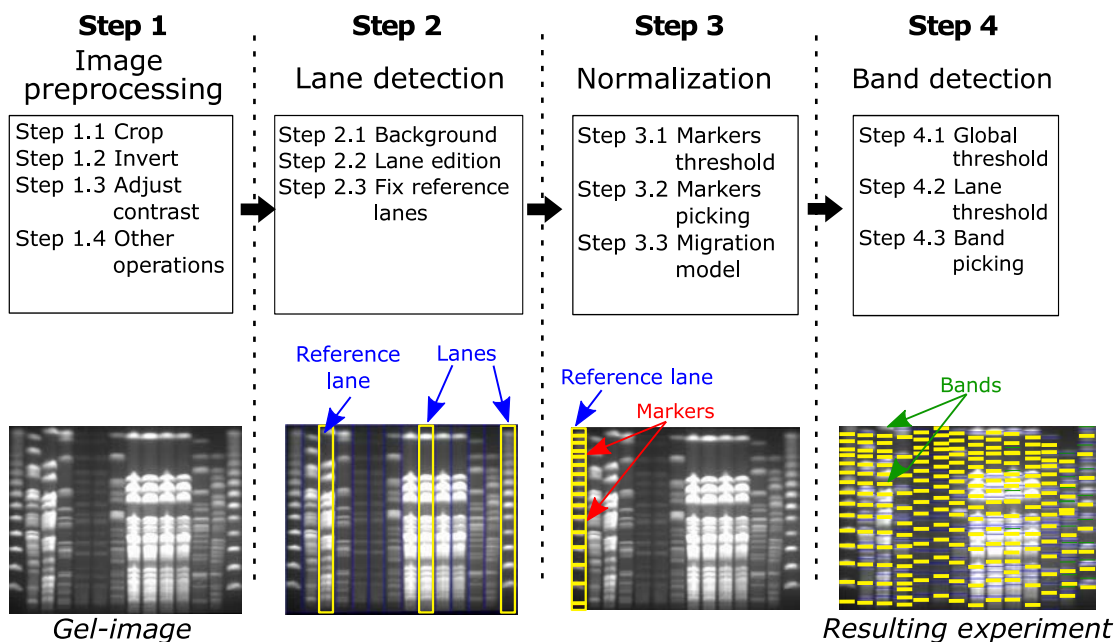


Fig. 6.1 Workflow of GelJ's *experiment wizard*

not store aspects such as the steps followed during the experiment creation or its origin (an experiment can be created from scratch, using the *experiment wizard*, duplicated or imported). Therefore, scientists usually have to ask other colleagues about the process followed to create an experiment, its authorship, or its origin.

In order to provide an unbiased evaluation of our proposal, we have selected an experiment with a concrete sequence of interactions among the 13 substeps of the GelJ *experiment wizard*. To select this sequence of interactions, we have analysed the successive actions of a user to generate 10 different experiments, each one for a different source gel-image. While the user worked, we wrote down the performed interactions with GelJ for creating those experiments. Among the 10 collected traces, we selected the one with the higher number and diversity of interactions to be used in obtaining the data for the performance evaluation. Concretely, the selected experiment comprises a sequence of 114 interactions among the 13 substeps of the GelJ *experiment wizard* (this sequence of interactions is shown in Appendix C).

6.2.1 UML design of GelJ

As stated, the starting point of UML2PROV for making an application provenance-aware is the UML design of the application. Thus, a key challenge regarding GelJ is that it is a legacy application that, in particular, lacks a UML design. So, aiming to use UML2PROV for making GelJ provenance-aware, we have decided to leverage reverse-engineering in order to obtain the UML design from the GelJ source code. Concretely, reverse-engineering is focused on the challenging task of understanding legacy program code without having suitable documentation [180]. This is done mainly by reconstructing a design that could have been used to specify the application implementation. Performing reverse engineering is known to be challenging [180], and thus, automatic techniques are desirable. These techniques can employ *static* or *dynamic* analysis.

Static analysis The structure of the software is described as it is written in the source code [181]. It relies on complex source code analysis, and on specific techniques (such as symbolic execution) to identify operation call sequences in the source code; thus, it may become challenging to apply it to large systems [182]. Several well-known CASE tools can obtain automatically CDs from code, e.g., Visual Paradigm [183], Papyrus [184], and Modelio [185], among others. Likewise, ObjectAid [186] applies *static* analysis for obtaining SqDs by checking the call hierarchy of a selected operation. In this case, to obtain SqDs demands more effort than for CDs since it requires each operation to be traversed.

Dynamic analysis It is defined by Ball as “the analysis of the properties of a running software system” [187]. This analysis usually involves instrumenting an application to

collect certain behaviour as the application is running. Thus, it gives information based on the behaviour captured from the execution rather than from the source code. Approaches such as MaintainJ [188] and the one given by Briand et al. [189] employ *dynamic* analysis to generate SqDs. This analysis is completely automatic and only requires to execute the application for capturing the behaviour to generate the concrete SqD.

As far as the reverse engineering of SMDs is concerned, we would like to note the scarce literature addressing this issue, regardless of the used type of analysis. Whilst the existing approaches show efficacy in contexts such as pattern identification and conformance checking [190], to the best of our knowledge, they are not able to automatically extract high-level abstract information that is not included in the code and is only known by the designer (e.g., descriptive names for the states). For instance, in [191] the states are named as q_N , in which N is a numeric identifier from 0 to 6. From our point of view, descriptive states' names are of great importance since they provide high-level information that cannot be inferred from CDs and SqDs.

Taking this into account, while we have chosen concrete reverse-engineering techniques to obtain the SqDs and CDs of the GelJ UML design, we have proposed to use SMDs manually defined by the designers of GelJ, so that they may provide SMDs' states with more meaningful names.

6.2.2 Strategies for obtaining GelJ design

To perform the evaluation of UML2PROV, we have identified three strategies in which the application of different reverse-engineering techniques, each requiring a different effort from the software engineer, leads to three different UML designs for GelJ. The reason why we have defined these three strategies is to cover a wide range of situations in which UML2PROV must be applied. It must be noted that assessing the level of effort is difficult, and often imprecise, because it closely depends on the software engineer's ability to perform each strategy. However, it is possible to give an insight by identifying those tasks performed manually and those carried out automatically. The more manual tasks, the greater the effort. Ultimately, each strategy comprises a set of tasks, some of them relative to analysis activities, and others to the design itself of the UML diagrams describing the system. The description of each strategy has been organised into four blocks that characterise it. The first block is the *aim* for which the strategy is proposed. In this block we will specify the types of UML diagrams we are interested on, and whether the UML design considered in the strategy derives from some provenance requirements. The second block is the *provenance analysis*. In case of having considered provenance requirements, in this block we explain how we

identify the provenance requirements and the involved system's elements. The third block is the *UML design generation*, in which we explain the use of reverse-engineering for obtaining the different UML designs of GelJ. Finally, the fourth block is the *required effort*, in which we sum up the effort that is needed to perform the tasks involved in the strategy. Here, we highlight if the tasks performed in the *UML design generation* are automatic, manual, or semi-manual. More specifically, the set of tasks comprising each strategy are summarized in Table 6.1, which can be used to easily compare the characteristics of each strategy. Next to each task there is a symbol indicating if the task is automatic (▶), manual (✋), or semi-manual (✋▶). Under each set of tasks, we depict a summary of its outcome in the form of number and type of elements generated by performing such tasks. In particular, in the *provenance analysis* block, these generated elements correspond to the provenance requirements taken into account in the strategy, as well as the GelJ code artefacts involved in fulfilling them. In the *UML design generation* block, the resulting elements refer to the UML elements making up the design. Next, we describe in detail each strategy.

Table 6.1 Overview of tasks performed to obtain the UML design in each strategy. Under each set of tasks is the number of elements that result from them.

Phase		Strategy 1	Strategy 2	Strategy 3
Provenance Analysis		Tasks: ✋ (T1.1) To identify provenance requirements ✋ (T1.2) To identify classes and operations involved in the provenance requirements Result: 9 provenance requirements / 17 classes / 66 operations	None -	None -
	SqD	Tasks: ✋▶ (T1.3) Static reverse engineering by identified operation Result: 76 messages	▶ (T2.1) Dynamic reverse engineering 133 messages	None -
UML design generation	SMD	Tasks: ✋ (T1.4) Manual generation Result: 33 states and initial pseudostates / 68 transitions	None -	None -
	CD	Tasks: ▶ (T1.5) Static reverse engineering ✋ (T1.6) To rule out classes and operations not identified ✋ (T1.7) To add stereotypes to the identified operations Result: 17 classes / 66 operations	▶ (T2.2) Static reverse engineering 279 classes / 1688 operations	▶ (T3.1) Static reverse engineering 279 classes / 1688 operations

▶ Those tasks performed automatically ✋ Those tasks requiring manual effort ✋▶ Those tasks requiring semi-manual effort

Design Strategy 1

- *Aim.* To obtain a UML design encompassing UML Sequence, State Machine, and Class diagrams that only exposes the information necessary to fulfil a set of stated provenance requirements.
- *Provenance analysis.* Here we have been inspired by the first phase of PrIMe [11] which, as we have mentioned in Section 3.4, is a methodology for adapting applications to become provenance-aware. More specifically, we involved a group of GelJ users, asking them for the typical questions they are seeking to answer, and which the current system cannot answer (task T1.1 in Table 6.1). Among these questions, first, we excluded those that can be answered by using information stored in the GelJ database, and second, we refactored them in terms of PROV. These resulting questions, shown in Table 6.2, reflect the provenance

requirements (called *provenance use case questions* in PrIME). Subsequently, inspired by the second phase of PrIME, in collaboration with the developers, we identified those classes and operations (called *actors* in PrIME) involved in answering the above questions (T1.2). We show such classes in column “Identified classes” of Table 6.2. To determine them, we performed the following process. For each class included in GelJ’s CD, the developers checked if the class was involved in answering any of the identified questions; in case of doubt, it was selected. Subsequently, for each operation of the selected classes, the developers confirmed if it was involved or not in the identified questions; again if they had doubts, it was selected.

As a result of these tasks, we identified 17 classes and 66 operations out of the 279 classes and 1688 operations that compose GelJ’s CD. Thus, we can say that, in the *UML design generation*, we used $\sim 6\%$ of the classes and $\sim 3.9\%$ of the operations of GelJ.

Table 6.2 Questions identified from Q1 to Q9 raised by GelJ users, together with GelJ classes involved in answering those questions

ID	Provenance requirements	Identified classes
Q1	What is the origin of an experiment? (from scratch, duplicated or imported)	Experiment Step4_3 Image_Assistant
Q2	What is the set of activities that has led an experiment as it is?	Step1_1-Step1_4 Step2_1-Step2_3 Step3_1-Step3_3 Step4_1-Step4_3
Q3	Which background (dark or light) has been used during an experiment construction?	Step1_1-Step1_4 Step2_1-Step2_3 Step3_1-Step3_3 Step4_1-Step4_3
Q4	Who is the user who has carried out a specific step of an experiment wizard?	Step1_1-Step1_4 Step2_1-Step2_3 Step3_1-Step3_3 Step4_1-Step4_3
Q5	How many lanes have been added/removed during an experiment’s generation process?	Step2_2
Q6	What is the height-threshold used for band detection during the experiment’s generation process?	Step4_1 Image_Assistant
Q7	How many bands have been added/removed during an experiment’s generation process?	Step4_3
Q8	What is the detailed information regarding the pre-processing activities? (source/target states, nested activities called)	Step1_1-Step1_4
Q9	What is the time-cost of creating a new experiment?	Menu Main Step1_1-Step1_4 Step2_1-Step2_3 Step3_1-Step3_3 Step4_1-Step4_3

- *UML design generation*. We used ObjectAid [186] (task T1.3 in Table 6.1) to obtain SqDs with the interactions between the objects of the identified classes. For each class, we performed reverse engineering of each of its operations, obtaining a Sequence diagram representing the interactions (including UML lifelines, messages, etc.) where such an operation was involved. This process resulted in a set of 76 UML messages. Regarding

SMDs, as we said previously, we had to involve the designers of GelJ to obtain SMDs with high-level abstract information that cannot be inferred from the source code (T1.4), such as meaningful states' names. They designed a UML SMD for each class whose states were related to the provenance requirements, which led to a set of 13 state machine diagrams with 33 states and initial pseudostates, and 68 transitions modeling the behaviour of the classes *StepN_M* (where *N* refers to one of the four steps, and *M* to one of its corresponding substeps). Finally, we used Papyrus [184] to obtain a Class diagram containing all the classes of GelJ (task T1.5 in Table 6.1), and finally, we ruled out those classes and operations not identified before (T1.6).

The resulting Class diagram comprises the 17 classes and 66 operations identified in the *provenance analysis*. Additionally, to obtain more meaningful provenance, we have assigned to these resulting UML operations a stereotype from Table 4.1, which denotes the concrete behaviour of the operation (task T1.7 in Table 6.1).

- *Required Effort*. Most of the effort devoted by the software designer to perform this strategy corresponds to the identification of the provenance requirements (T1.1) and the selection of the involved classes and operations (T1.2), both tasks corresponding to the *provenance analysis* phase. Besides, to perform these tasks, the software engineer needed to consult with users and designers of the application. As for the *UML design generation*, the SqDs were obtained by means of a manual traversal of the operations of each class in the CD, followed by an automatic reverse engineering of each operation, that is, we performed a semi-manual process (T1.3), while the SMDs were fully-manually obtained (T1.4). In contrast, the CD was generated automatically (T1.5); however, we had to transverse the generated CD ruling out classes and operations not identified previously (T1.6), and assigning to identified UML operations a stereotype from Table 4.1 (T1.7).

Design Strategy 2

- *Aim*. To obtain a UML design encompassing UML Sequence and Class diagrams not derived from provenance requirements.
- *Provenance analysis*. As this strategy was applied without considering specific provenance requirements, there was no need to perform a provenance analysis, which considerably reduced the effort involved in this strategy.

As a consequence, the subsequent phases took into account 100% of the classes and operations in GelJ.

- *UML design generation.* First, again using MaintainJ [188], we performed dynamic reverse engineering of every class (considering all its operations), resulting in a set of SqDs with 133 messages. Then, we applied Papyrus [184] to obtain a Class diagram with 279 classes and 1688 operations (task T2.2 in Table 6.1).
- *Required effort.* This strategy only involved the automatic tasks carried out in the *UML design generation* to generate the SqDs and the CD (T2.1 and T2.2).

Design Strategy 3

- *Aim.* To obtain a UML design encompassing only a Class diagram not derived from provenance requirements.
- *Provenance analysis.* Since this strategy does not focus on provenance requirements, as with Strategy 2, we took into account 100% of classes and operations.
- *UML design generation.* Same as in Strategy 2, but just for obtaining the CD.
- *Effort required.* This strategy only involved the automatic task carried out in the *UML design generation* to obtain the CD (task T3.1 in Table 6.1).

6.3 Analysing the benefits and trade-offs of using UML2PROV

We have identified five aspects to consider when analysing the benefits and trade-offs of using UML2PROV for provenance generation. As advanced previously these aspects attend to: *generation of the provenance design, instrumentation of the application, maintenance of provenance capabilities, run-time overhead and storage needs, and quality of provenance.* The above stated strategies provide us with a basis for benchmarking UML2PROV throughout the identified aspects, where we put emphasis on the implications of the UML design applied in each case. Additionally, throughout these aspects, we have also taken into account whether the concrete implementation of UML2PROV could have any implication, in particular, in the *run-time overhead and storage needs* aspect. Thus, the analysis of this aspect is based not only on the UML diagrams yielded from the strategies, but also on the different configurations of the BGM provided by our reference implementation.

6.3.1 Aspect 1: Generation of the provenance design

Prior to UML2PROV, software designers had to manually develop the PROV templates with the structure of the provenance to be generated. This constituted a cumbersome, time-consuming and error-prone task which was significantly facilitated by the assistance of PROV

experts working closely with the application developer or designer, to reflect the application’s functionality in the templates [3]. To make matters worse, this procedure does not scale up when the amount of provenance to be designed increases. To address this challenge, UML2PROV makes the design of provenance straightforward, while providing significant benefits for the software engineer. Concretely, we can identify two main benefits. First, the automatic generation of templates avoids human intervention, thus preventing any kind of human mistake. Consequently, it is not required a PROV expert supporting the development of templates, since the templates are automatically generated from the UML design. Second, the time cost of generating the full set of templates, a few seconds (see column “Templates generation time cost” in Table 6.3), may be considered negligible compared to the hours, even days, needed to create them manually.

Regarding the influence of the strategy followed to obtain the UML design, it should not be surprising that part of the results collected in Table 6.3 shows that the closer the UML design fits the application provenance requirements, the less templates are generated, the smaller their total size, and the faster their generation. Table 6.3 shows that Strategy 1 (more adjusted design, with less UML elements, but requiring more effort to obtain) presents the lowest number of templates (198). Conversely, Strategies 2 and 3 (less precise designs, with more elements, and automatically generated), result in a higher number of templates (1,821 and 1,688, respectively). Considering these results, it could be said that a greater initial effort to obtain a more accurate UML design sifted by a set of provenance requirements, as made in Strategy 1, results in fewer templates. As we will see later on, this will positively affect the instrumentation, the maintenance, and the performance of the application. However, it has little impact in either templates generation time cost or size (see columns “Templates generation time cost” and “Total size of templates” in Table 6.3), since the differences between the strategies (few seconds and bytes) are considered negligible.

6.3.2 Aspect 2: Instrumentation of the application

Having the templates generated, developers would have to instrument the application to collect *bindings* conforming to these templates. To address this challenge, UML2PROV

Table 6.3 Information about PROV templates—number, size, number of variables, time cost—and time cost for generating the BGM per strategy.

Strategy (UML diagrams)	Num. templates	Total size of templates (MB)	Num. variables	Templates generation time cost (s)	BGM generation time cost (s)
Strategy 1 (SqD, SMD, CD)	198	0.30	954	3.94	6.77
Strategy 2 (SqD, CD)	1,821	1.48	5,980	8.98	10.31
Strategy 3 (CD)	1,688	1.30	5,236	8.10	9.72

advocates providing each application with a BGM that must fulfil a set of requirements. To support the development of these BGMs, our reference implementation of UML2PROV automatically generates a specific BGM from the UML design of each application, so that the instrumentation of the application boils down to deploy the generated BGM into the application at hand.

Nevertheless, the generated BGM, independently of its configuration, changes depending on the source UML design. As concluded from Section 6.3.1, the more UML elements making up the UML design, the more templates are generated (and variables in these templates). Typically, for each variable in a template, an operation call is needed to assign a value to it, thus, a developer would need to write one line of code for each variable in a template. Hence, the intuition is that the number of instructions included in the BGM grows as the number of UML elements does. The column “Num. variables” of Table 6.3 corresponds to the number of instructions for bindings generation contained in the BGM. The strategy with least UML elements (Strategy 1) leads to the BGM with fewest instructions (954), as opposed to Strategies 2 and 3 that generate BGMs with more instructions (5,980 and 5,236, respectively). Once again, the effort devoted to obtaining a more precise design according to the provenance requirements incurs in a simplification of the BGM, which has significant performance implications, as we will discuss later in Section 6.3.4.

Here, we remark that the above analysis is independent of the configuration of the BGM. All of our stated configurations are defined in such a way that only an operation call is needed to assign a value to a variable from a template (that is, to create a binding). However, these details about the internal implementation of the BGM are irrelevant to its usage, since one of the main advantages of BGMs’ requirements is that they ensure that software engineers do not need to write any line of code. Thus, the complexity of the BGM is an irrelevant aspect for them. Conversely, potential users that implement their own BGMs (or other mechanisms for generating bindings) must do it under their responsibility, since the number of instructions may grow out of control.

6.3.3 Aspect 3: Maintenance of provenance capabilities

The provenance intended to be generated by an application must describe what the application actually does. For this reason, when an application is redesigned, it is necessary to analyse if the PROV templates need to be changed [3]. In turn, the instrumented code may have to be updated in order to generate bindings conforming the updated templates. Without UML2PROV, these tasks would need to be performed by a software engineer with PROV skills, who eventually would have to modify the templates and the instrumented code to reflect the new design.

Moreau et al. [3] deal with some types of template evolution that are likely to occur in practice. For instance, a template may be *renamed*. Likewise, a new template may be *added*, when the application includes new functionality, or even *dropped*, in case of decommissioning a component. Another type of template evolution takes place when templates are *merged* or *split*. Additionally, the content of templates may be modified in several ways, e.g., *adding/removing a variable*, *adding/removing a node* or a *relation*, and so on. The analysis performed by Moreau et al. in [3] states that, whilst the bindings remain correct in most cases, some modifications to templates result in a partially generated provenance, or even in errors. For instance, when a template adds a new variable, bindings become partially incomplete because the new variable is not considered in the bindings. This fact will potentially result in a partially generated provenance. Against this background, UML2PROV entails huge benefits for the compatibility and synchronisation between the UML design and the application's provenance facets. Every time the UML design is updated, the templates and the BGM can be regenerated, which guarantees the immediate and automatic redesign of the application provenance facets. Additionally, the generation of these provenance facets is fully automatic, so that it does not require provenance skills.

Since templates are obtained from the UML design and bindings from the execution, another remarkable fact regarding this third aspect is the consistency between the application's UML design and implementation. It is important to note that the UML diagrams describe a part of the application, and therefore, its implementation must conform to such a UML design. However, although it is not a good practice, it is very common to have applications that do not strictly follow the design specified by the UML diagrams [192]. In these cases, users could leverage reverse-engineering to obtain the UML design according to the source code, as we have done with GelJ. In fact, users in this situation could apply the *design strategy* that best suits her/his needs.

6.3.4 Aspect 4: Run-time overhead and storage needs

As we mentioned previously in Section 3.4.3, a remarkable aspect when choosing a provenance system is the cost of capturing provenance, thus, it is desirable to minimise the influence of provenance collection on the actual application execution. Here, we analyse the run-time overhead and the storage needs, attributable to our reference implementation of UML2PROV for the instrumentation of GelJ under Strategies 1 to 3. Additionally, since the run-time overhead and storage needs depend on the way in which the BGM manages bindings, in this analysis we have also taken into account the three different BGM configurations provided in our reference implementation of Chapter 5 (*BindingsBGMEventListener*, *SetBindingsBGMEventListener*, and *ProvenanceBGMEventListener*). Recall that *Bindings-*

BGMEventListener stores each binding independently. *SetBindingsBGMEventListener*, for its part, creates sets of bindings and stores them. Finally, *ProvenanceBGMEventListener* generates the sets of bindings and additionally instantiates the PROV templates with them in order to obtain the final PROV documents. These final PROV documents are stored in the database.

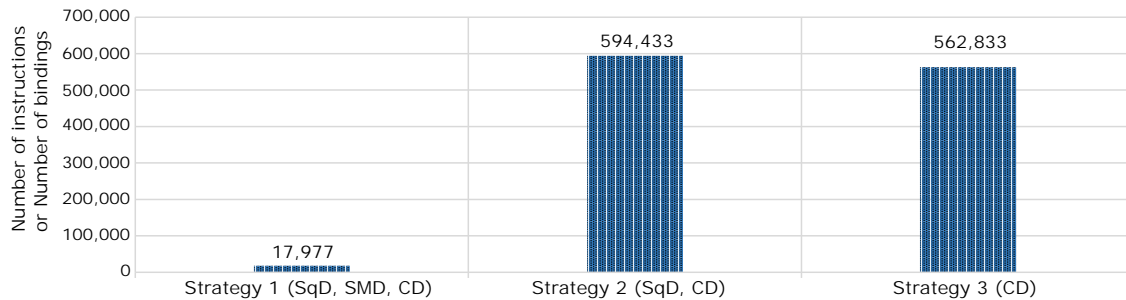


Fig. 6.2 Number of instructions executed for collecting provenance data organised by strategies. This number of instructions matches with the number of bindings collected.

This performance evaluation has been based on the sequence of interactions of the GelJ *experiment wizard* previously chosen (shown in Appendix C), and has been run on a personal computer, Intel(R) Core(TM) i7 CPU, 2.8GHz, with Oracle JDK1.8 and a Windows 10 Enterprise OS running MongoDB [177].

It is clear that the run-time overhead and storage needs associated with provenance depend on the execution of the instructions included for capturing it (see the differences between strategies in Figures 6.3 and 6.4, respectively). Hence, it is believed that the more instructions executed, the more run-time overhead and storage needs are required. As we see in Figure 6.2, Strategy 1 yields the least number of bindings (17,977) in comparison with Strategies 2 and 3 (594,433 and 562,833 respectively). Note that Strategy 1 captures less bindings since the collection of provenance is focused on satisfying a specific set of requirements. Conversely, Strategy 2 and 3 have higher results since they capture provenance for all the classes and methods derived from the whole UML diagrams produced by the reverse engineering tools, much more than the necessary to satisfy the application provenance requirements. Whilst this number of instructions/bindings has a direct impact on the final run-time overhead and storage needs (as we will see below), they can vary depending on the configuration of the BGM. Below, we will analyse how these two aspects are affected by (1) the three different UML designs yielded by the aforementioned design strategies (Strategies 1, 2 and 3), and (2) the three different configurations of the BGM (*BindingsBGMEventListener*, *SetBindingsBGMEventListener*, and *ProvenanceBGMEventListener*).

Run-time overhead Figure 6.3 depicts the run-time overhead (%) per strategy and configuration. In general terms, we can see that, as Strategy 1 generates the least number of bindings (17,977), it yields the best overheads regardless of the configuration (6.21% with *BindingsBGMEventListener*, 1.26% with *SetBindingsBGMEventListener*, and 1.53% with *ProvenanceBGMEventListener*). As for the implications each configuration has in the run-time overhead, *SetBindingsBGMEventListener* gives the best overheads, followed by the *ProvenanceBGMEventListener* and by the *BindingsBGMEventListener*, respectively. Concretely, *BindingsBGMEventListener* yields the higher results (6.21% in Strategy 1, 115.77% in Strategy 2, and 112.36% in Strategy 3) because it stores the bindings one by one, and therefore, it needs to execute persistence operations many times, which are time-consuming. Conversely, *ProvenanceBGMEventListener* stores the expanded PROV documents, which considerably reduces the number of time-consuming database operations and, consequently, the run-time overhead (1.53%, 68.67%, and 53.42%). Finally, *SetBindingsBGMEventListener* yields the best results (1.26%, 55.05%, and 48.45%) because it reduces the number of time-consuming database operations by storing sets of bindings rather than one by one (in contrast to *BindingsBGMEventListener*), and avoids expanding templates (as opposed to *ProvenanceBGMEventListener*).

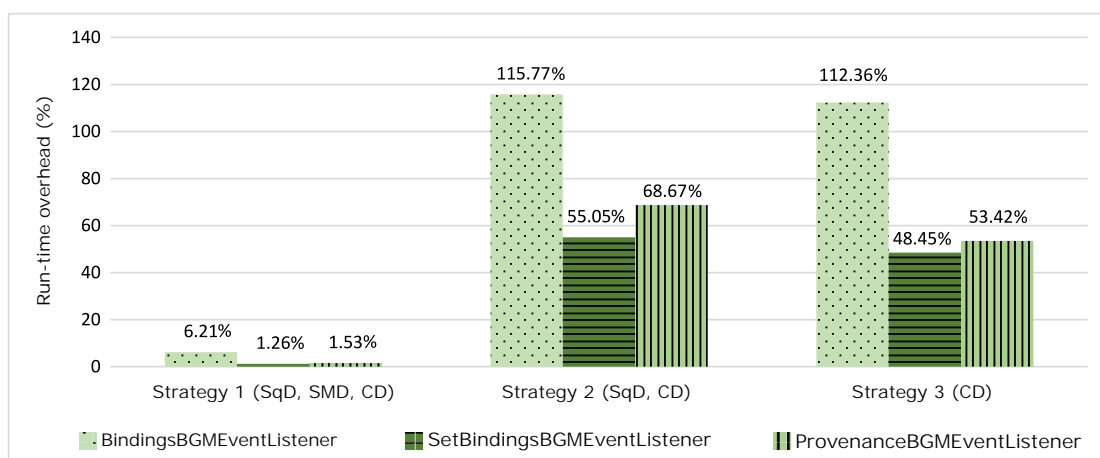


Fig. 6.3 Information about run-time overhead (%) for Strategies 1 to 3 recording provenance with configurations *BindingsBGMEventListener*, *SetBindingsBGMEventListener*, and *ProvenanceBGMEventListener*.

Storage overhead Figure 6.4 depicts the storage needs (MB) per strategy and configuration. Since Strategy 1 generates the least number of bindings, it requires the least storage regardless of the configuration (3.5MB with *BindingsBGMEventListener*, 2.3MB with *SetBindingsBGMEventListener*, and 3.6MB with *ProvenanceBGMEventListener*). As for the implications of each configuration in terms of storage needs, *SetBindingsBGMEventListener*

leads to the most compact storage in all the strategies (2.3MB in Strategy 1, 64.9MB in Strategy 2, and 57.2MB in Strategy 3). This is due to two main facts. First, each binding collected by means of the *BindingsBGMEEventListener* is stored with additional information needed to create the sets of bindings for expanding the templates (e.g., a unique identifier of the operation from which it was recorded, its associated templates, and so on). This fact leads to store a lot of recurrent information. With *SetBindingsBGMEEventListener*, the sets of bindings (i.e., not each binding) are associated with the information needed for expanding the templates, thereby reducing the recurrent information stored in the database. Second, as opposed to *ProvenanceBGMEEventListener*, *SetBindingsBGMEEventListener* stores sets of bindings that contain no topological information; thus, *SetBindingsBGMEEventListener* results in a more compact storage than *ProvenanceBGMEEventListener*.

All in all, we can say that a whole UML design produced by using reverse-engineering techniques provokes much more provenance than the necessary to satisfy the application provenance requirements, and therefore, it increases the run-time overhead and storage needs. On the contrary, the effort devoted to obtaining a UML design focused on specific provenance requirements (which means the capture of the provenance necessary to satisfy those requirements) results in a run-time overhead reduction and a compact storage. Additionally, we have also seen that using a configuration for recording bindings that delays the expansion of templates until the application finishes and stores sets of bindings (rather than bindings independently) incurs in less run-time overhead and more compact storage.

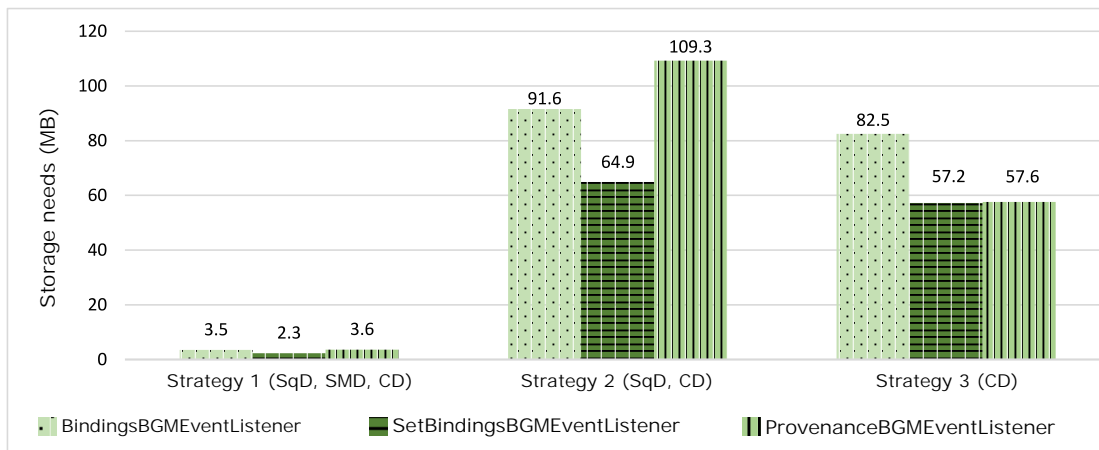


Fig. 6.4 Information about storage needs (MB) for Strategies 1 to 3 recording provenance with configurations *BindingsBGMEEventListener*, *SetBindingsBGMEEventListener*, and *ProvenanceBGMEEventListener*

6.3.5 Aspect 5: Quality of provenance

The provenance generated with each strategy is different since each strategy produces UML designs that expose different levels of detail about the application. Taking this into account, we have considered of relevance to analyse how the level of detail affects to the quality of the obtained provenance. With this aim, we have studied if the collected provenance can answer completely, partially, or if it cannot answer at all the questions in Table 6.2. Our conclusions are summarised in Table 6.4, where we also shown, in those cases in which the provenance can give an answer, the number of elements (`prov:Entity`, `prov:Activity`, `prov:Agent`) involved in such an answer. This number of elements has helped us identify three kinds of implications the strategy used to obtain the UML design may have on the ability to produce provenance answers: *no effect*, when the followed strategy does not affect the results; *more detailed information*, when the retrieved answer gives a different level of detail depending on the strategy; and *crucial*, when concrete information included in the UML diagrams is crucial for responding to the question. Next, we explain these implications in detail.

Table 6.4 For each strategy, it is indicated whether questions Q1-Q9 of Table 6.2 can be answered completely (Y), partially (P) or cannot be answered (N). If a question can be answered, the number of elements of the provenance involved in its answer appears in brackets.

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
St. 1	Y (5)	Y (206)	Y (3)	Y (152)	Y (12)	Y (3)	Y (10)	Y (198)	Y (234)
St. 2	Y (7)	Y (386)	Y (3)	P (35)	Y (12)	N	Y (10)	P (155)	Y (483)
St. 3	N	Y (386)	Y (3)	P (35)	Y (12)	N	Y (10)	N	Y (489)

No effect In order to answer question Q3, it is necessary to get the value of an attribute belonging to classes *StepN_M*, whereas to answer questions Q5 and Q7 we need to identify the execution of certain operations from classes *Step2_2* and *Step4_3*, respectively. The provenance from the three strategies answers the three questions because of different reasons. On the one hand, provenance from Strategy 1 answers the three questions due to the fact that the involved classes and operations were identified in the *provenance analysis* performed in such a strategy. On the other hand, provenance from Strategies 2 and 3 answers the questions because these strategies include in their class diagrams all the classes and operations of GelJ. Consequently, we can conclude that the strategy followed to obtain the UML design does not influence the ability to answer questions Q3, Q5 and Q7.

More detailed information A detailed UML design may lead to the generation of more provenance, obtaining more detailed answers. For instance, to answer question Q1 is crucial

to have information referred to nested operation calls. Since nested operations are specified by UML Sequence diagrams, the Strategy 2, which has the most detailed SqDs, gives a more detailed answer to Q1. Likewise, to answer Q2 and Q9 it is necessary to get information about operations identified in classes *StepN_M*. Thus, the provenance from Strategies 2 and 3, which identify all the operations in classes *StepN_M*, gives a more detailed information than Strategy 1. This is because the provenance from Strategy 1 only contains information about the operations of classes *StepN_M* identified in the *provenance analysis*. Whilst, in principle, giving answers with a higher level of detail seems to be a valuable fact, it is worth considering the extra time and storage that this implies (see Section 6.3.4). Taking into account that Strategy 1 has been designed precisely to expose only the information necessary for responding to the identified questions, the additional provenance elements in Strategies 2 and 3 could be considered unnecessary, rather than a good characteristic of the result.

Crucial There are some aspects of an application that are modeled only by certain types of UML diagrams, even by certain elements of those diagrams. For example, to answer question Q6, it is necessary to have information provided by SMDs. Thus, due to the fact that Strategy 1 is the only one that considers SMDs, it is the only strategy that answers Q6. Similarly, in order to answer Q8, we need information exposed by SqDs and SMDs. This fact explains (1) why Strategy 2, which has SqDs but lacks SMDs, can only partially answer; and (2) why Strategy 3, which lacks SqDs and SMDs, cannot even give an answer. Another example of relevant UML elements that help respond to some kind of answers are operations' stereotypes, which explain the nuances of class operations, thus helping to capture a more meaningful provenance. These nuances are crucial to answer Q4. Hence, Strategy 1, which is the only one that has stereotyped operations, can completely respond to Q4, while Strategies 2 and 3 only partially answer it. These examples delve into the arguments in favour of the convenience of making a prior investment of time to conduct a detailed UML design of the application, guided by a set of provenance requirements.

6.4 Conclusions and discussion

Throughout this chapter, we have presented an evaluation of UML2PROV that uses quantitative and qualitative arguments to show the benefits and trade-offs of applying UML2PROV. To do this, we have applied UML2PROV to a Bioinformatics application called GelJ. More specifically, we have applied our reference implementation of UML2PROV to three different UML designs of GelJ, which contain different levels of detail. Taking such UML designs as source of UML2PROV has allowed us to evaluate the implications of the UML design over a

set of aspects with regard to provenance such as *generation of the provenance design, instrumentation of the application, maintenance of provenance capabilities, run-time overhead and storage needs, and quality of provenance*.

All in all, we have seen that UML2PROV links two key artefacts for generating provenance (PROV templates and BGM) with the actual design of an application, requiring only few seconds to generate them. Likewise, with UML2PROV, the instrumentation of an application is transparent to software engineers since it only consists in deploying the BGM into the application at hand. These two aspects incur in great benefits with regard to the maintenance of the provenance capabilities. Software engineers do not need to devote time making the PROV templates and instrumenting applications to generate bindings and, what is more, they do not even need to have knowledge about provenance. Something similar happens with the maintenance of the provenance capabilities, every time the UML design is updated, the templates and the BGM can be regenerated immediately. Thus, it is completely transparent to software engineers. Finally, based on this evaluation, we have reached two main conclusions regarding UML2PROV and its associated provenance costs (run-time overhead and storage needs). First, using a detailed UML design incurs in the capture of more provenance than required to satisfy provenance requirements. Alternatively, whether the UML designer focuses on addressing provenance requirements, only relevant provenance is collected, resulting in low run-time overhead and storage needs (as low as $\sim 1.26\%$ of in execution time and 2.3MB for storing provenance with the configuration *SetBindingsBGMEventListener*). Second, the configuration of the BGM has a high impact in the provenance costs. Concretely, we have seen that the number of operations against the database is crucial in increasing the costs. The largest number of operations against the database, the highest the run-time overhead and storage needs. Thus, to group bindings and to store set of bindings reduces considerably the costs of provenance.

To conclude, there is an important aspect we would like to discuss. Whilst we have carried out each design strategy relying on the whole source code of GelJ, a potential software engineer may be interested in either focusing on a concrete application's module (unlike the whole source code) or combining strategies so as to collect different provenance information depending on the module (as opposed to using only one strategy). For example, a potential software engineer may be interested in obtaining provenance from the whole application execution, but she/he has concrete provenance requirements with regard to a specific module. In this situation, it is not needed to apply Strategy 1 to the whole application, which requires a great manual effort. The software engineer should apply Strategy 1 only to the module with provenance requirements, thereby aiming at capturing provenance to meet the requirements, and to apply another strategy, which does not require manual effort, to the remaining modules.

This procedure has two main benefits for the software engineer. First, she/he can fulfil their provenance requirements, since part of the collected provenance is focused on them. Second, she/he does not need to devote an extra effort to apply Strategy 1 to the whole application, since Strategy 1 is only applied to the module of interest. Similarly, whether a software engineer is only interested in provenance regarding a specific module, she/he only needs to apply one strategy to such a module. The strategy chosen by the software engineer depends on her/his needs, such as if she/he has provenance requirements, or the level of effort she/he is willing to do for collecting provenance. In any case, software engineers can focus on the module of interest, avoiding the collection of provenance from the rest of the application.

Chapter 7

Conclusions and future work

We started the writing of this thesis aiming at developing a framework for making applications provenance-aware. A framework that took as a starting point the UML design of an application, and that ultimately generated the suitable artefacts to capture provenance data about its execution. In addition, our claim was for the developed framework to be minimally invasive both for software designers, responsible for designing the provenance to be captured, as well as for developers, who had to write the code to instrument the application. So much so, that we advocated that both participants in the development of the application do not need to be knowledgeable of provenance.

Thus, UML2PROV was born. And now the time has come to recap and analyze what has been achieved, and what are the aspects that we shall have to do further work on.

7.1 Summary of results

We believe we can say that the goal has been met. As we already advanced in the introduction of this thesis, there have been three main contributions that have led this research work to its intended goal. Let's start by mentioning our review of the literature, conducted as a systematic review, which, besides helping motivate UML2PROV, constitutes a contribution in itself. Later we proposed a conceptual definition for UML2PROV, which established the conceptual aspects for bridging the gap between application design and provenance design. Finally, a UML2PROV reference implementation was provided, which fully automatises the generation of provenance by following the stated definition of UML2PROV.

Let us now revisit how we have achieved these contributions.

7.1.1 A systematic review of provenance systems

Aimed at identifying open problems in the field of provenance that may serve as motivation for our main goal, we have performed a thorough review of provenance systems. This systematic literature review identifies a comprehensive set of 105 relevant published works in the provenance field from more than 500 analysed works. Based on these selected works, we have defined a six-dimensional taxonomy of provenance systems characteristics attending to: general aspects, data capture, data access, subject, storage, and non-functional aspects. This taxonomy encompasses common characteristics of provenance systems thoroughly renowned through-out the literature on the topic. Additionally, based on this taxonomy, we have performed an exhaustive analysis and comparison of 25 systems, which illustrated different solutions covering a broad spectrum of alternatives along the aspects identified in our taxonomy. In this thesis, we have given a concise version of this systematic review (published in [10]). Concretely, we have presented an overview of the taxonomy of provenance systems keeping in mind two main aspects with regard to the main goal of this thesis.

First, to present a consistent background of provenance concepts and techniques that have allowed potential readers to grasp the performed work. Our taxonomy encompasses the overall classifications considered by other works, and more fully captures the unique characteristics of provenance systems. Thus, it has familiarised the reader with the terminology used in the field of provenance, in general, and in this thesis, in particular.

Second, to identify the uncovered open problems that have served as foundations for our main goal in mind. Concretely, the analysis and comparison of the provenance systems have given us an idea about the current available approaches and technologies, and additionally, it has also allowed us to discover the open research problems that motivated this thesis. More precisely, we have identified four open problems: integration, interoperability, computational overhead, and querying.

7.1.2 A conceptual definition of UML2PROV

We have established the conceptual definition of UML2PROV by means of an architecture that defines two key elements: PROV templates and BGM. They are obtained from the UML diagrams of an application.

To address the creation of PROV templates from UML diagrams, we have defined a comprehensive, extensive, and systematic set of 17 transformation patterns that ultimately associates UML elements with PROV elements of a template. Concretely, 4 of these patterns address Sequence diagrams and 3 patterns address State Machine diagrams. We have chosen these types of diagrams because they are widely used to represent the behaviour of a system,

which is one of the main purposes of capturing provenance information. Additionally, 10 of these patterns address Class diagrams. We have decided to consider this type of diagrams since it is the most widely adopted formalism for modeling the intentional structure of a system, giving low level aspects from objects' internal status, information not given by the Sequence and State Machine diagrams. With these three types of UML diagrams, we can obtain provenance from three different perspectives related to the types of UML diagrams. Concretely, the provenance obtained from SqDs reflects how collaborating objects interact for executing operations, and the exchange of information between them. The provenance obtained from CDs contains information about (1) the objects' characteristics at some point, i.e. the object's status, and (2) the operations that have led the objects' status to be as they are. Finally, the provenance coming from SMDs shows information about the evolution of the objects' state as a consequence of operations executions taken place. Aiming at easing the use of these patterns, we have presented a set of charts that guides users through concrete features of the patterns, in order to help them find the pattern that best suits their needs. What is more, this set of charts also permits readers to see at a glance all the contexts addressed by the patterns, as well as to compare the patterns easily.

As for the creation of the BGM for each application, we have defined a set of requirements that each BGM must fulfil so as to minimise the intrusion on software designers' and developers' modus operandi, as well as to facilitate the maintenance of the provenance-aware applications. To do this, we have leveraged the performed systematic review of the approaches for capturing data in order to define a set of features that, from our point of view, the BGMs should have. Later, relying on these features, we have defined a set of requirements that the BGM for an application must meet to satisfy such features. Generally speaking, with this set of requirements we have ensured that the generation of bindings boils down to deploy a BGM into an application, without requiring from developers to instrument such an application.

All in all, the transformation patterns, together with the BGM's requirements, define a conceptual background over which an implementation of UML2PROV should rely on.

7.1.3 An implementation of UML2PROV

In order to verify the feasibility of the conceptual definition of UML2PROV, as well as to validate if a potential implementation of it were possible, our starting point was to develop a proof of concept implementation of UML2PROV. Subsequently, based on the lessons learned from this proof of concept, we developed a reference implementation of UML2PROV. This reference implementation provides a complete development of the conceptual definition of

UML2PROV to automatise the process for generating templates and BGM from the UML design of an application.

As for the generation of templates from the UML design, the reference implementation of UML2PROV leverages ATL, to perform M2M transformations, and XPand, to carry out M2T transformations. This implementation consists of two stages. First, an ATL module, which implements the transformation patterns, takes the UML diagram models and automatically generates PROV template models, which are serialised in a PROV independent format (XMI). Second, an XPand module takes the previous PROV template models and automatically generates the PROV template files in PROV-N format. These two stages allowed us to distinguish between the implementation of the transformation patterns (which is agnostic about any PROV serialization format) and the serialization format of the final PROV templates files. In this way, the implementation of the transformation patterns in ATL can be seen as a generic solution regardless of the final PROV template format, since the XPand module defines the serialization format of the final PROV template files (PROV-N in our case). Thus, potential users who are interested in another serialization format could implement their own XPand module that transforms the PROV template models into such a format, without affecting the implementation of the transformation patterns.

Regarding the BGM, we have advocated using a generic event-based approach developed on top of AOP in order to provide a fully automatic solution to generate bindings. Based on the UML design of an application, our reference implementation of UML2PROV automatically generates a BGM for such an application. Each generated BGM meets the set of requirements stated in Section 4.4, so that the instrumentation of the application to generate bindings boils down to deploy the BGM into the application. More specifically, these generated BGMs ensure that the generation of bindings is fully automatic (R1), without affecting the design and maintenance of the application's source code (R2 and R3), and providing mechanisms to configure the capture of provenance (R4). Additionally, these BGMs also ensures that the generated bindings are consistent with the previously obtained PROV templates (R5 and R6).

At this point, we would like to note that our reference implementation for generating templates could be seen as a generic solution suitable for being used by any final user of UML2PROV 'as it is', without any modification. This reference implementation gives a complete automatic translation of any UML design into PROV templates as it is stated by our patterns. Unlike the generation of templates, our reference implementation for generating the BGM of an application is limited to Java applications. Nevertheless, any potential developer can implement her/his own BGM relying on the generic structure we have defined for BGMs. The unique requirement for leveraging this structure is to have an AOP implementation compatible with the programming language of the target application. We do not consider

this requirement as an obstacle because, since the initial release of the AOP concept, a large number of implementations for various programming languages have emerged (such as Python, C, Ada, JavaScript, C#, and so on). In any case, whatever developer may implement her/his own mechanisms for generating bindings under her/his responsibility. For example, Johnson et al. in [170] used UML2PROV for obtaining PROV templates from their UML design, but, since their application was not developed in Java, they implemented their own mechanisms to generate bindings.

Finally, we would like to note that we have performed a systematic evaluation that showed the benefits and trade-offs of applying UML2PROV for software engineers looking to make applications provenance-aware. This evaluation, based on different UML designs of a legacy application, yielded two key lessons that are worth mentioning. First, taking as source a detailed UML design leads to the capture of more provenance than required to fulfil provenance requirements and thus, unnecessarily increases the run-time overhead and storage needs. Otherwise, whether the UML design sticks to address provenance requirements, only appropriate provenance is collected, and consequently, the run-time overhead and storage needs decrease.

7.2 Limitation of the study and future work

Although we have successfully evaluated UML2PROV with different UML designs of a legacy application, obtaining encouraging results, we recognise there are certain limitations to the presented work. Here, we touch upon six issues, which provide a basis for the extension of this research.

In our proposal, the PROV templates obtained from the application UML design define what information is included in the final provenance; thus, they define the granularity of the provenance to be generated. More specifically, when we defined these transformation patterns, we made the decision of considering a high level of detail in the flow of logic within the system (SqDs), in the objects' status (CDs), or in the change of objects' states (SMDs). Thus, our proposal generates fine-grained provenance. Future research could examine how to manage the level of detail of the provenance to be generated (i.e., its *granularity*). An alternative to do this could be to examine how to adapt the transformation patterns, by selectively discarding some PROV elements or relations, and consequently, generating coarser-grained data provenance. In this line, we also consider of interest to provide the UML designer with a mechanism to specify the elements in the UML diagrams for provenance capture. In this way, she/he could manage the amount of provenance information to be generated. For instance, a potential UML designer could selectively avoid provenance

capture regarding specific input parameters of an operation, a fact that cannot be avoided with the current approach.

UML2PROV supports the generation of templates from UML Class diagrams, which is guided by operations associated with stereotypes from our taxonomy of operations. Thus, before the transformation process, designers have to traverse the whole set of operations identifying the stereotype that represents the operation's behaviour. In order to prevent designers from performing this task, an interesting future line of work is to reverse engineer the operation stereotype from the UML operation's signature. For instance, a possible starting point could be operations with the stereotype `«get»`, `«set»` or `«search»`. It is well-known that an operation that directly returns a value of a concrete attribute of an object (denoted by the stereotype `«get»`) is named with the prefix *get* followed by the name of the returned attribute. Something similar occurs with stereotype `«set»` and operations which start with *set*, as well as stereotype *search* and operations which start with *find*. To do this, we have thought in enriching the implementation of the transformations from UML Class diagrams into PROV templates to support reverse engineering operation stereotypes from the UML operation signature.

Although our proposal takes into account three of the most used UML diagrams, considering a wider number of UML elements. Nevertheless, we consider that to include other kinds of UML Diagrams (such as UML Activity Diagrams), and other elements (such as State Machine Diagram's effects and internal activities, not considered in our patterns) could constitute a more complete proposal. Thus, this is an interesting line for future research to explore.

Another line of future work that we consider of interest is the application of UML2PROV in a system comprising distributed components. We plan to tackle this goal by automatically generating a BGM for each component of the distributed system, and implementing the provenance exchange among the BGMs. To address the provenance exchange, one option could be to explore the PROV-AQ specification [193]. This document specifies how to use standard Web protocols (such as HTTP) to obtain information about the provenance of resources on the web. It would constitute our mechanism for exchanging data among distributed systems.

In the provenance field, there is a well-known problem called *n-by-m* [45]. This problem refers to those situations in which may not be possible to accurately determine the dependencies between input and output. For example, an operation with the stereotype `«command»` changes the status of an object as a whole, sometimes relying upon input data. In this situation, it is not possible to infer which inputs affected the change of the object's status, so the provenance graph has to link the resulting object, in a concrete status, with all of the inputs.

In our context, this problem can be solved by adding more semantics to UML operations by means of new stereotypes. For instance, we could define a stereotype associated with UML operation's parameters to mark those parameters involved in the generation of outputs.

Finally, we would like to state two more lines of further work regarding the implementation of UML2PROV, which are focused on expanding the number of potential users. First, we believe that providing UML2PROV as a web service could incur in large benefits for potential users. Users could upload their UML diagrams models and they could automatically obtain both the templates and the BGM. Second, we consider that the generation of the BGM only for Java applications limits the number of potential users; thus, a possible line of future work is to offer support to other programming languages. Concretely, we have thought in scripting programming languages as a good starting point to expand our approach, since this kind of languages is widely used for implementing scientific applications.

Conclusiones

Al comenzar la redacción de esta tesis se planteó como objetivo general la creación de un marco de desarrollo para hacer aplicaciones con capacidades de *provenance*, que tomase como punto de partida el diseño software en UML de la aplicación y que generase los artefactos necesarios para obtener información *provenance* sobre la ejecución de la misma. Además, nuestra pretensión era que el framework desarrollado fuese mínimamente invasivo tanto para los diseñadores software encargados del diseño de *provenance* a capturar, como para los desarrolladores de la aplicación. Tanto era así, que ambos partícipes del desarrollo de la aplicación no necesitasen ser expertos en *provenance*.

Así surgió UML2PROV. Y ahora ha llegado el momento de recapitular y analizar qué es lo que se ha conseguido.

Como ya hemos avanzado en la introducción de esta tesis, tres han sido las contribuciones principales que han llevado a este trabajo de investigación a su meta pretendida. Comencemos mencionando nuestra revisión de la literatura, realizada como una revisión sistemática que, además de motivar UML2PROV, constituye una contribución en sí misma. Más tarde, propusimos una definición conceptual para UML2PROV, que estableció los aspectos conceptuales para salvar la brecha entre el diseño de la aplicación y el diseño del *provenance*. Por último, proporcionamos una implementación de referencia de UML2PROV, que automatiza completamente la generación de *provenance* siguiendo la definición establecida de UML2PROV.

Finalmente, nos gustaría señalar que hemos realizado una evaluación sistemática que mostró los beneficios e inconvenientes de UML2PROV para aquellos ingenieros de software que buscan hacer que sus aplicaciones tengan capacidades de *provenance*. Esta evaluación, basada en diferentes diseños UML de una aplicación ya existente, arrojó dos lecciones clave que vale la pena mencionar. Primero, el utilizar un diseño detallado de UML conduce a la captura de más cantidad de *provenance* que el requerido para cumplir con los requisitos de *provenance* y, por lo tanto, aumenta innecesariamente las necesidades de almacenamiento y los gastos generales de tiempo de ejecución. De lo contrario, si el diseño UML se atiene a los requisitos de *provenance*, solo se recopila el *provenance* necesario y, en consecuencia,

el coste que supone disponer de dichos artefactos en tiempo de ejecución, así como, las necesidades de almacenamiento disminuyen.

References

- [1] OMG. Unified Modeling Language (UML). Version 2.5, 2015. Document formal/15-03-01, March, 2015.
- [2] Paul Groth and Luc Moreau (eds.). PROV-Overview. An Overview of the PROV Family of Documents. W3C Working Group Note NOTE-prov-overview-20130430, World Wide Web Consortium, April 2013.
- [3] Luc Moreau, Belfrit Victor Batlajery, Trung Dong Huynh, Danius Michaelides, and Heather Packer. A templating system to generate provenance. *IEEE Transactions on Software Engineering*, 44(2):103–121, 2018.
- [4] Anneke G Kleppe, Jos Warmer, Jos B Warmer, and Wim Bast. *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Professional, 2003.
- [5] Luc Moreau and Paolo Missier (eds.). PROV-N: The Provenance Notation. W3C Recommendation REC-prov-n-20130430, World Wide Web Consortium, April 2013.
- [6] Natalia Dragan, Michael L. Collard, and Jonathan I. Maletic. Automatic identification of class stereotypes. In *Proceedings of the 26th IEEE International Conference on Software Maintenance*, pages 1–10, 2010.
- [7] Statement on Algorithmic Transparency and Accountability. Available at https://www.acm.org/binaries/content/assets/public-policy/2017_joint_statement_algorithms.pdf. Last visited on July 2019.
- [8] Donald Ervin Knuth. *The Art of Computer Programming: Fundamental Algorithms. Fundamental Algorithms*. Addison-Wesley, 1997.
- [9] BBC News. Microsoft chatbot is taught to swear on Twitter. Available at <https://www.bbc.com/news/technology-35890188>. Last visited on July 2019.
- [10] Beatriz Pérez, Carlos Sáenz-Adán, and Julio Rubio. A systematic review of provenance systems. *Knowl. Inf. Syst.*, 2018.
- [11] Simon Miles, Paul T. Groth, Steve Munroe, and Luc Moreau. PrIME: A methodology for developing provenance-aware applications. *ACM Trans. Softw. Eng. Methodol.*, 20(3):8:1–8:42, 2011.
- [12] Provenance Challenge Series. Available at: <http://twiki.ipaw.info/bin/view/Challenge/>. Last visited on July 2019.

- [13] The Open Provenance Model v1.01. Available at: <http://eprints.soton.ac.uk/266148/1/opm-v1.01.pdf>. Last visited on July 2019.
- [14] Luc Moreau, Paul Groth, James Cheney, Timothy Lebo, and Simon Miles. The rationale of prov. *Web Semantics: Science, Services and Agents on the World Wide Web*, 35:235–257, 2015.
- [15] ProvPy. Python implementation of the PROV data model. Available at <https://pypi.org/project/provpy/>. Last visited on July 2019.
- [16] ProvToolbox. Available at lucmoreau.github.io/ProvToolbox/. Last visited on July 2019.
- [17] Gianna Reggio, Maurizio Leotta, Filippo Ricca, and Diego Clerissi. What are the used UML diagrams? A preliminary survey. In *In EESSMOD@MoDELS, USA*, pages 3–12, October 2013.
- [18] Martina Seidl, Marion Brandsteidl, Christian Huemer, and Gerti Kappel. *UML@Classroom*. Springer, 2012.
- [19] Sinan Si Alhir. *UML in a nutshell: a desktop quick reference*. " O'Reilly Media, Inc.", 1998.
- [20] OMG. Analysis and Design Platform Task Force. White Paper on the Profile Mechanism, Version 1.0. OMG Document ad/99-04-07. Available at <http://www.omg.org/>. Last visited on July 2019.
- [21] Luc Moreau and Paolo Missier (eds.). PROV-DM: The PROV Data Model. W3C Recommendation REC-prov-dm-20130430, World Wide Web Consortium, 2013.
- [22] Timothy Lebo, Satya Sahoo, Deborah McGuinness, Khalid Belhajjame, James Cheney, David Corsar, Daniel Garijo, Stian Soiland-Reyes, Stephan Zednik, and Jun Zhao. PROV-O: The PROV Ontology. W3C Recommendation REC-prov-o-20130430, World Wide Web Consortium, 2013.
- [23] Hook Hua, Curt Tilmes, and Stephan Zednik (eds.). PROV-XML: The PROV XML Schema. W3C Note, World Wide Web Consortium, 2013.
- [24] James Cheney (ed.). Semantics of the PROV Data Model. W3C Note, World Wide Web Consortium, 2013.
- [25] James Cheney, Paolo Missier, and Luc Moreau (eds.). Constraints of the PROV Data Model. W3C Recommendation, World Wide Web Consortium, 2013.
- [26] Daniel Garijo and Kai Eckert (eds.). PROV-Constraints: Constraints of the PROV Data Model. W3C Recommendation, World Wide Web Consortium, 2013.
- [27] M. Dürst and M. Suignard. Internationalized Resource Identifiers (IRIs) (RFC 3987). Available at <http://www.ietf.org/rfc/rfc3987.txt>. Last visited on July 2019.
- [28] Danus Michaelides, Trung Dong Huynh, and Luc Moreau. PROV-TEMPLATE: A Template System for PROV Documents. Jun 2014, Technical Note. Available at provenance.ecs.soton.ac.uk/prov-template-2014-06-07/. Last visited on July 2019.

- [29] PROV Graph Layout Conventions. Available at www.w3.org/2011/prov/wiki/Diagrams. Last accessed on July 2019.
- [30] JSON-LD 1.0. A JSON-based Serialization for Linked Data. January, 2014. Available at <https://www.w3.org/TR/2014/REC-json-ld-20140116/>. Last visited on July 2019.
- [31] Bran Selic. The pragmatics of model-driven development. *IEEE software*, 20(5):19–25, 2003.
- [32] OMG. OMG Model Driven Architecture. Available at <https://www.omg.org/mda/>. Last visited on July 2019.
- [33] ATL - a model transformation technology, version 3.8. Available at www.eclipse.org/atl/. Last visited on July 2019.
- [34] XSL Transformations (XSLT) Version 3.0. W3C Candidate Recommendation 7 February 2017. Available at www.w3.org/TR/xslt-30/. Last visited on July 2019.
- [35] XPand. Eclipse platform. Available at wiki.eclipse.org/Xpand. Last visited on July 2019.
- [36] XML Path Language (XPath) 3.1 . W3C Recommendation 21 March 2017 . Available at <https://www.w3.org/TR/xpath-31/>. Last visited on July 2019.
- [37] ATL Integrated Environment. Available at <https://projects.eclipse.org/projects/modeling.mmt.atl>. Last visited on July 2019.
- [38] XPand plugins for Eclipse platform. Available at <https://www.eclipse.org/modeling/m2t/downloads/?project=xpand>. Last visited on July 2019.
- [39] Ramnivas Laddad. *Aspectj in action: enterprise AOP with spring applications*. Manning Publications Co., 2009.
- [40] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'97)*, pages 220–242, Berlin, Heidelberg, 1997.
- [41] The AspectJ Project. Available at www.eclipse.org/aspectj/. Last visited on July 2019.
- [42] The Ajaxpect library. Available at <https://github.com/deepcode/ajaxpect>. Last visited on July 2019.
- [43] The Aspects module for Python. Available at <https://pypi.org/project/aspects/>. Last visited on July 2019.
- [44] Carlos Saénz-Adán, Francisco J. García-Izquierdo, Ángel Luis Rubio, Eduardo Sáenz de Cabezón Irigaray, Emilio Rodríguez-Priego, and Oscar Díaz. A Tool for Management of Knowledge Dispersed throughout Multiple References. In *Proceedings of the 10th International Conference on Software Paradigm Trends (ICSOFT-PT'15)*, pages 79–86, 2015.

- [45] Lucian Carata, Sherif Akoush, Nikilesh Balakrishnan, Thomas Bytheway, Ripduman Sohan, Margo Seltzer, and Andy Hopper. A primer on provenance. *Commun. ACM*, 57(5):52–60, 2014.
- [46] Susan B. Davidson and Juliana Freire. Provenance and scientific workflows: challenges and opportunities. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (MOD’08)*, pages 1345–1350, New York, NY, USA, 2008. ACM.
- [47] Juliana Freire, David Koop, Emanuele Santos, and Cláudio T. Silva. Provenance for Computational Tasks: A Survey. *Comput. Sci. Eng.*, 10(3):11–21, May 2008.
- [48] Susan B. Davidson, Sarah Cohen-Boulakia, Anat Eyal, Bertram Ludäscher, Timothy M. McPhillips, Shawn Bowers, Manish Kumar Anand, and Juliana Freire. Provenance in Scientific Workflow Systems. *IEEE Data Eng. Bull.*, 30(4):44–50, 2007.
- [49] Yogesh L. Simmhan, Beth Plale, and Dennis Gannon. A Survey of Data Provenance in e-Science. *SIGMOD Rec.*, 34(3):31–36, 2005.
- [50] Sérgio Manuel Serra da Cruz, María Luiza Machado Campos, and Marta Mattoso. Towards a Taxonomy of Provenance in Scientific Workflow Management Systems. In *Proceedings of the IEEE Congress on Services, Part I, SERVICES I*, pages 259–266, 2009.
- [51] Boris Glavic and Klaus R. Dittrich. Data Provenance: A Categorization of Existing Approaches. In *Proceedings of Datenbanksysteme in Business, Technologie und Web (BTW’07)*, pages 227–241, 2007.
- [52] Wang Chiew Tan. Provenance in Databases: Past, Current, and Future. *IEEE Data Eng. Bull.*, 30(4):3–12, 2007.
- [53] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. Why and Where: A Characterization of Data Provenance. In *Proceedings of the 8th Intl. Conf. on Database Theory (ICDT’01)*, pages 316–330, 2001.
- [54] Peter Buneman and Wang Chiew Tan. Provenance in Databases. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (MOD’07)*, pages 1171–1173. ACM, 2007.
- [55] James Cheney, Laura Chiticariu, and Wang Chiew Tan. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases*, 1(4):379–474, April 2009.
- [56] Luc Moreau. The Foundations for Provenance on the Web. *Foundations and Trends in Web Science*, 2(2–3):99–241, 2010.
- [57] P. Buneman and S. B. Davidson . Data provenance – the foundation of data quality. Technical report, A Technical Report, September. Available at <https://pdfs.semanticscholar.org/9ec4/275fed43df7145dec34cba9743a9186dc972.pdf>. Last visited on July 2019.

- [58] Rajendra Bose and James Frew. Lineage retrieval for scientific data processing: a survey. *ACM Comput. Surv.*, 37(1):1–28, 2005.
- [59] Yogesh L. Simmhan, Beth Plale, and Dennis Gannon. A Survey of Data Provenance Techniques. *Computer Science Department, Indiana University, Bloomington IN*, (612), 2005. Extended version of SIGMOD Record 2005. Available at: www.cs.indiana.edu/pub/techreports/TR618.pdf.
- [60] Alfredo Cuzzocrea. Big data provenance: State-of-the-art analysis and emerging research challenges. In *Proceedings of the Workshops of the EDBT/ICDT*, pages 797–800, 2016.
- [61] Gulustan Dogan. A survey of provenance in wireless sensor networks. *Ad Hoc & Sensor Wireless Networks*, 30(1-2):21–45, 2016.
- [62] Yu Shyang Tan, Ryan K. L. Ko, and Geoff Holmes. Security and data accountability in distributed systems: A provenance survey. In *Proceedings of the IEEE 10th International Conference on High Performance Computing and Communications (HPCC'13)*, pages 1571–1578. IEEE Computer Society, 2013.
- [63] Changda Wang, Wen-yi Zheng, and Elisa Bertino. Provenance for wireless sensor networks: A survey. *Data Science and Engineering*, 1(3):189–200, 2016.
- [64] B.A. Kitchenham. Procedures for Performing Systematic Reviews. *Keele, UK, Keele University*, 2004. At: <http://www.inf.ufsc.br/aldo.vw/kitchenham.pdf>. Last visited on July 2019.
- [65] B. Kitchenham and S. Charters. Guidelines for performing Systematic Literature Reviews in Software Engineering. Technical Report EBSE 2007–01, Technical report, Ver. 2.3 EBSE Technical Report. EBSE. At: <http://pages.cpsc.ucalgary.ca/~sillito/cpsc-601.23/readings/kitchenham-2007.pdf>. Last visited on July 2019.
- [66] B. Kitchenham, O. Pearl Brereton, D. Budgen, et al. Systematic Literature Reviews in Software Engineering - A Systematic Literature Review. *Inf. Softw. Technol.*, 51(1):7–15, January 2009.
- [67] Louis Bavoil, Steven P Callahan, Patricia J Crossno, et al. Vistrails: Enabling interactive multiple-view visualizations. In *Proceedings of the IEEE Visualization (VIS'05)*, pages 135–142. IEEE, 2005.
- [68] Juliana Freire, Cláudio T Silva, Steven P Callahan, et al. Managing rapidly-evolving scientific workflows. In *Proceedings of the International Provenance and Annotation Workshop (IPAW'06)*, pages 10–18. Springer, 2006.
- [69] David Gammack, Steve Scott, and Adriane P Chapman. Modelling provenance collection points and their impact on provenance graphs. In *Proceedings of the International Provenance and Annotation Workshop (IPAW'16)*, pages 146–157, 2016.
- [70] Fernando Chirigati, Juliana Freire, David Koop, and Cláudio Silva. Vistrails provenance traces for benchmarking. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, pages 323–324, 2013.

- [71] Carlos E Scheidegger, Huy T Vo, et al. Querying and re-using workflows with vistrails. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (MOD'08)*, pages 1251–1254, 2008.
- [72] P. Missier, Stian Soiland-Reyes, Stuart Owen, et al. Taverna, reloaded. In *Proceedings of the International conference on scientific and statistical database management (SSDBM'10)*, pages 471–481, 2010.
- [73] K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers, S. Owen, S. Soiland-Reyes, I. Dunlop, A. Nenadic, P. Fisher, et al. The Taverna workflow suite: designing and executing workflows of Web Services on the desktop, web or in the cloud. *Nucleic acids research*, pages 557–561, 2013.
- [74] Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew R. Pocock, Anil Wipat, and Peter Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 2004.
- [75] Pinar Alper, Khalid Belhajjame, and Carole A Goble. Static analysis of taverna workflows to predict provenance patterns. *Futur Gener Comput Syst*, 75:310–329, 2017.
- [76] Ilkay Altintas, Oscar Barney, and Efrat Jaeger-Frank. Provenance Collection Support in the Kepler Scientific Workflow System. In *Proceedings of the International Provenance and Annotation Workshop (IPAW'06)*, pages 118–132, 2006.
- [77] Daniel Crawl and Ilkay Altintas. A provenance-based fault tolerance mechanism for scientific workflows. In *Proceedings of the International Provenance and Annotation Workshop (IPAW'08)*, pages 152–159, 2008.
- [78] Kiran Kumar Muniswamy-Reddy, David A Holland, Uri Braun, and Margo I Seltzer. Provenance-Aware Storage Systems. In *USENIX Annual Technical Conference, General Track*, pages 43–56, 2006.
- [79] K. K. Muniswamy-Reddy, Uri Braun, David A. Holland, et al. Layering in Provenance Systems. In *USENIX Annual Technical Conference*, 2009.
- [80] David A Holland, Margo I Seltzer, Uri Braun, and Kiran-Kumar Muniswamy-Reddy. Passing the provenance challenge. *Concurrency and Computation: Practice and Experience*, 20(5):531–540, 2008.
- [81] Omar Benjelloun, Anish Das Sarma, Chris Hayworth, and Jennifer Widom. An Introduction to ULDBs and the Trio System. Technical Report 2006-7, Stanford InfoLab. Avail. at: <http://ilpubs.stanford.edu:8090/793/>. Last visited on July 2019.
- [82] Jennifer Widom. Trio: A System for Integrated Management of Data, Accuracy, and Lineage. Technical Report 2004-40, Stanford InfoLab. Available at: <http://ilpubs.stanford.edu:8090/658/>. Last visited on July 2019.
- [83] Parag Agrawal, Omar Benjelloun, Anish Das Sarma, et al. Trio: A System for Data, Uncertainty, and Lineage. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'06)*, pages 1151–1154, September 2006.

- [84] Michi Mutsuzaki, Martin Theobald, and et al. Trio-One: Layering Uncertainty and Lineage on a Conventional DBMS. In *Proceedings of the Conference on Innovative Data Systems (CIDR'07)*, pages 269–274, 2007.
- [85] Jennifer Widom. Trio: A System for Data, Uncertainty, and Lineage. In *Managing and Mining Uncertain Data*. Springer, 2008.
- [86] Parag Agrawal, Robert Ikeda, Hyunjung Park, and Jennifer Widom. Trio-ER: The Trio System as a Workbench for Entity-Resolution. Technical report (march 2009), Stanford University, 2009. Avail. at: <http://ilpubs.stanford.edu:8090/912/>. Last visited on July 2019.
- [87] Yogesh L Simmhan, Beth Plale, Dennis Gannon, and Suresh Marru. Performance evaluation of the karma provenance framework for scientific workflows. In *Proceedings of the International Provenance and Annotation Workshop (IPAW'06)*, pages 222–236, 2006.
- [88] Yogesh L Simmhan, Beth Plale, and Dennis Gannon. A framework for collecting provenance in data-centric scientific workflows. In *Proceedings of the International Conference on Web Services (ICWS'06)*, pages 427–436. IEEE, 2006.
- [89] Mohammad Rezwani Huq, Andreas Wombacher, and Peter MG Apers. Inferring fine-grained data provenance in stream data processing: reduced storage cost, high accuracy. In *Proceedings of the International Conference on Database and Expert Systems Applications (DEXA'11)*, pages 118–127, 2011.
- [90] Yogesh L Simmhan, Beth Plale, and Dennis Gannon. Karma2: Provenance management for data-driven workflows. *Web Services Research for Emerging Applications: Discoveries and Trends: Discoveries and Trends*, 317, 2010.
- [91] Ian Foster, Jens Vöckler, Michael Wilde, and Yong Zhao. Chimera: A virtual data system for representing, querying, and automating data derivation. In *Proceedings of SSDBM'02*, pages 37–46. IEEE, 2002.
- [92] Ben Clifford, Ian Foster, Jens-S Voeckler, et al. Tracking provenance in a virtual data grid. *Concurr Compute*, 20(5):565–575, 2008.
- [93] Yong Zhao, Michael Wilde, and Ian Foster. Applying the virtual data provenance model. In *Proceedings of the International Provenance and Annotation Workshop (IPAW'06)*, volume 4145 of LNCS, pages 148–161. Springer, 2006.
- [94] Olivier Biton, Sarah Cohen-Boulakia, and Susan B Davidson. Zoom*UserViews: Querying relevant provenance in workflow systems. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'07)*, pages 1366–1369. VLDB Endowment, 2007.
- [95] Sarah Cohen-Boulakia, Olivier Biton, Shirley Cohen, and Susan Davidson. Addressing the Provenance Challenge using ZOOM. *Concurr Comput*, 20(5):497–506, 2008.

- [96] Olivier Biton, Sarah Cohen-Boulakia, Susan B Davidson, and Carmem S Hara. Querying and managing provenance through user views in scientific workflows. In *Proceedings of the IEEE 24th International Conference on Data Engineering (ICDE'08)*, pages 1072–1081. IEEE, 2008.
- [97] James Cheney and Roly Perera. An analytical survey of provenance sanitization. In *Proceedings of the International Provenance and Annotation Workshop (IPAW'14)*, pages 113–126, 2014.
- [98] Yingwei Cui and Jennifer Widom. Lineage tracing in data warehouses. In *Proceedings of the 16th International Conference on Data Engineering*, pages 367–378. IEEE, 2000.
- [99] Yingwei Cui, Jennifer Widom, and Janet L Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Transactions on Database Systems (TODS)*, 25(2):179–227, 2000.
- [100] Yingwei Cui and Jennifer Widom. Practical lineage tracing in data warehouses. In *Proceedings of the 16th International Conference on Data Engineering (ICDE'00)*, pages 367–378. IEEE, 2000.
- [101] Janet Wiener, Himanshu Gupta, Wilburt Labio, et al. A system prototype for warehouse view maintenance. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (MOD'95)*, pages 26–33, 1995.
- [102] Yong Zhao, M Hategan, and et al. Swift: Fast, Reliable, Loosely Coupled Parallel Computation. In *IEEE International Conference on Services Computing - Workshops (SCW'07)*, pages 199–206, 2007.
- [103] Luiz MR Gadelha Jr, Ben Clifford, Marta Mattoso, et al. Provenance management in Swift. *Futur Gener Comput Syst*, 27(6):775–780, 2011.
- [104] Anderson Marinho, Daniel de Oliveira, Eduardo Ogasawara, et al. Deriving scientific workflows from algebraic experiment lines: A practical approach. *Futur Gener Comput Syst*, 68:111–127, 2017.
- [105] Michael Wilde, Mihael Hategan, Justin M Wozniak, et al. Swift: A language for distributed parallel scripting. *Parallel Computing*, 37(9):633–652, 2011.
- [106] Paul Groth, Simon Miles, and Luc Moreau. PReServ: Provenance recording for services. *UK e-Science All Hands Meeting*, 2005.
- [107] Michael Stonebraker, Jolly Chen, Nobuko Nathan, et al. Tioga: Providing Data Management Support for Scientific Visualization Applications. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'93)*, pages 25–38, 1993.
- [108] Allison Woodruff and Michael Stonebraker. Supporting Fine-grained Data Lineage in a Database Visualization Environment. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE'97)*, pages 91–102, 1997.

- [109] Alexander Aiken, Jolly Chen, Michael Stonebraker, and Allison Woodruff. Tioga-2: A Direct Manipulation Database Visualization Environment. In *Proceedings of the Twelfth International Conference on Data Engineering (ICDE'96)*, pages 208–217, 1996.
- [110] E. Deelman, Gurmeet Singh, Mei-Hui Su, et al. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237, 2005.
- [111] Jihie Kim, Ewa Deelman, Yolanda Gil, et al. Provenance Trails in the Wings/Pegasus System. *Concurr Comput*, 20(5):587–597, 2008.
- [112] Ewa Deelman, James Blythe, Yolanda Gil, et al. Pegasus: Mapping scientific workflows onto the grid. In *Proceedings of the 2nd European Across Grids Conference (EAGC'04)*, pages 11–20. Springer, 2004.
- [113] Yolanda Gil, Varun Ratnakar, Ewa Deelman, et al. Wings for pegasus: Creating large-scale scientific applications using semantic representations of computational workflows. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence (AAAI'07)*, volume 22, page 1767, 2007.
- [114] Ewa Deelman, Gaurang Mehta, Gurmeet Singh, et al. Pegasus: mapping large-scale workflows to distributed resources. In *Workflows for e-Science*, pages 376–394. Springer, 2007.
- [115] Daniel Garijo, Yolanda Gil, and Óscar Corcho. Abstract, link, publish, exploit: An end to end framework for workflow sharing. *Futur Gener Comput Syst*, 75:271–283, 2017.
- [116] Yolanda Gil, Varun Ratnakar, Jihie Kim, et al. Wings: Intelligent workflow-based design of computational experiments. *IEEE Intelligent Systems*, 26(1):62–72, 2011.
- [117] Peter Buneman, Adriane Chapman, and James Cheney. Provenance Management in Curated Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (MOD'06)*, pages 539–550, 2006.
- [118] Ashish Gehani and Dawood Tariq. SPADE: support for provenance auditing in distributed environments. In *Proceedings of the 13th International Middleware Conference*, pages 101–120, 2012.
- [119] Ashish Gehani and Minyoung Kim. Mendel: Efficiently verifying the lineage of data modified in multiple trust domains. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC'10)*, pages 227–239. ACM, 2010.
- [120] Sheung Chi Chan, Ashish Gehani, James Cheney, et al. Expressiveness benchmarking for system-level provenance. In *Proceedings of the International Workshop on Theory and Practice of Provenance (TaPP'17)*. USENIX Association, 2017.
- [121] Manolis Stamatogiannakis, Hasanat Kazmi, Hashim Sharif, et al. Trade-offs in automatic provenance capture. In *Proceedings of the International Provenance and Annotation Workshop (IPAW'16)*, pages 29–41, 2016.

- [122] Ashish Gehani, Hasanat Kazmi, and Hassaan Irshad. Scaling spade to “big provenance”. In *Proceedings of the International Workshop on Theory and Practice of Provenance (TaPP’16)*, pages 26–33. USENIX Association, 2016.
- [123] Zachary G Ives, Nitin Khandelwal, Aneesh Kapur, and Murat Cakir. ORCHESTRA: Rapid, Collaborative Sharing of Dynamic Data. In *Proceedings of the Conference on Innovative Data Systems (CIDR’05)*, pages 107–118, 2005.
- [124] Todd J. Green, Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. Update Exchange with Mappings and Provenance. In *Proceedings of the International Conference on Very Large Data Bases (VLDB’07)*, pages 675–686, 2007.
- [125] Todd J. Green, Gregory Karvounarakis, Nicholas E. Taylor, et al. ORCHESTRA: facilitating collaborative data sharing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (MOD’07)*, pages 1131–1133, 2007.
- [126] Todd J Green and Val Tannen. The semiring framework for database provenance. In *Proceedings of PODS’17*, pages 93–99. ACM, 2017.
- [127] Zachary G Ives, Todd J Green, Grigoris Karvounarakis, et al. The orchestra collaborative data sharing system. *ACM SIGMOD Record*, 37(3):26–32, 2008.
- [128] Boris Glavic. *Perm: Efficient Provenance Support for Relational Databases*. PhD thesis, University of Zurich, 2010. Available at: <http://www.zora.uzh.ch/44573/1/dissGlavic.pdf>. Last visited on July 2019.
- [129] Bahareh Arab, Dieter Gawlick, Venkatesh Radhakrishnan, et al. A Generic Provenance Middleware for Database Queries, Updates, and Transactions. In *Proceedings of the International Workshop on Theory and Practice of Provenance (TaPP’14)*, 2014.
- [130] Boris Glavic, Renée J. Miller, and Gustavo Alonso. Using SQL for Efficient Generation and Querying of Provenance Information. In *In search of elegance in the theory and practice of computation: a Festschrift in honour of Peter Buneman*, pages 291–320, 2013.
- [131] Boris Glavic and Gustavo Alonso. Perm: Processing Provenance and Data on the same Data Model through Query Rewriting. In *Proceedings of the 25th IEEE International Conference on Data Engineering (ICDE’09)*, pages 174–185, 2009.
- [132] Boris Glavic and Gustavo Alonso. Provenance for Nested Subqueries. In *Proceedings of the 12th International Conference on Extending Database Technology (EDBT’09)*, pages 982–993, 2009.
- [133] Boris Glavic and Gustavo Alonso. The Perm Provenance Management System in Action. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (MOD’09) (Demonstration Track)*, pages 1055–1058, 2009.
- [134] Xing Niu, Raghav Kapoor, Boris Glavic, et al. Interoperability for Provenance-aware Databases using PROV and JSON. In *Proceedings of the International Workshop on Theory and Practice of Provenance (TaPP’15)*, 2015.

- [135] Bahareh Arab, Dieter Gawlick, Vasudha Krishnaswamy, et al. Reenacting Transactions to Compute their ProvenA system prototype for warehouse view maintenance. Technical Report IIT/CS-DB-2014-02, Illinois Institute of Technology, 2014. Available at: <http://cs.iit.edu/dbgroup/pdfpubs/AD14.pdf>. Last visited on July 2019.
- [136] Seokki Lee, Yuchen Tang, Sven Köhler, et al. An Efficient Implementation Of Game Provenance In DBMS. Technical Report IIT/CS-DB-2015-02, Illinois Institute of Technology, 2015.
- [137] Xing Niu, Raghav Kapoor, and Boris Glavic. Heuristic and Cost-based Optimization for Provenance Computation. In *Proceedings of the International Workshop on Theory and Practice of Provenance (TaPP'15)*, 2015.
- [138] James Frew and Peter Slaughter. Es3: A demonstration of transparent provenance for scientific computation. In *Proceedings of the International Provenance and Annotation Workshop (IPAW'08)*, volume 5272 of *LNCS*, pages 200–207. Springer, 2008.
- [139] James Frew, Dominic Metzger, and Peter Slaughter. Automatic capture and reconstruction of computational provenance. *Concurr Comput*, 20(5):485–496, 2008.
- [140] Shawn Bowers, Timothy M. McPhillips, and Bertram Ludäscher. Provenance in collection-oriented scientific workflows. *Concurr Comput*, 20(5):519–529, 2008.
- [141] Deepavali Bhagwat, Laura Chiticariu, Wang Chiew Tan, and Gaurav Vijayvargiya. An annotation management system for relational databases. *The VLDB Journal*, 14(4):373–396, 2005.
- [142] Laura Chiticariu, Wang Chiew Tan, and Gaurav Vijayvargiya. DBNotes: A Post-It System for Relational Databases based on provenance. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (MOD'05)*, pages 942–944. ACM, 2005.
- [143] Y. Amsterdamer, Susan B. Davidson, Daniel Deutch, et al. Putting Lipstick on Pig: Enabling Database-style Workflow Provenance. *PVLDB*, 5(4):346–357, 2011.
- [144] Roger S Barga and Luciano A Digiampietri. Automatic capture and efficient storage of e-Science experiment provenance. *Concurr Comput*, 20(5):419–429, 2008.
- [145] Philip J. Guo and Margo Seltzer. BURRITO: Wrapping Your Lab Notebook in Computational Infrastructure. In *Proceedings of the International Workshop on Theory and Practice of Provenance (TaPP'12)*, 2012.
- [146] Philip J. Guo. *Software Tools to Facilitate Research Programming*. PhD thesis, Stanford University, May 2012.
- [147] Peter Macko and Margo Seltzer. Provenance map orbiter: Interactive exploration of large provenance graphs. In *Proceedings of the International Workshop on Theory and Practice of Provenance (TaPP'11)*, 2011.
- [148] Adriane Chapman, Barbara T Blaustein, Len Seligman, and M David Allen. Plus: A provenance manager for integrated information. In *Proceedings of the IEEE International Conference on Information Reuse and Integration (IRI'11)*, pages 269–275. IEEE, 2011.

- [149] Adriane Chapman, M David Allen, Barbara Blaustein, et al. Plus: Provenance for life, the universe and stuff. *VLDB'10, VLDB Endowment*, pages 13–17, 2010.
- [150] Barbara Blaustein, Len Seligman, Michael Morse, et al. Plus: Synthesizing privacy, lineage, uncertainty and security. In *Proceedings of the 24th International Conference on Data Engineering Workshop (ICDEW'08)*, pages 242–245, 2008.
- [151] Robert Ikeda, Hyunjung Park, and Jennifer Widom. Provenance for generalized map and reduce workflows. In *Proceedings of the Fifth Biennial Conference on Innovative Data Systems (CIDR'11)*, pages 273–283, 2011.
- [152] Hyunjung Park, Robert Ikeda, and Jennifer Widom. Ramp: A system for capturing and tracing provenance in mapreduce workflows. *PVLDB*, 4(12):1351–1354, 2011.
- [153] Emilio Rodriguez-Priego, Francisco J. García-Izquierdo, and Ángel Luis Rubio. References-enriched Concept Map: a tool for collecting and comparing disparate definitions appearing in multiple references. *J. Information Science*, 39(6):789–804, 2013.
- [154] Novak, Joseph D. and Cañas, Alberto J. The theory underlying concept maps and how to construct and use them. Technical report, 2008. Research report, 2006-01 Rev 2008-01. Florida Institute for Human and Machine Cognition, (2006). Available at <http://cmap.ihmc.us/docs/theory-of-concept-maps>. Last visited on July 2019.
- [155] Sérgio Manuel Serra da Cruz, Maria Luiza Machado Campos, and Marta Mattoso. Towards a Taxonomy of Provenance in Scientific Workflow Management Systems. In *Proceedings of the IEEE Congress on Services, Part I, SERVICES I*, pages 259–266, 2009.
- [156] Szymon Klarman, Stefan Schlobach, and Luciano Serafini. Formal Verification of Data Provenance Records. In *Proceedings of the 11th International Semantic Web Conference (ISWC'12) Part I*, pages 215–230, 2012.
- [157] Sean Bechhofer, David De Roure, Matthew Gamble, Carole Goble, and Iain Buchan. Research objects Towards exchange and reuse of digital knowledge. *The Future of the Web for collaborative science. Raleigh: nature Precedings*, 2010.
- [158] Leonardo Murta, Vanessa Braganholo, Fernando Chirigati, David Koop, and Juliana Freire. noworkflow: capturing and analyzing provenance of scripts. In *International Provenance and Annotation Workshop*, pages 71–83. Springer, 2014.
- [159] Paul Groth. *The origin of data: Enabling the determination of provenance in multi-institutional scientific systems through the documentation of processes*. PhD thesis, University of Southampton, 2007. Available at <https://eprints.soton.ac.uk/264649/>. Last visited on July 2019.
- [160] ProvExtract. Available at <https://openprovenance.org/tools/extract/>. Last visited on July 2019.
- [161] ProvVis. Available at <https://openprovenance.org/vis/>. Last visited on July 2019.

- [162] L. Carata, Sherif Akoush, Nikilesh Balakrishnan, et al. A primer on provenance. *Communications of the ACM*, 57(5):52–60, January 2014.
- [163] L. Carata, Sherif Akoush, Nikilesh Balakrishnan, et al. A Primer on Provenance. *Communications of the ACM*, 12(3):10:10–10:23, March 2014.
- [164] Carlos Sáenz-Adán, Beatriz Pérez, Trung Dong Huynh, and Luc Moreau. UML2PROV: automating provenance capture in software engineering. In *Proceedings of the 44th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*, pages 667–681, 2018.
- [165] Carlos Sáenz-Adán, Luc Moreau, Beatriz Pérez, Simon Miles, and Francisco J. García-Izquierdo. Automating provenance capture in software engineering with UML2PROV. In *Proceedings of the International Provenance and Annotation Workshop (IPAW'18)*, pages 58–70, 2018.
- [166] Alexander Knapp and Stephan Merz. Model checking and code generation for uml state machines and collaborations. *Proceedings of the 5th Wsh. Tools for System Design and Verification (FM-TOOLS'02)*, pages 59–64, 2002.
- [167] Natalia Kwasnikowska, Luc Moreau, and Jan Van Den Bussche. A formal account of the open provenance model. *ACM Trans. Web*, 9(2):10:1–10:44, May 2015.
- [168] Carlos Sáenz-Adán, and Beatriz Pérez and Francisco J. García-Izquierdo and Luc Moreau. Supplementary material submitted to IEEE Transactions on Software Engineering. Available at <https://www.unirioja.es/cu/casaenad/thesis.html>.
- [169] P. Clarke, B. Malloy, and P. Gibson. Using a taxonomy tool to identify changes in OO software. In *Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR'03)*, pages 213–222, 2003.
- [170] Michael A. C. Johnson, Luc Moreau, Adriane Chapman, Poshak Gandhi, and Carlos Sáenz-Adán. Using the provenance from astronomical workflows to increase processing efficiency. In *Proceedings of the International Provenance and Annotation Workshop (IPAW'18)*, pages 101–112, 2018.
- [171] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison Wesley, 1995.
- [172] Mikaël Peltier, Jean Bézivin, and Gabriel Guillaume. Mtrans: A general framework, based on xslt, for model transformations. In *Workshop on Transformations in UML (WTUML)*, Genova, Italy, 2001.
- [173] Catalina Martínez-Costa, Marcos Menárguez-Tortosa, and Jesualdo Tomás Fernández-Breis. Clinical data interoperability based on archetype transformation. *Journal of Biomedical Informatics*, 44(5):869–880, 2011.
- [174] Java Class Proxy. Java Platform, Standard Edition 8 API Specification (Update 121), January 2017. Available at url-<https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Proxy.html>. Last visited on July 2019.

- [175] Java Interface InvocationHandler. Java Platform, Standard Edition 8 API Specification (Update 121), January 2017. Available at <https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/InvocationHandler.html>. Last visited on July 2019.
- [176] Danius Michaelides, Trung Dong Huynh, and Luc Moreau. PROV-TEMPLATE: A Template System for PROV Documents, 2014. Available at <https://provenance.ecs.soton.ac.uk/prov-template>. Last visited on July 2019.
- [177] MongoDB Inc. Version 4.0.2, August 2018. Available at www.mongodb.org/. Last visited on July 2019.
- [178] Jónathan Heras, César Domínguez, Eloy Mata, Vico Pascual, Carmen Lozano, Carmen Torres, and Myriam Zarazaga. GelJ – a tool for analyzing DNA fingerprint gel images. *BMC Bioinformatics*, 16(1), Aug 2015.
- [179] Marina May Read. *Trends in DNA Fingerprint Research*. Nova Science Publishers, Inc, New York, USA, 2005.
- [180] Ira D. Baxter and Michael Mehlich. Reverse engineering is reverse forward engineering. *Science of Computer Programming*, 36(2):131 – 147, 2000.
- [181] Akanksha Agarwal. *Trace Abstraction Based on Automatic Detection of Execution Phases*. PhD thesis, Concordia University, 2011.
- [182] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of software engineering (2. ed.)*. Prentice Hall, 2003.
- [183] Visual Paradigm, UML modeling tool. Version 15.0, December 2018. Available at www.visual-paradigm.com/. Last visited on July 2019.
- [184] Papyrus, Modeling environment . Version 2.0.2, December 2018. Available at eclipse.org/papyrus/. Last visited on July 2019.
- [185] Modelio, UML modeling tool. Version 3.6, December 2018. Available at www.modeliosoft.com/. Last visited on July 2019.
- [186] The ObjectAid UML Explorer for Eclipse. Version 1.2.2, November 2017. Available at www.objectaid.com/. Last visited on July 2019.
- [187] Thoms Ball. The concept of dynamic analysis. In *ACM SIGSOFT Software Engineering Notes*, volume 24, pages 216–234. Springer-Verlag, 1999.
- [188] MaintainJ. Version 4.2.0, 2014. Available at maintainj.com/. Last visited on July 2019.
- [189] Lionel C. Briand, Yvan Labiche, and Y. Miao. Towards the reverse engineering of UML sequence diagrams. In *10th Working Conference on Reverse Engineering, WCRE 2003, Victoria, Canada, November 13-16, 2003*, pages 57–66, 2003.
- [190] Yann-Gaël Guéhéneuc and Tewfik Ziadi. Automated reverse-engineering of uml v2. 0 dynamic models. In *Proceedings of the 6th ECOOP Workshop on Object-Oriented Reengineering*, 2005.

-
- [191] Neil Walkinshaw, Kirill Bogdanov, Shaukat Ali, and Mike Holcombe. Automated discovery of state transitions and their functions in source code. *Software Testing, Verification and Reliability*, 18(2):99–121, 2008.
- [192] Pieter Van Gorp, Hans Stenten, Tom Mens, and Serge Demeyer. Towards automating source-consistent uml refactorings. In *International Conference on the Unified Modeling Language*, pages 144–158. Springer, 2003.
- [193] Graham Klyne and Paul Groth (eds.). PROV-AQ: Provenance Access and Query. W3C Note, World Wide Web Consortium, 2013. Available at <https://www.w3.org/TR/2013/NOTE-prov-aq-20130430/#accessing-provenance-records>. Last visited on July 2019.

Appendix A

Provenance systems analysed in the systematic review

In Tables A.1 and A.2 we show the set of provenance systems we have identified in the systematic review in Chapter 3. More specifically, per each system we show (1) the identifier together with its name, and (2) a brief description of the system.

Table A.1 A brief description of the surveyed systems (I)

[ID] System	Description
[S1] VisTrails	It is a scientific workflow system developed at the University of Utah. One remarkable aspect of this system is the notion of provenance of workflow evolution.
[S2] myGrid/Taverna	It is a domain-independent Workflow Management System. Initially created by the myGrid team, it is now an Apache Incubator project.
[S3] Kepler provenance	Kepler-provenance is an add-on module which adds provenance features to the Kepler Scientific Workflow System presented in [76].
[S4] PASS	PASS (Provenance Aware Storage System) was developed at Harvard University as a storage system that supports the automatic collection and maintenance of provenance.
[S5] Trio	Developed at Stanford was defined as a new kind of database management system which extends relational databases with support for provenance and uncertainty.
[S6] Karma	It is a provenance collection and management system developed at Indiana University. It collects provenance for data-centric workflows in a service oriented architecture [87].
[S7] Chimera	Chimera is a part of the <i>GryPhyN</i> project, a research project which develops techniques for processing and managing large distributed data sets in data grids.
[S8] ZOOM	ZOOM, for short, presents a model of provenance for scientific workflows. Its main goal is to provide users with an interface to query provenance information provided by a workflow system as well as to help them construct an appropriate user view [94].
[S9] Cui 2000	We refer as <i>Cui 2000</i> to the lineage tracing system developed by Cui and Widom in 2000 [99], and which is based on the WHIPS [101] data warehousing system.

Table A.2 A brief description of the surveyed systems (II)

[ID] System	Description
[S10] Swift	A successor to Chimera (an outgrowth of the Chimera's Virtual Data Language), Swift is a scalable and reliable Grid workflow system that bridges scientific workflows with parallel computing.
[S11] PASOA/PreServ	PreServ (Provenance Recording for Services), developed within the PASOA project, is a software package that allows developers to integrate process documentation recording into their applications.
[S12] Tioga	Tioga is a DBMS-centric visualization tool developed at Berkeley. Based on this tool, Tioga-2 was designed in 1996 as a more powerful and much easier to program direct manipulation system.
[S13] Wings-Pegasus	Wings-Pegasus constitute two different tools which have been integrated to form a provenance system which works as a whole used for grid computing [113].
[S14] Buneman	In [117], Buneman et al. present a proposal in which the user's actions are tracked while browsing source databases and copying data into a target curated database, in order to record the user's actions in a convenient, queryable form. This proposal is known by the copy-paste-model. In this review, we refer to this proposal as Buneman, for short.
[S15] SPADE	SPADE was developed as a distributed service for collecting, certifying, and querying the provenance of Grid data.
[S16] ORCHESTA	It is a collaborative data sharing system which focuses mainly on managing disagreement among multiple data representations and instances.
[S17] Perm-GProM	GProM (Generic Provenance Middleware) is defined as a generic provenance database middleware that enables computation of provenance for queries, updates, and transactions over several database back-ends (e.g., Oracle).
[S18] ES3	ES3 (Earth System Science Server) is a software system designed for automatically and transparently capturing, managing, and reconstructing the provenance of arbitrary, unmodified computational sequences.
[S19] COMAD	CoMaD (Collection-Oriented Modeling and Design) is a provenance annotation-based framework, implemented as a part of the Kepler Scientific Workflow System.
[S20] DBNotes	DBNotes is an annotation management system for relational database systems.
[S21] Lipstick	It is a framework that marries database-style and workflow provenance models, capturing internal state as well as fine-grained dependencies in workflow provenance.
[S22] Redux	REDUX is a provenance management system which generates workflow provenance automatically during runtime. It uses the Windows Workflow Foundation (WinWF) [144] as workflow engine.
[S23] BURRITO	It is a Linux-based system that helps programmers organize, annotate and recall past insights about their experiments [146].
[S24] PLUS	PLUS is a provenance manager that was inspired by U.S. government requirements to enable provenance capture, storage and use across multi-organizational systems.
[S25] RAMP	RAMP (Reduce And Map Provenance) is an extension to Hadoop that supports provenance capture and tracing for MapReduce workflows. More specifically, MapReduce is a programming model and an associated implementation for processing and generating large data sets.

Appendix B

Description of the transformation patterns used in this memoir

This chapter provides the systematic explanation of those transformation patterns that have been used to illustrate this memoir (see Table B.1). The complete explanation of all the patterns together is provided in [168].

Table B.1 Set of patterns used to illustrate this memoir

UML Sequence Diagrams Patterns	
Pattern identifier	Page
<i>Sequence diagram Pattern 1 (SeqP1)</i>	168
<i>Sequence diagram Pattern 2 (SeqP2)</i>	172

UML Class Diagrams Patterns	
Pattern identifier	Page
<i>Class diagram Pattern 10 (CIP10)</i>	177

UML State Machine Patterns	
Pattern identifier	Page
<i>State machine diagram Pattern 1 (StP1)</i>	184
<i>State machine diagram Pattern 3 (StP3)</i>	189

Identifier *Sequence diagram Pattern 1 (SeqP1)*

Context

A participant (the sender) interacts with another participant (the recipient) by calling an operation in the recipient, and then, it continues immediately. The call causes the recipient to execute the operation.

Key elements

<i>Sender</i>	The participant that makes the operation call.
<i>Operation call</i>	The call that starts the execution of the operation.
<i>Input data</i>	The information (if any) passed to the operation through the <i>Operation call</i> .
<i>Operation execution</i>	The execution of the operation.

UML Diagram

Key Element	UML	Rationale
<i>Sender</i>	Lifeline 1	It models the <i>Sender</i> participant involved in the interaction.
<i>Operation call</i>	Asynchronous Message	It models the <i>Operation call</i> when the <i>Sender</i> does not wait for a response, but instead continues immediately after sending the message.
<i>Input data</i>	Input Arguments	They specify the information passed to the operation through the <i>Operation call</i> .
<i>Operation execution</i>	ExecutionSpecification	It shows the period of time that the recipient's participant devotes to the <i>Operation execution</i> .

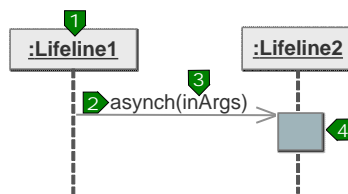


Fig. B.1 UML representation that models the context given by *SeqP1*

Sequence diagram Pattern 1 (SeqP1)

Mapping to PROV

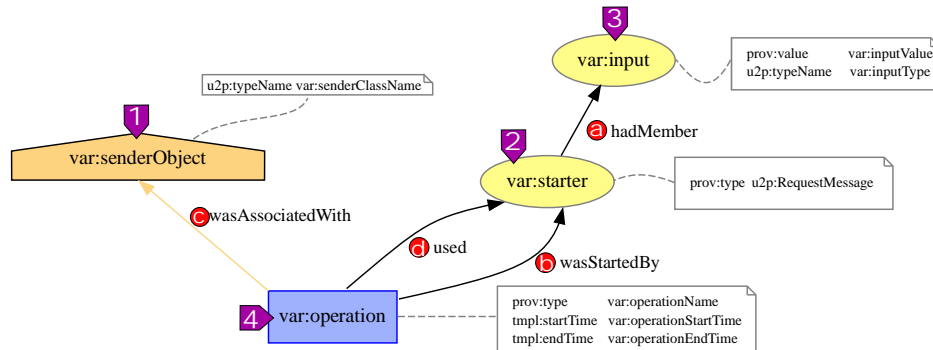


Fig. B.2 PROV template generated from the UML representation used in *SeqP1* (Figure B.1)

PROV elements

UML	PROV / id	Rationale
Lifeline 1	prov:Agent 1 / var:senderObject	The sender Lifeline 1 is mapped to a prov:Agent identified by var:senderObject. It assumes the responsibility for starting the ExecutionSpecification 4.
Asynchronous Message 2	prov:Entity 2 / var:starter	The Asynchronous Message 2 that initiates the ExecutionSpecification 4 of the recipient is a prov:Entity with identifier var:starter.
Input Arguments 3	prov:Entity 3 / var:input	Each argument of Input Arguments 3 is a separate prov:Entity identified as var:input.
ExecutionSpecification 4	prov:Activity 4 / var:operation	The ExecutionSpecification 4 is a prov:Activity with identifier var:operation.

Attributes

PROV Element	Attribute / Value	Description
<code>var:senderObject</code> 1	<code>u2p:typeName /</code> <code>var:senderClassName</code>	The value <code>var:senderClassName</code> is the string with the name of the class to which the <code>var:senderObject</code> 1 belongs.
<code>var:starter</code> 2	<code>prov:type /</code> <code>u2p:RequestMessage</code>	The value <code>u2p:RequestMessage</code> shows that <code>var:starter</code> 2 is a request message.
<code>var:input</code> 3	<code>prov:value /</code> <code>var:inputValue</code> <code>u2p:typeName /</code> <code>var:inputType</code>	The value <code>var:inputValue</code> is the direct representation of <code>var:input</code> 3. The value <code>var:inputType</code> is the string with the name of the class to which <code>var:input</code> 3 belongs.
<code>var:operation</code> 4	<code>prov:type /</code> <code>var:operationName</code> <code>tmpl:startTime /</code> <code>var:operationStartTime</code> <code>tmpl:endTime /</code> <code>var:operationEndTime</code>	The value <code>var:operationName</code> is the name of the operation <code>var:operation</code> 4. <code>var:operationStartTime</code> is an <code>xsd:dateTime</code> value for the start of <code>var:operation</code> 4. <code>var:operationEndTime</code> is an <code>xsd:dateTime</code> value for the end of <code>var:operation</code> 4.





PROV relations

PROV Relation	Description
a <code>prov:hadMember</code>	It states that <code>var:input</code> is one of the elements in <code>var:starter</code> .
b <code>prov:wasStartedBy</code>	<code>var:operation</code> is deemed to have been started by <code>var:starter</code> .
c <code>prov:wasAssociatedWith</code>	It is the assignment of responsibility to <code>var:senderObject</code> for <code>var:operation</code> .
d <code>prov:used</code>	It is the beginning of utilizing <code>var:starter</code> by <code>var:operation</code> .

Discussion

- Figure B.2 depicts the responsibility of the *Sender* lifeline (`var:senderObject`) for the recipient lifeline to execute the operation (`var:operation`). However, the recipient lifeline is not modeled in this PROV template, even though it is the participant that executes the operation. This decision is based on other patterns' better ability to both (1) identify the participant responsible for executing that operation, and (2) give a more detailed information about the implications that the execution of that operation has in the recipient participant. More specifically, these patterns are: *StP1-StP3*, which mainly focus on representing possible changes in an object's state caused by an *Operation execution*; and patterns *CIP1-CIP10*, which put more stress on how the execution affects the status of the object responsible for performing such an execution.

Sequence diagram Pattern 1 (*SeqP1*)

-
- Although the *context* of this pattern does not explicitly state that *Input data* should be passed to the operation, we have considered this circumstance with the aim of covering a wider spectrum of cases. When the *Operation call* lacks *Input data*, the UML representation in Figure B.1 will not include *Input Arguments* . As a consequence, the resulting PROV template in Figure B.2 will also lack *var:input*  and its associated PROV relations. Finally, we remark that the resulting PROV template does not reflect the *usage* of *var:input*  by *var:operation*  because SqDs stick to the flow of information, not its usage. Patterns addressing CDs (*CIP1-CIP10*) are better suited for this purpose.

Identifier *Sequence diagram Pattern 2 (SeqP2)*

Context

A participant (the sender) interacts with another participant (the recipient) by calling an operation in the recipient and waiting for a response. The call causes the recipient to execute the operation and to respond the sender after the execution.

Key elements

<i>Sender</i>	The participant that makes the operation call.
<i>Operation call</i>	The call that starts the execution of the operation.
<i>Input data</i>	The information (if any) passed to the operation through the <i>Operation call</i> .
<i>Operation execution</i>	The execution of the operation.
<i>Response</i>	The recipient's response to the <i>Operation call</i> .
<i>Output data</i>	The information contained in the <i>Response</i> .

UML Diagram

Key Element	UML	Rationale
<i>Sender</i>	Lifeline 1▶	It models the <i>Sender</i> participant involved in the interaction.
<i>Operation call</i>	Synchronous Message 2▶	It models the <i>Operation call</i> when the <i>Sender</i> waits for a response.
<i>Input data</i>	Input Arguments 3▶	They specify the information passed to the operation through the <i>Operation call</i> .
<i>Operation execution</i>	ExecutionSpecification 4▶	It shows the period of time that the recipient's participant devotes to the <i>Operation execution</i> .
<i>Response</i>	Reply Message 5▶	It specifies the response to the <i>Operation call</i> .
<i>Output data</i>	Output Arguments 6▶	They specify the information contained in the <i>Response</i> .

Sequence diagram Pattern 2 (SeqP2)

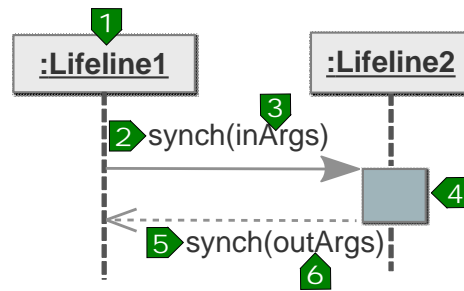


Fig. B.3 UML representation that models the context given by *SeqP2*

Mapping to PROV

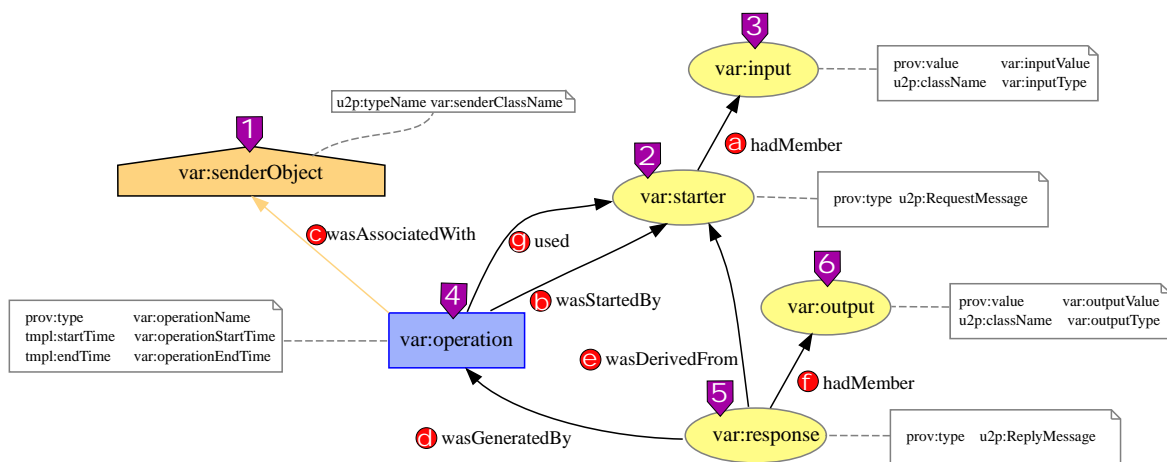


Fig. B.4 PROV template generated from the UML representation used in *SeqP2* (Figure B.3)

PROV elements

UML	PROV / id	Rationale
Lifeline 1	prov:Agent 1 / var:senderObject	The sender Lifeline 1 is mapped to a prov:Agent identified by var:senderObject. It assumes the responsibility for starting the ExecutionSpecification 4.
Synchronous Message 2	prov:Entity 2 / var:starter	The Synchronous Message 2 that initiates the ExecutionSpecification 4 of the recipient is a prov:Entity with identifier var:starter.

PROV elements

UML	PROV / id	Rationale
Input Arguments 3	prov:Entity 3 / var:input	Each argument of Input Arguments 3 is a separate prov:Entity identified as var:input.
ExecutionSpecification 4	prov:Activity 4 / var:operation	The ExecutionSpecification 4 is a prov:Activity with identifier var:operation.
Reply Message 5	prov:Entity 5 / var:response	The Reply Message 5 that responds to the Synchronous Message 2 is a prov:Entity with identifier var:response.
Output Arguments 6	prov:Entity 6 / var:output	Each argument of Output Arguments 6 is a separate prov:Entity identified as var:output.

Attributes

PROV Element	Attribute / Value	Description
var:senderObject 1	u2p:typeName / var:senderClassName	The value var:senderClassName is the string with the name of the class to which the var:senderObject 1 belongs.
var:starter 2	prov:type / u2p:RequestMessage	The value u2p:RequestMessage shows that var:starter 2 is a request message.
var:input 3	prov:value / var:inputValue	The value var:inputValue is the direct representation of var:input 3.
	u2p:typeName / var:inputType	The value var:inputType is the string with the name of the class to which the var:input 3 belongs.
var:operation 4	prov:type / var:operationName	The value var:operationName is the name of the operation var:operation 4.
	tmpl:startTime / var:operationStartTime	The var:operationStartTime is an xsd:dateTime value for the start of var:operation 4.
	tmpl:endTime / var:operationEndTime	The var:operationEndTime is an xsd:dateTime value for the end of var:operation 4.

Attributes

PROV Element	Attribute / Value	Description
var:response 5	prov:type / u2p:ReplyMessage	The value u2p:ReplyMessage shows that var:response 5 is a reply message.
var:output 6	prov:value / var:outputValue	The value var:outputValue is the direct representation of var:output 6.
	u2p:typeName / var:outputType	The value var:outputType is a string with the name of the class to which var:output 6 belongs.

PROV relations

PROV Relation	Description
a prov:hadMember	It states that var:input is one of the elements in var:starter.
b prov:wasStartedBy	var:operation is deemed to have been started by var:starter.
c prov:wasAssociatedWith	It is the assignment of responsibility to var:senderObject for var:operation.
d prov:wasGeneratedBy	It is the completion of production of var:response by var:operation.
e prov:wasDerivedFrom	It is the construction of var:response based on var:starter reception.
f prov:hadMember	It states that var:output is one of the elements in var:response.
g prov:used	It is the beginning of utilizing var:starter by var:operation.

Discussion

- Figure B.4 depicts the responsibility of the *Sender* lifeline (var:senderObject) for executing the operation (var:operation) in a recipient lifeline. However, the recipient lifeline is not modeled in this PROV template, even though it is the participant that executes the operation. This decision is based on other patterns' better ability to both (1) identify the participant responsible for executing that operation, and (2) give a more detailed information about the implications that the execution of that operation has in the recipient participant. More specifically, these patterns are: *StP1-StP3*, which mainly focus on representing possible changes in an object's state caused by an *Operation execution*; and patterns *CIP1-CIP10*, which put more stress on how the execution affects the status of the object responsible for performing such an execution.
- Although the *context* of this pattern does not explicitly state that *Input data* should be passed to the operation, we have considered this circumstance with the aim of covering a wider

spectrum of cases. When the *Operation call* lacks *Input data*, the UML representation in Figure B.3 will not include *Input Arguments* ③. As a consequence, the resulting PROV template in Figure B.4 will also lack *var:input* ③ and its associated PROV relations. Finally, we remark that the resulting PROV template does not reflect the *usage* of *var:input* ③ by *var:operation* ④ because SqDs stick to the flow of information, not its usage. Patterns addressing CDs (*CIP1-CIP10*) are better suited for this purpose.

Identifier *Class diagram Pattern 10 (CIP10)*

Context

The execution of an operation on an object directly adds the information passed to the operation as new element(s) of a concrete object's collection attribute, thus provoking a change in the object's status.

Key elements

<i>Object</i>	The object to which the operation to be executed belongs.				
<i>Pre-operation object</i>	The object with the status before the execution of the operation.				
<i>Post-operation object</i>	The object with the status after the execution of the operation.				
<i>Operation execution</i>	The execution of the behaviour specified by the operation.				
<i>Input data</i>	The information passed into the <i>Operation execution</i> .				
<i>Object's attributes</i>	All the characteristics of the <i>Object</i> . Since, as a consequence of the <i>Operation execution</i> , a concrete collection attribute changes, we have identified: <table> <tr> <td><i>Modified collection attribute</i></td> <td>The modified <i>Object's attribute</i>.</td> </tr> <tr> <td><i>Unmodified attributes</i></td> <td>The not modified <i>Object's attributes</i>.</td> </tr> </table>	<i>Modified collection attribute</i>	The modified <i>Object's attribute</i> .	<i>Unmodified attributes</i>	The not modified <i>Object's attributes</i> .
<i>Modified collection attribute</i>	The modified <i>Object's attribute</i> .				
<i>Unmodified attributes</i>	The not modified <i>Object's attributes</i> .				

UML Diagram

Key Element	UML	Rationale
<i>Object</i>	Class 1	<i>Objects</i> are classified attending to their characteristics and behaviour by means of classes. Thus, we use Class 1 to represent the <i>Object</i> both before and after the execution of the operation (<i>Pre-operation object</i> and <i>Post-operation object</i> , respectively).
<i>Operation execution</i>	Operation 2 «add»	The Operation 2 stereotyped by «add» represents the executed operation. Concretely, the stereotype «add» denotes that a new element (or elements) is directly added to a concrete collection attribute.
<i>Input data</i>	Input Parameters 3	They specify the information passed into the <i>Operation execution</i> .
<i>Object's attributes</i>	Attributes 4	They represent the characteristics of the <i>Object</i> .

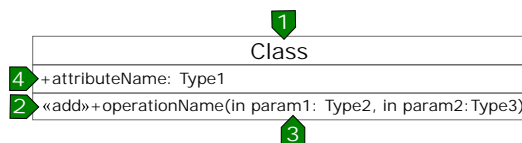


Fig. B.5 UML representation that models the context given by CIP10

Mapping to PROV

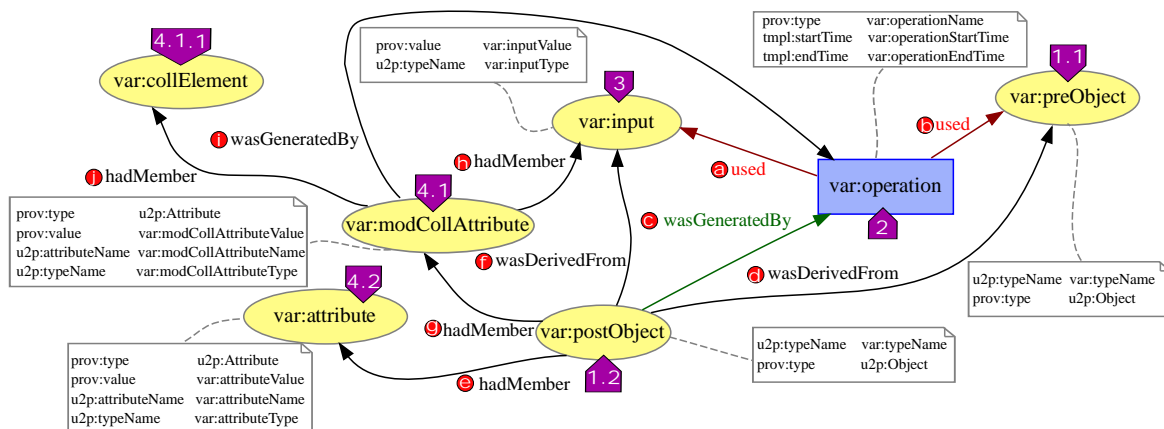


Fig. B.6 PROV template generated from the UML representation used in CIP10 (Figure B.5)

PROV elements

UML	PROV / id	Rationale
Class 1	prov:Entity 1.1 / var:preObject	The <i>Pre-operation object</i> , i.e. the object with the status before the execution of the operation, which is represented by Class 1, is a prov:Entity identified as var:preObject.
	prov:Entity 1.2 / var:postObject	The <i>Post-operation object</i> , i.e. the object with the status after the execution of the operation, which is represented by Class 1, is a prov:Entity identified as var:postObject.
Operation 2 «add»	prov:Activity 2 / var:operation	The execution of Operation 2 stereotyped by «add» is a prov:Activity identified by var:operation.

PROV elements

UML	PROV / id	Rationale
Input Parameters 3	prov:Entity 3 / var:input	Each parameter of Input Parameters 3 is a separate prov:Entity identified as var:input.
Attributes 4	prov:Entity 4.1 / var:modCollAttribute	The <i>Modified collection attribute</i> (belonging to Attributes 4) is a prov:Entity with identifier var:modCollAttribute. Additionally, each element in this collection is a separate prov:Entity identified by var:collelement 4.1.1
	prov:Entity 4.2 / var:attribute	Each <i>Unmodified attribute</i> (belonging to Attributes 4) is mapped to a separate prov:Entity with identifier var:attribute.

Attributes

PROV Element	Attribute / Value	Description
var:preObject 1.1	u2p:typeName / var:typeName	The value var:typeName is the string with the name of the class to which var:preObject 1.1 belongs.
	prov:type / u2p:object	The value u2p:object shows that var:preObject 1.1 is an object.
var:postObject 1.2	u2p:typeName / var:typeName	The value var:typeName is the string with the name of the class to which var:postObject 1.2 belongs.
	prov:type / u2p:object	The value u2p:object shows that var:postObject 1.2 is an object.
var:operation 2	prov:type / var:operationName	The value var:operationName is the name of the operation var:operation 2.
	tmpl:startTime / var:operationStartTime	The var:operationStartTime is an xsd:dateTime value for the start of var:operation 2.
	tmpl:endTime / var:operationEndTime	The var:operationEndTime is an xsd:dateTime value for the end of var:operation 2.
var:input 3	prov:value / var:inputValue	The value var:inputValue is the direct representation of var:input 3.
	u2p:typeName / var:inputType	The value var:inputType is the string with the name of the type of var:input 3.




Attributes

PROV Element	Attribute / Value	Description
var:modCollAttribute 4.1	prov:type / u2p:Attribute	The value u2p:Attribute shows that var:modCollAttribute 4.1 is an attribute.
	prov:value / var:newCollValue	The value var:newCollValue is the direct representation of var:modCollAttribute 4.1.
	u2p:attributeName / var:newCollName	The value var:newCollName is the string with the name of var:modCollAttribute 4.1.
	u2p:typeName / var:newCollType	The value var:newCollType is the string with the name of the type of var:modCollAttribute 4.1.
var:attribute 4.2	prov:type / u2p:Attribute	The value u2p:Attribute shows that attribute 4.2 is an attribute.
	prov:value / var:attributeValue	The value var:attributeValue is the direct representation of attribute 4.2.
	u2p:attributeName / var:attributeName	The value var:attributeName is the string with the name of attribute 4.2.
	u2p:typeName / var:attributeClass	The value var:attributeClass is the string with the name of the type of attribute 4.2.




PROV relations

PROV Relation	Description
a prov:used	It is the beginning of utilizing var:input by var:operation.
b prov:used	It is the beginning of utilizing var:preObject by var:operation.
c prov:wasGeneratedBy	It is the completion of production of var:postObject by var:operation.
d prov:wasDerivedFrom	It is the update of var:preObject resulting in var:postObject.
e prov:hadMember	It states that var:attribute is one of the elements in var:postObject.
f prov:wasDerivedFrom	It is the construction of var:postObject based on var:input.
g prov:hadMember	It states that var:modCollAttribute is one of the elements in var:postObject.

PROV relations

PROV Relation	Description
 prov:hadMember	It states that <code>var:input</code> is one of the elements in <code>var:modCollAttribute</code> . This is due to the fact that in this context the input information is directly added to the object's collection attribute.
 prov:wasGeneratedBy	It is the completion of production of <code>var:modCollAttribute</code> by <code>var:operation</code> .
 prov:hadMember	It states that <code>var:colleElement</code> is one of the elements in <code>var:modCollAttribute</code> .

Discussion

- Among the Class Diagrams patterns, patterns from *CIP6* to *CIP10* address the execution of operations that change an object's status. While, *CIP6* changes the object's status as a whole (being the concrete modified attributes unknown or irrelevant), in patterns *CIP7*-*CIP10* the concrete attributes modified by the *Operation execution* are explicitly known. In contrast to *CIP7* which directly sets the information passed into the *Operation execution* as values of concrete object's attributes, the other mentioned patterns use such information to change the object's status as a whole or the values of concrete object's attributes. It must also be noted that patterns *CIP9* and *CIP10* address the execution of operations which remove or add elements from/into an object's collection attribute, while patterns *CIP7* and *CIP8* affect either a univalued attribute or a collection attribute as a whole.
- A question that might arise is why in Figure B.6 `var:attribute`  is associated with `var:postObject`  (which represents the object with the status after the execution of the operation), but it is not associated with `var:preObject`  (the object with the status before the execution). We have made this decision because an object that acts as a `var:preObject` in an operation execution, was a `var:postObject` in a previous operation execution. Thus, the attributes associated with such an object in a `var:preObject` were registered when it previously played the role of `var:postObject`.
- Although the *context* of this pattern does not explicitly state that output data should be obtained from the *Operation execution*, this could be the case. However, we do not include this output data in this pattern description to avoid overburden both the UML and PROV explanations with information out of the scope of the *context*.

Aiming at giving an insight into how the inclusion of *Output data* affects both UML representation and the resulting PROV template, Figure B.7 depicts a UML representation

Class diagram Pattern 10 (*CIP10*)

with the *Output data* modeled as Output Parameters 5 (in this case with *return* direction, though the translation of *inout* and *out* directions would be equivalent). Figure B.8 depicts its transformation into PROV. Both Figure B.7 and B.8 highlight the elements related to the inclusion of the *Output data* by blurring the elements coming from Figure B.5 and B.6, respectively.

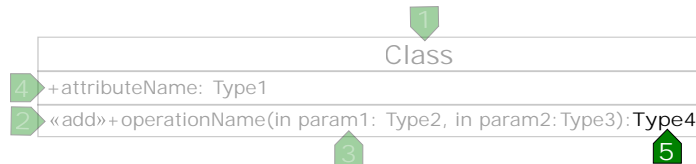


Fig. B.7 UML representation that models the context given by *CIP10*, including Output Parameters

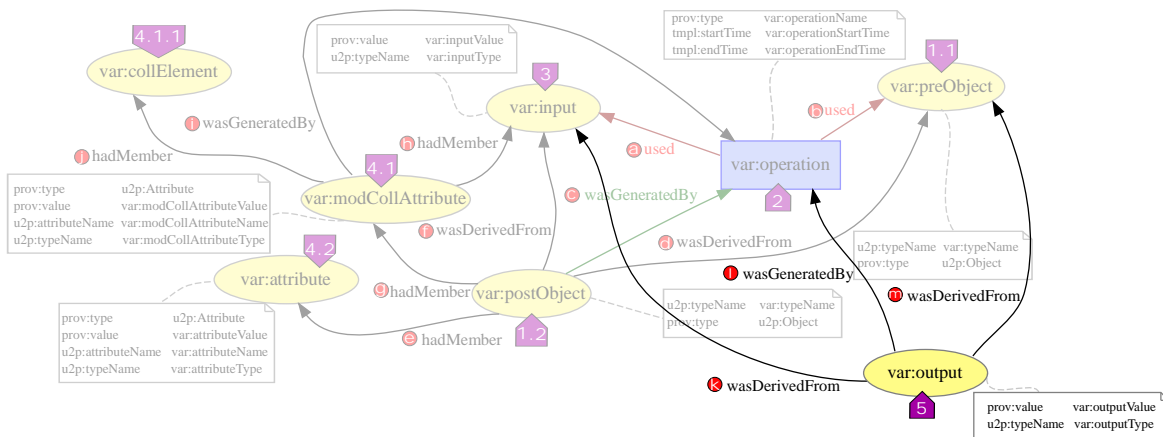


Fig. B.8 PROV template generated from the UML representation used in *CIP10*, including Output Parameters (Figure B.7)




PROV elements

UML	PROV / id	Rationale
Output Parameters 5	prov:Entity 5 / var:output	Each parameter of Output Parameters 5 is a separate prov:Entity identified as var:output.

PROV relations

PROV Relation	Description
prov:wasDerivedFrom (K)	It is the construction of var:output based on var:input.
prov:wasGeneratedBy (I)	It is the completion of production of var:output by var:operation.
prov:wasDerivedFrom (T)	It is the construction of var:output based on var:preObject.

Attributes

PROV Element	Attribute / Value	Description
var:output 	prov:value / var:outputValue	The value var:outputValue is the direct representation of var:output  .
	u2p:typeName / var:outputType	The value var:outputType is the string with the name of the type of var:output  .

Identifier *State machine diagram Pattern 1 (StP1)*

Context

As a consequence of the execution of an operation, an object is created in its first state. This operation is usually the constructor of the object.





Key elements

Object The object created as a consequence of the execution of the operation.

First object's state The first state after the object creation. This is the first state the object may undergo during its lifetime.

Object creation The execution of the operation that creates the object.

UML Diagram

Key Element	UML	Rationale
<i>Object</i>	Object 	It represents the created object. <i>Note:</i> since Object lacks a graphical representation in UML State Machine diagrams, Figure B.9 does not depict this element.
	StateMachine 	In UML, a StateMachine represents the set of states an <i>Object</i> can go through during its lifetime in response to events.
<i>Object creation</i>	Initial Pseudostate 	It refers to the execution of the operation that creates the <i>Object</i> , leading it to its first state.
	<i>First object's state</i>	State 

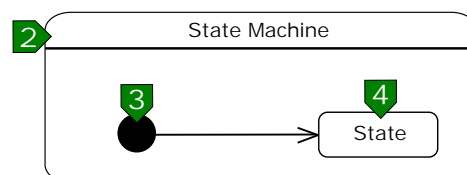


Fig. B.9 UML representation that models the context given by *StP1*

State machine diagram Pattern 1 (StP1)

Mapping to PROV

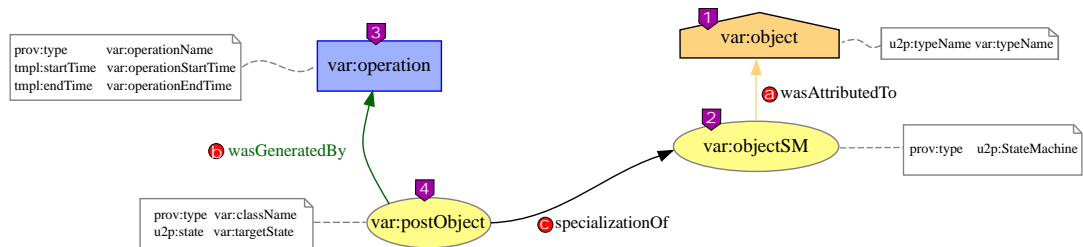


Fig. B.10 PROV template generated from the UML representation used in *StPI* (Figure B.9)

PROV elements

UML	PROV / id	Rationale
Object 1	prov:Agent 1 / var:object	The Object 1 bears some form of responsibility for the existence of the StateMachine 2, since the existence of StateMachine 2 does not make sense without an Object 1. To reflect this fact, the Object 1 is mapped to a prov:Agent identified by var:object.
StateMachine 2	prov:Entity 2 / var:objectSM	The StateMachine 2 is a prov:Entity identified by var:objectSM. It reflects the abstraction of the object's states, which will be specialized by each concrete state the object goes through.
Initial Pseudostate 3	prov:Activity 3 / var:operation	The Initial Pseudostate 3, referring to the execution of the operation that creates the Object 1, is a prov:Activity with the identifier var:operation.
State 4	prov:Entity 4 / var:postObject	The State 4 is a prov:Entity identified by var:postObject. We use this name for this identifier because it corresponds to the state of the Object 1 after (post) the object creation.

Attributes

PROV Element	Attribute / Value	Description
var:object 1▶	u2p:typeName / var:typeName	The value var:typeName is the string with the name of the class to which var:object 1▶ belongs.
var:objectSM 2▶	prov:type / u2p:StateMachine	The value u2p:StateMachine shows that var:objectSM 2▶ is a state machine.
var:operation 3▶	prov:type / var:operationName	The value var:operationName is the name of the operation var:operation 3▶.
	tmpl:startTime / var:operationStartTime	The var:operationStartTime is an xsd:dateTime value for the start of var:operation 3▶.
	tmpl:endTime / var:operationEndTime	The var:operationEndTime is an xsd:dateTime value for the end of var:operation 3▶.
var:postObject 4▶	prov:type / var:typeName	The value var:typeName is the name of the class to which the object in the state var:postObject 4▶ belongs.
	u2p:state / var:targetState	The value var:targetState is the string with the name of the state var:postObject 4▶.

PROV relations

PROV Relation	Description
a prov:wasAttributedTo	It is the assignment of responsibility to var:object for var:objectSM.
b prov:wasGeneratedBy	It is the completion of production of var:postObject by var:operation.
c prov:specializationOf	var:postObject is a specialization of var:objectSM.

Discussion

- Note that Figure B.9 only contains simple states. We do not deal with composite or submachine states, and focus only on simple states, because the former may be transformed into the latter by resorting to a flattening process consisting of removing composite states as well as submachine states. In fact, to flatten State Machine diagrams is a very common approach in contexts such as model checking and code generation [166]. However, the user might be interested in representing composite states directly into the PROV templates, perhaps because she/he is interested in collecting information about them, or just because she/he does not want to flatten the State Machine diagram. We can give an insight into how composite states can be mapped to PROV by placing the elements from Figure B.9 inside a Composite State 5▶ (see Figure B.11). A reader familiar with the UML specification will realize that the semantics of the Initial Pseudostate 3▶ in Figures B.9 and B.11 are

State machine diagram Pattern 1 (StPI)

different, but these semantic nuances would have no effect on the PROV transformation. The transformation of Figure B.11 is shown in Figure B.12. Both Figure B.11 and B.12 highlight the added elements by blurring the elements coming from Figure B.9 and Figure B.10, respectively. Briefly speaking, the new Composite State 5 is translated into a `prov:Entity` identified by `var:compState` 5, which is associated with `var:objectSM` 2 and `var:targetState` 4 by means of the relations `prov:specializationOf` and `prov:hadMember`, respectively. At this point, it is also worth remarking that for this example we have used a *simple composite state* (i.e., Composite State 5), which means that only one substate is active at a given time within such a state; but we could have used *orthogonal composite states* instead, which means that within such a state several substates are active at the same time. Note that both types of *composite states* would be translated into the same PROV template (see Figure B.12); nevertheless, the generated bindings would be different. In case of a *simple composite state*, as there can be only one active substate at the same time, there would be only one value associated with the variable `var:postObject` 4. Conversely, in case of an *orthogonal composite state*, `var:postObject` 4 will be associated with several values (as many as active states).

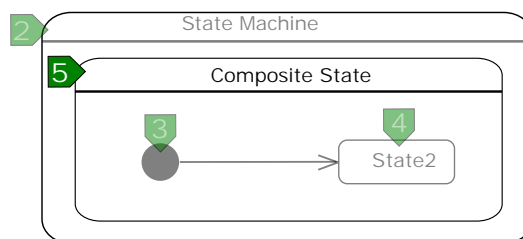


Fig. B.11 Excerpt of a UML State Machine diagram locating the UML elements from *StP1* in a *simple composite state*

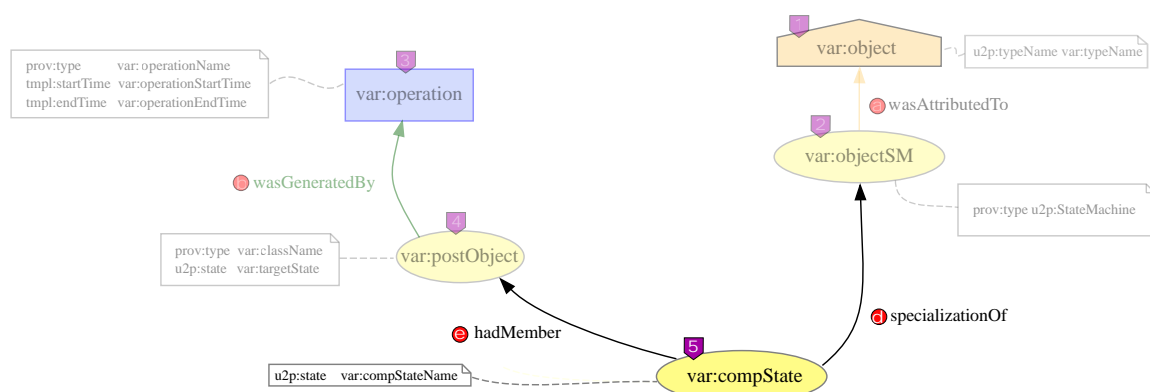







Fig. B.12 PROV template generated from the UML diagram in Figure B.11

State machine diagram *Pattern 1 (StP1)*



PROV elements

UML	PROV / id	Rationale
Composite State 	prov:Entity  / var:compState	The Composite State  is a prov:Entity identified by var:compState.

Attributes

PROV Element	Attribute / Value	Description
var:compState 	u2p:state / var:compStateName	The value var:compStateName is the string with the name of the state var:compState 

PROV relations

PROV Relation	Description
 prov:specializationOf	var:compState is a specialization of var:objectSM.
 prov:hadMember	It states that var:postObject is one of the elements in var:compState.

Identifier State machine diagram Pattern 3 (StP3)

Context

As a consequence of the execution of an operation, an object changes its state.

Key elements

<i>Object</i>	The object that changes its state.
<i>Pre-operation object's state</i>	The state of the object before the execution of the operation. This is one of the states the object may undergo during its lifetime.
<i>Post-operation object's state</i>	The state of the object after the execution of the operation. This is one of the states the object may undergo during its lifetime.
<i>Operation execution</i>	The execution of the operation that leads a change in the <i>Object's</i> state.

UML Diagram

Key Element	UML	Rationale
<i>Object</i>	Object 1▶	It represents the object that changes its state. <i>Note:</i> since Object lacks a graphical representation in UML State Machine diagrams, Figure B.13 does not depict this element.
	StateMachine 2▶	In UML, a StateMachine can be used to express the set of object's states through which the <i>Object</i> goes during its lifetime in response to events.
<i>Pre-operation object's state</i>	State 3▶	It models the state of the <i>Object</i> before the <i>Operation execution</i> .
<i>Post-operation object's state</i>	State 4▶	It models the state of the <i>Object</i> after the <i>Operation execution</i> .
<i>Operation execution</i>	Event 5▶	It specifies that the <i>Operation execution</i> that triggers the change in the <i>Object's</i> state has taken place.

State machine diagram Pattern 3 (StP3)

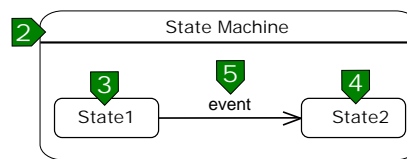


Fig. B.13 UML representation that models the context given by *StP3*

Mapping to PROV

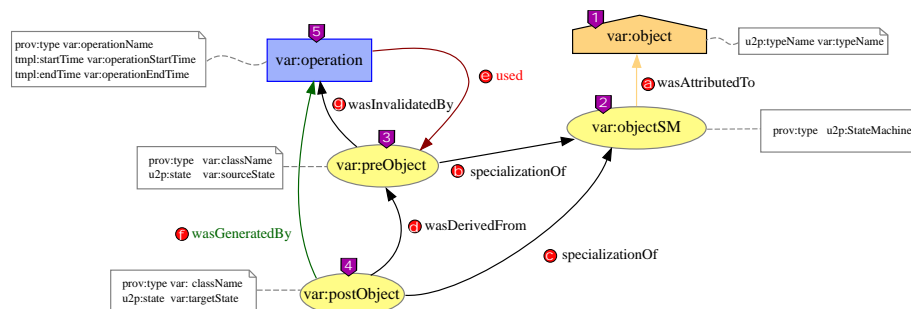


Fig. B.14 PROV template generated from the UML representation used in *StP3* (Figure B.13)

PROV elements

UML	PROV / id	Rationale
Object 1	prov:Agent 1 / var:object	The Object 1 bears some form of responsibility for the existence of the StateMachine 2, since the existence of StateMachine 2 does not make sense without an Object 1. To reflect this fact, the Object 1 is mapped to a prov:Agent identified by var:object.
StateMachine 2	prov:Entity 2 / var:objectSM	The StateMachine 2 is a prov:Entity identified by var:objectSM. It reflects the abstraction of the object's states, which will be specialized by each state the object goes through.
State 3	prov:Entity 3 / var:preObject	The State 3 is a prov:Entity identified by var:preObject. We use this name for this identifier because it corresponds to the state of the Object 1 before (pre) the execution of the operation.
State 4	prov:Entity 4 / var:postObject	The State 4 is a prov:Entity identified by var:postObject. We use this name for this identifier because it corresponds to the state of the Object 1 after (post) the execution of the operation.
Event 5	prov:Activity 5 / var:operation	The Event 5 represents that the execution of an operation has taken place. Such an execution is a prov:Activity with the identifier var:operation.

State machine diagram Pattern 3 (*StP3*)

Attributes

PROV Element	Attribute / Value	Description
var:object 1	u2p:typeName / var:typeName	The value var:typeName is the string with the name of the class to which var:object 1 belongs.
var:objectSM 2	prov:type / u2p:StateMachine	The value u2p:StateMachine shows that var:objectSM 2 is a state machine.
var:preObject 3	prov:type / var:typeName	The value var:typeName is the name of the class to which the object in the state var:preObject 3 belongs.
	u2p:state / var:sourceState	The value var:sourceState is the string with the name of the state var:preObject 3.
var:postObject 4	prov:type / var:typeName	The value var:typeName is the name of the class to which the object in the state var:postObject 4 belongs.
	u2p:state / var:targetState	The value var:targetState is the string with the name of the state var:postObject 4.
var:operation 5	prov:type / var:operationName	The value var:operationName is the name of the operation var:operation 5.
	tmpl:startTime / var:operationStartTime	The var:operationStartTime is an xsd:dateTime value for the start of var:operation 5.
	tmpl:endTime / var:operationEndTime	The var:operationEndTime is an xsd:dateTime value for the end of var:operation 5.

PROV relations

PROV Relation	Description
a prov:wasAttributedTo	It is the assignment of responsibility to var:object for var:objectSM.
b prov:specializationOf	var:preObject is a specialization of var:objectSM.
c prov:specializationOf	var:postObject is a specialization of var:objectSM.
d prov:wasDerivedFrom	It is the update of var:preObject resulting in var:postObject.
e prov:used	It is the beginning of utilizing var:preObject by var:operation.
f prov:wasGeneratedBy	It is the completion of production of var:postObject by var:operation.
g prov:wasInvalidatedBy	It shows that var:preObject is not longer available for use.

Discussion

- Figure B.13 only contains simple states. We do not deal with composite or submachine states, and focus only on simple states, because the former may be transformed into the

State machine diagram *Pattern 3 (StP3)*

latter by resorting to a flattening process consisting of removing composite states as well as submachine states. In fact, to flatten State Machine diagrams is a very common approach in contexts such as model checking and code generation [166]. However, the user might be interested in representing composite states directly into the PROV templates, perhaps because she/he is interested in collecting information about them, or just because she/he does not want to flatten the State Machine diagram. We can give an insight into how composite states can be mapped to PROV by placing the elements from Figure B.13 inside a Composite State **5** (see Figure B.15). A reader familiar with the UML specification will realize that the semantics of the UML representation in Figures B.13 and B.15 are different, but these semantic nuances would have no effect on the PROV transformation. The transformation of Figure B.15 is shown in Figure B.16. Both Figure B.15 and B.16 highlight the added elements by blurring the elements coming from Figure B.13 and Figure B.14, respectively. Briefly speaking, the new Composite State **6** is translated into a `prov:Entity` identified by `var:compState` **6**, which is associated with `var:objectSM` **2**, `var:preObject` **3**, and `var:postObject` **4** by means of the relations **1** `prov:specializationOf`, **h** `prov:hadMember`, and **i** `prov:hadMember`, respectively. At this point, it is also worth remarking that for this example we have used a *simple composite state* (i.e., Composite State **6**), which means that only one substate is active at a given time within such a state; but we could have used *orthogonal composite states* instead, which means that within such a state several substates are active at the same time. Note that both types of *composite states* would be translated into the same PROV template (see Figure B.16); nevertheless, the generated bindings would be different. In case of a *simple composite state*, as there can be only one active substate at the same time, there would be only one value associated with the variable `var:preObject` **3** and another value with `var:postObject` **4**. Conversely, in case of an *orthogonal composite state*, `var:preObject` **3** and `var:postObject` **4** will be associated with several values (as many as active states).

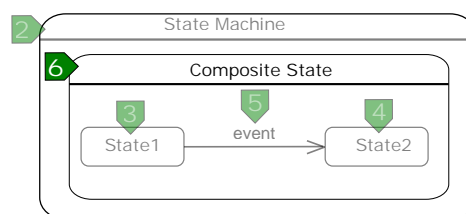


Fig. B.15 Excerpt of a UML State Machine diagram locating the UML elements from *StP3* in a *simple composite state*

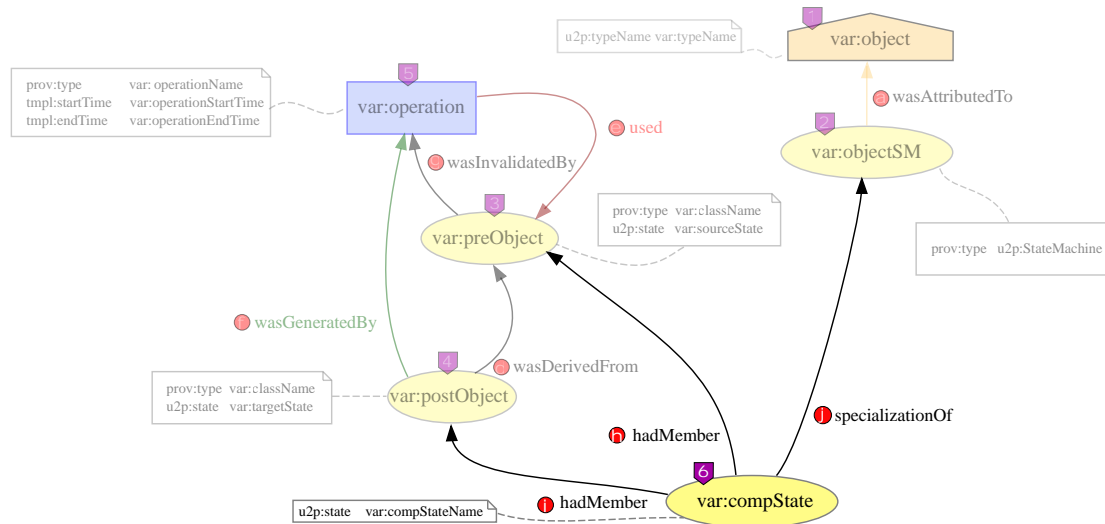


Fig. B.16 PROV template generated from the UML diagram in Figure B.15

PROV elements

UML	PROV / id	Rationale
Composite State	<code>prov:Entity</code> / <code>var:compState</code>	The Composite State is a <code>prov:Entity</code> identified by <code>var:compState</code> .

Attributes

PROV Element	Attribute / Value	Description
<code>var:compState</code>	<code>u2p:state</code> / <code>var:compStateName</code>	The value <code>var:compStateName</code> is the string with the name of the state <code>var:compState</code> .

PROV relations

PROV Relation	Description
<code>prov:hadMember</code>	It states that <code>var:preObject</code> is one of the elements in <code>var:compState</code> .
<code>prov:hadMember</code>	It states that <code>var:postObject</code> is one of the elements in <code>var:compState</code> .
<code>prov:specializationOf</code>	<code>var:compState</code> is a specialization of <code>var:objectSM</code> .

Appendix C

Sequence of interactions with GelJ

Figure C.1 depicts the sequence of interactions among the 13 substeps of GelJ used for performing the evaluation of UML2PROV. Each rounded rectangle corresponds to a task performed by the user, such a task is named by the text provided by the GelJ interface (e.g., *crop* for cropping the image). When there is a rounded rectangles without text means that there are no tasks performed in that step. Additionally, these rounded rectangles may go with a label at the top right, this label denotes the number of times that such a task has been performed. Finally, we use green arrows for specifying when the user goes to the following step (*next*), or red arrows when the user goes to the previous step (*back*).

