UNIVERSIDAD COMPLUTENSE DE MADRID
FACULTAD DE INFORMÁTICA
Departamento de Sistemas Informáticos y Computación

TESIS DOCTORAL

Mapping OCL as a Query and Constraint Language

Traduciendo OCL como lenguaje de consultas y restricciones

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Carolina Inés Dania Flores

Directores

Manuel García Clavel
Marina Egea González

Madrid, 2018

Tesis Doctoral

Mapping OCL as a Query and Constraint Language

Traduciendo OCL como
Lenguaje de Consultas y Restricciones

Carolina Inés Dania Flores

Supervisors:
Manuel García Clavel
Marina Egea González

Facultad de Informática
Universidad Complutense de Madrid

Mapping OCL as a Query and Constraint Language

Traduciendo OCL como
Lenguaje de Consultas y Restricciones

PhD Thesis

Carolina Inés Dania Flores

Advisors
Manuel García Clavel
Marina Egea González

Departamento de Sistemas Informáticos y Computación
Facultad de Informática
Universidad Complutense de Madrid

2017

A mi familia de allá.
Mi mamá Estela, mi papá Héctor
y mis hermanos Flor, Marcos y Leila.

A mi familia de acá.
Mi esposo Israel, mi perro Yiyo
y mis suegros Marinieves y Jacinto.

A mi gran amigo de allá,
mi querido Julio.

A mi gran amigo de acá,
mi querido Juli.

Acknowledgments

Undertaking this PhD has been a truly life-changing experience for me, and it would not have been possible without the support and guidance that I received from many people.

I would like to express my sincere gratitude to my supervisors, Manuel García Clavel and Marina Egea, for their support during these years. To Manuel, not only for being my advisor, but mainly for believing in me. He is responsible for giving me this great opportunity that is to pursue a PhD, letting me grow as a researcher and get to this final point. To Marina, because she has been present in every hard stage of my journey, always ready to help. It has also been an honor to be her first PhD student. Both are great advisors and friends.

I would like to thank Martin Gogolla because he was (and remains to be) my best role model of a scientist. He has been helpful in providing advice many times during my career, being always available to talk and to offer guidance. Today, I am here thanks to him.

My sincere appreciation and gratitude to Pedro R. D'Argenio for his guidance during my research. His support and inspiring suggestions have been precious for the development of this thesis.

A thank you to Nazareno Aguirre for teaching me how to become a researcher and inducing me on this path.

Thanks to David Basin for those six-months in his team at ETH Zurich. It became a very enriching experience and contributed tremendously towards my professional development.

I thank César Sánchez for adopting me as his PhD student in all those uncountable times he stepped in the office, asking me how everything was going and encouraging me to move on.

A special thanks to the IMDEA Software Institute. In particular, to Maria Alcaraz, Manuel Hermegildo, Manuel Carro and Begonia Moreno for all their support. To Juan Céspedes, Roberto Lumbreras and Gabriel Trujillo for their technical support, and to Paola Huerta, Tania Rodriguez, Carlota Gill, Andrea Iannetta, and Silvia Díaz-Plaza for all their day-to-day support in administrative tasks. Finally, to my colleagues Miriam García, Germán del Bianco, Joaquín Arias, Goran Doychev, Juan Manuel Crespo and Platon Kotzias for all those great and enjoyable lunches we had together.

Thanks to the Universidad Complutense de Madrid. In special, to Narciso Martí Oliet and Miguel Palomino Tarjuelo for their advice and for their support in all administrative tasks related to the doctoral process.

I would like to say thank to my thesis committee members for all of their guidance through this process; your discussion, ideas, and feedback have been absolutely invaluable.

I am very grateful to all the people I have met along the way and have contributed to the development of my research.

Por último, y no por ello menos importante, a todas esas personas que han estado junto a mí durante todo este tiempo:

A mí familia de allá, por quienes yo soy lo que soy. A mis viejos, por su apoyo incondicional, consejos, comprensión y amor. Me han dado todo lo que soy como persona, mis valores, mis principios, mi perseverancia y mi coraje para conseguir mis objetivos. A Marcos, Flor y a mi cuñada Leila, por estar siempre. Han estado en cada momento que los necesité y siempre me han apoyado en todas mis decisiones.

A mi familia de acá. A mi esposo Israel, que es la persona con quien comparto mi día a día, me hace felíz al despertar cada día a su lado, quien aguanta todos mis malhumores y me apoya incondicionalmente. A mis suegros, por adoptarme como un hija más, o mejor aún, como la hija que nunca habían tenido. A mi perro Yiyo quien en este úlitmo año ha estado junto a mí, hora tras hora, sentado a mi lado terminando la tesis. A mis otros suegros (por las dudas dos no fueran suficientes), Carlos y Marimar, porque ellos también me han adoptado como a una hija y están siempre alentándome con todas las decisiones que he tomado.

Al resto de la familia de allá: mi abuela Negrita, mis tíos Laly, Ricardo, Amalia, Vivian, Cacho y Cristina; mis primos hermanos Stefi, Maxi, Nico, Mili, Cande, Guillermo y Diego, porque siempre preguntan como va todo y ¡cuándo voy a terminar esta tesis! ¡He aquí la tesis! =)

A toda la familia Marguatti, por estar siempre!

A Olga y Osvaldo, los vecinos de mis abuelos; dos personas que me criaron dándome todo lo que tenían a su alcance. Su amor incondicional no se puede describir.

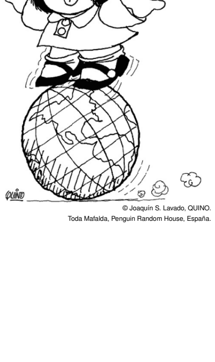
A Pedro y su familia, porque jugaron un rol muy importante en mi vida por mucho tiempo y sin su ayuda no hubiese comenzado esta tesis.

A mis amigos de allá. Los de la universidad: Julio, Caro G., el Flaco, Caro M., Waldo, Guille K. y Juli I. Mi gran amiga de la infancia: la Juli. A mis amigos que vienieron más de una vez a visitarme, Naty y Mati. Gracias a todos por estar cada vez que anduve por allá y estar siempre disponible via Skype y/o whatsapp.

A mis amigas de acá: Andie, Paola, Bego, Tania y Carlota. ¡Gracias a todas! Gracias por todos esos momentos que compartimos.

Ni de acá y ni de allá. A Javi, gracias por esos años compartidos en el equipo, todos esos viajes por Suiza, y por tantas horas de charla. A Ale (yo si te cito [86]), gracias por aguantarme en la oficina y en casa sin quejarte. A César K. por las esporádicas cenas de viernes y esas largas horas de reflexión. A Belén, mi guardavidas favorita, gracias por compartir el verano del 2011 conmigo y, de ahí en adelante, tu vida. Finalmente, a Juli, simplemente gracias. Gracias por estar siempre, en los buenos, en los malos, en los mejores y en los peores momentos.

Carolina Dania.
Amsterdam, Netherlands, 2017.



© Joaquín S. Lavado, QUINO.
Toda Mafalda, Penguin Random House, España.

Contents

Contents **v**

List of Figures **vii**

List of Tables **ix**

Abstract **xi**

Resumen **xv**

1 Introduction **1**

1.1 Model Driven Architecture (MDA) 2

1.2 Unified Modeling Language (UML) 3

1.2.1 Data models 3

1.2.2 Object models 4

1.3 Object Constraint Language (OCL) 5

1.4 Outline 5

2 Mapping OCL as a query language **7**

2.1 Procedural extensions of SQL 7

2.2 From OCL to SQL-PL 8

2.2.1 Mapping data models 9

2.2.2 Mapping OCL expressions 12

2.2.3 The SQL-PL4OCL tool 30

2.3 Benchmark 31

2.3.1 Description 31

2.3.2 Results 32

3 Mapping OCL as a constraint language **35**

3.1 From OCL to many-sorted first-order logic 35

3.1.1 Mapping data models 36

3.1.2 Mapping OCL expressions 39

3.1.3 Checking satisfiability 54

3.1.4 The OCL2MSFOL tool 54

3.2 Benchmark 58

3.2.1 Description 58

3.2.2 Results 61

4 Application domains **67**

4.1 Checking model satisfiability . . . 67

4.1.1 The eHealth record management system . . . 67

4.1.2 Checking data model satisfiability . . . 71

4.1.3 Concluding remarks . . . 72

4.2 Validating and instantiating metamodels . . . 72

4.2.1 The Core Security Metamodel (CSM) . . . 72

4.2.2 Validating the Core Security metamodel . . . 75

4.2.3 Security enhanced CSM instances . . . 76

4.2.4 Concluding remarks . . . 78

4.3 Analyzing security models . . . 78

4.3.1 SecureUML . . . 78

4.3.2 A running example . . . 80

4.3.3 Analyzing fine-grained access control policies . . . 83

4.3.4 Concluding remarks . . . 87

4.4 Analyzing privacy models . . . 87

4.4.1 Facebook: posting and tagging . . . 87

4.4.2 Modeling Facebook privacy policy . . . 90

4.4.3 Analyzing Facebook privacy policy . . . 98

4.4.4 Concluding remarks . . . 99

4.5 Checking data invariants preservation . . . 99

4.5.1 Modeling sequences of states . . . 100

4.5.2 Modeling sequences of data actions . . . 103

4.5.3 Checking data invariants preservation . . . 105

4.5.4 Concluding remarks . . . 108

5 Related work **109**

5.1 Mapping OCL as a query language . . . 109

5.2 Mapping OCL as a constraint language . . . 111

5.3 Analizing security models . . . 113

5.4 Checking data invariants preservation . . . 114

6 Conclusions and future work **115**

Bibliography **119**

List of Figures

1.1 MDA: Models and languages 2
1.2 Overview of the basic structure of MDA 3

2.1 Car-Company: A data model for a car company 10
2.2 (a) Simple Car company model. (b) Car company table. 12
2.3 Nested blocks structure in Stored Procedures 13
2.4 Sequential blocks structure in Stored Procedures 14
2.5 SQL-PL4OCL tool component architecture 30
2.6 SQL-PL4OCL tool: screen-shots 31

3.1 BasicSocNet: A data model for a basic social network. 37
3.2 OCL2MSFOL tool component architecture 55
3.3 OCL2MSFOL tool: screenshots 56
3.4 CivilStatus: A civil status model 58
3.5 WritesReviews: A writes reviews model 59
3.6 DisjointSubclasses: A disjoint subclasses datamodel 61

4.1 EHR: a data model for a basic eHealth Record Management System. 68
4.2 Automatically generated instance of the case study's data model satisfying all the invariants. 71
4.3 Core Security Metamodel 73
4.4 Automatically generated instance of the security metamodel presented in the Figure 4.3. 76
4.5 Domain Security Metamodel 77
4.6 EmplBasic: a data model for employees' information. 80
4.7 Two instances of EmplBasic 81
4.8 Empl: a SecureUML model for accessing employees' information. 82
4.9 Modeling Facebook's data model (partial). 91
4.10 Modeling a Facebook scenario. 92
4.11 EHR: a sample data model. 100
4.12 Inst_EHR: a sample object model. 101
4.13 Film(EHR,3): a filmstrip model of length 3 of EHR. 102
4.14 Three instances of EHR. 103
4.15 An instance of Film(EHR, 3). 104
4.16 EHRM case study: summary. 107

List of Tables

2.1 SQL-PL4OCL. Evaluation times. 32

3.1 Checking satisfiability of OCL constraints. 57
3.2 Analyzing CivilStatus with OCL2MSFOL 62
3.3 Analyzing WritesReviews with OCL2MSFOL 64
3.4 Analyzing DisjointSubclasses with OCL2MSFOL 64

4.1 Automatic reasoning over the examples 21-29 introduced in Section 4.3.3. . 88

5.1 Support of OCL2SQL for primitive operators 110
5.2 Support of OCL2FOL for operators over collections 111
5.3 Other mappings from UML/OCL to other formalism. 112

Abstract

This doctoral dissertation owes a great deal of its initial motivation and final focus to the very lively and insightful discussion that took place during the Dagstuhl Seminar "Automated Reasoning on Conceptual Schemas" (19-24 May, 2013) [18], which we have the fortune to participate in.

Even before attending the seminar, based on our own experience applying the model-driven development methodology within the ActionGUI project [1], we were already convinced of the truthfulness and importance of three key statements contained in the seminar's presentation, which summarize very well this dissertation's ultimate motivations:

- "The quality of an information system is largely determined early in the development cycle, i.e., during requirements specification and conceptual modeling, since errors introduced at these stages are usually much more expensive to correct than errors made during design or implementation."
- "Thus, it is desirable to prevent, detect, and correct errors as early as possible in the development process by assessing the correctness of the conceptual schemas built."
- "The high expressivity of conceptual schemas requires to adopt automated reasoning techniques to support the designer in this important task."

Among the research questions that were pursued during the seminar, we were particularly intrigued —based again on our experience within the ActionGUI project— by the following one:

- "Are the existing techniques and tools ready to be used in an industrial environment?"

The question was specifically addressed by a working group, which we were invited to join, focused "on the practical applicability of current techniques for reasoning on the structural schema". The other research questions discussed during the seminar included:

- "Does it make sense to renounce to decidability to be able to handle the full expressive power of the language used with and without textual integrity constraints?"
- "Which is the current state of the achievements as far as reasoning on the behavioral part is concerned?"
- "Which are the new challenges for automated reasoning on conceptual schemas?"

All these questions, but specially the first one, have had also an impact, in one way or another, on the shaping of our own research agenda.

The conclusions of the aforementioned working group were clear-cut: "there is still a lot of things to do for convincing the industry about the practical applicability of current

techniques for reasoning on the structural schema. (...) Having practical tools to show that all of this works was agreed to be a necessary condition for this purpose." The conclusions ended with an optimistic view about the future: "the promising results achieved so far and the existence of several prototype tools that can be applied in practice allow us to be optimistic about the achievement of this ambitious goal." Unfortunately, this view has so far proven to be overoptimistic. As it has been recently reported [76]: "Although a variety of tools exists for this purpose [model verification], the majority are academic —used as a proof of concept for the theory behind it. (...) implementations are mostly applicable to subsets of model verification tasks only. (...) the model under verification has to be manually prepared for each tool. (...) the manual work requires expert knowledge and is a source for errors." Thus, [76] continues: "most [of the tools are] far too often poorly maintained and updated (...) are using only strategies resulting in a feasibility only for few classes of problems (...) this may leave the user with a very unpleasant tool-chain. (...) additionally, most verification tools suffer from certain limitations, due to a limited focus, and out-dated underlying modeling language version or simply bugs." Finally, [76] also reports that "the long duration of the solving process remains a limiting factor in most cases."

In many ways, this doctoral dissertation is an attempt to address, the best we could and within our limited resources, the "things to do for convincing the industry about the practical applicability of current techniques for reasoning on the structural schemas", which, according to the aforementioned Dagstuhl seminar's working group, included:

- *Explanations*: "In addition to being able to check these properties, these tools should also explain the results of performing automated reasoning on the conceptual schema. (...) explanations should abstract away from whatever logic is used underneath and they should be given regarding to the model the user is referred to."

- *Benchmarks*: "Benchmarks are very important for industry. However, little attention has been paid to them in the area."

- *Scalability*: "There was a clear agreement that scalability has to be necessarily addressed to convince the industry."

In this attempt, our focus has been on creating well-founded, rigorous tools that (i) could be used by the ordinary model-driven software developers (with knowledge of UML and OCL), (ii) could seamlessly integrate with their usual modeling activities and environments, and (iii) could effectively contribute to their development of high-quality models. Notice that (i) rules out, as valid solutions in this case, tools that would require, on the part of the users, *learning a new modeling languages* or a *new logical formalism* to interact with the tool. Secondly, (ii) rules out also tools that would require, on the part of the users, *manually creating new artifacts* (e.g., input models, proofs, tactics) to interact with the tools. Finally, (iii) rules out as well tools that would not provide: universal (or, at least, very wide) coverage of the class of problems the tools are designed for; immediate (or, at least, very quick) response time; and clear and useful responses.

More constructively, as for the challenges of *explanations* and *scalability* highlighted by the Dagstuhl seminar's working group, this doctoral dissertation provides tools that cover a very wide class of the problems they were designed for. The greatest challenge here was to define an SMT-based automated reasoning tool that could handle the OCL 4-valued logic. Secondly, it provides tools with very quick response time. The challenge here was to understand sufficiently well the heuristics of the different SMT solvers so as to define a translation from OCL to first-order logic that would benefit the most, in terms of response time, from the heuristics implemented in each solver. Thirdly, it provides tools with clear and useful responses for the users. The challenge here was to understand sufficiently well the finite model finding capabilities of the different SMT solvers so as to define a translation from OCL to first-order logic that would authorize us to use these capabilities when reasoning about models, avoiding in this way useless responses of the type "unknown" from the part of the SMT solvers. Finally, as for the challenge of *benchmark*, this doctoral dissertation passes the aforementioned tools from different benchmarking exercises, using whenever possible available benchmarks, or creating new benchmarks when they were not available.

Resumen

Esta tesis doctoral debe gran parte de su motivación inicial y enfoque final a la discusión muy animada y perspicaz que tuvo lugar durante el seminario “Automated Reasoning on Conceptual Schemas” en Dagstuhl (19-24 Mayo, 2013) [18], en el cual tuvimos la fortuna de participar.

Incluso antes de asistir al seminario, sobre la base de nuestra propia experiencia aplicando la metodología de desarrollo dirigida por modelos en el proyecto ActionGUI [1], ya estábamos convencidos de la veracidad y la importancia de tres declaraciones claves contenidas en la presentación del mismo, que resumen muy bien las motivaciones finales de esta tesis:

- “La calidad de un sistema de información se determina en gran medida a principios del ciclo de desarrollo, es decir, durante la especificación de los requisitos y el modelado conceptual, ya que los errores introducidos en estas etapas suelen ser mucho más costosos de corregir que los errores cometidos durante el diseño o la implementación.”
- “Por lo tanto, es deseable prevenir, detectar y corregir errores tan pronto como sea posible en el proceso de desarrollo evaluando la corrección de los esquemas conceptuales construidos.”
- “La alta expresividad de los esquemas conceptuales requiere adoptar técnicas de razonamiento automatizadas para apoyar al diseñador en esta importante tarea.”

Entre las preguntas de investigación que se siguieron durante el seminario, nos quedamos particularmente intrigados, basados nuevamente en nuestra experiencia dentro del proyecto ActionGUI, por la siguiente:

- “Las técnicas existentes y herramienta disponibles, están preparadas para ser utilizadas en un entorno industrial?”

La pregunta fue abordada específicamente por un grupo de trabajo al que se nos invitó a unirnos. Este se centró en la aplicación práctica de las técnicas actuales de razonamiento sobre el esquema estructural”. Las otras preguntas de investigación discutidas durante el seminario incluyeron:

- “Tiene sentido renunciar a la capacidad de decisión para]manejar todo el poder expresivo del lenguaje utilizado con y sin restricciones de integridad textual?”
- “Cuál es el estado actual de los logros en lo que concierne aL razonamiento sobre el comportamiento”
- “Cuáles son los nuevos desafíos para el razonamiento automatizado en esquemas conceptuales?”

Todas estas cuestiones, pero especialmente la primera, han tenido también un impacto, de una manera u otra, en la configuración de nuestra propia agenda de investigación.

Las conclusiones del mencionado grupo de trabajo fueron claras: “todavía hay muchas cosas por hacer para convencer a la industria sobre la aplicación práctica de las técnicas actuales para el razonamiento sobre esquemas estructurales. (...) Se concluyó que tener herramientas prácticas es una condición necesaria para demostrar que todo esto funciona.”

Las conclusiones finalizaron con una visión optimista acerca del futuro: “Los prometedores resultados alcanzados hasta la fecha y la existencia de varios prototipos que pueden ser aplicados en la práctica nos permiten ser optimistas sobre la posibilidad de alcanzar este objetivo tan ambicioso”. Desafortunadamente, esta visión ha demostrado hasta ahora ser demasiado optimista. Como recientemente ha sido reportado [76]: “Aunque existe una variedad de herramientas para este propósito [verificación de modelos], la mayoría son académicas —utilizado como prueba de concepto para la teoría detrás de ella. (...) En general estas implementaciones se aplican principalmente a un subconjunto de las tareas llevadas a cabo de la verificación de modelos. (...) el modelo a verificar debe ser manualmente preparado para cada herramienta. (...) el trabajo manual requiere conocimiento experto y es fuente de errores ”. Por tanto, citegogolla-student continúa: ”la mayoría de las herramientas no suelen ser mantenidas ni actualizadas (...) sólo utilizan estrategias que resultan en la viabilidad en un número reducido de clases de problemas (...) esto puede dejar al usuario con un poco conveniente cadena de herramientas. (...) Además, la mayoría de las herramientas de verificación sufren ciertas limitaciones, debido a un enfoque limitado, y versión de lenguaje de modelado subyacente desactualizado o simplemente con errores.” Por último, [76] también informa que “la larga duración del proceso de resolución sigue siendo un factor limitante en la mayoría de los casos”.

En muchos sentidos, esta tesis doctoral es un intento para tratar, lo mejor posible y dentro de nuestros limitados recursos, las “cosas que hay que hacer para convencer a la industria sobre la aplicación práctica de las técnicas actuales de razonamiento sobre esquemas estructurales”, que según el grupo de trabajo del seminario Dagstuhl, incluyen:

- *Explicaciones*:

  “Además de poder comprobar estas propiedades, estas herramientas también deben explicar los resultados de realizar el razonamiento automatizado sobre esquemas conceptuales. (...) Las explicaciones deben abstraerse de cualquier lógica que se utilice por debajo y deben darse relacionadas con ell modelo al que el usuario es referido”.

- *Puntos de referencia*: “Los estándates comparativos son muy importantes para la industria. Sin embargo, se ha prestado muy poca atención en esta área.”

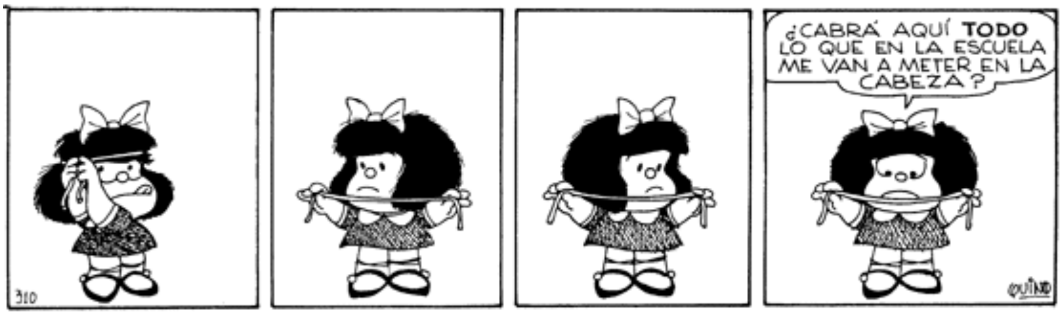
- *Escalabilidad*: “Hubo un claro acuerdo de que la escalabilidad debe ser necesariamente dirigida a convencer a la industria.”

En este intento, nuestro enfoque ha sido crear herramientas bien fundamentadas y rigurosas que: (i) que puede ser utilizado por desarrolladores de software guiado por modelos (con conocimientos de UML and OCL) , (ii) que pueda integrarse perfectamente con sus actividades habituales de modelado y entornos, y (iii) que puedan contribuir eficazmente al desarrollo de modelos de alta calidad. Observemos que: (i) esto descarta, como soluciones válidad en este caso, herramientas que requiriesen , por parte del usuario, *el aprendizajelearning de nuevos lenguajes de modelado* o un *nuevo formalismo lógico* con el fin de interactuar con la herramienta. Segundo, (ii) esto descarta también herramientas que requiriesen, por parte del usuario, *la construcción manual de nuevos artefactos* (por ejemplo., modelos de entrada, demostraciones, tácticas) para interactuar con las herramientas. Finalmente, (iii) esto descarta también herramientas que no provean: cobertura universal de las clases de problemas para los cuales las herramientas están diseñadas; tiempos de respuesta rápido; y respuestas claras y útiles.

De manera más constructiva, en cuanto a los desafíos de *explicaciones* y *escalabilidad* destacados por el grupo de trabajo del seminario Dagstuhl, esta tesis doctoral proporciona herramientas que cubren una clase muy amplia de los problemas para los que fueron diseñados. El mayor reto aquí fue definir una herramienta de razonamiento automatizado basada en SMT que pudiera manejar la lógica OCL de 4 valores. En segundo lugar, provee herramientas con tiempo de respuesta rápidos. El reto aquí fue comprender suficientemente bien la heurística de los diferentes *SMT solvers* para definir una traducción de OCL a lógica de primer orden que pudiese beneficiarse lo máximo posible, en términos de tiempo de respuesta, de las heurísticas implementadas en cada resolutor. En tercer lugar, era proporcionar herramientas con respuestas claras y útiles para los usuarios. El reto aquí fue entender suficientemente bien la capacidad de encontrar modelos finitos en los diferentes resolutores SMT para definir una traducción de OCL a lógica de primer orden que nos permitiese utilizar estas capacidades al razonar sobre modelos, evitando así respuestas inútiles del tipo ”desconocido” por parte de los *SMT solvers.* Por último, en cuanto al desafío de los *puntos de referencia*, esta tesis doctoral evalúa las herramientas anteriormente citadas con diferentes puntos de referencia, utilizando siempre que sea posible puntos de referencia existentes, y creando nuevos cuando no estaban disponibles.

Chapter 1

Introduction


¿CABRÁ AQUÍ TODO LO QUE EN LA ESCUELA ME VAN A METER EN LA CABEZA?
310
QUINO

© Joaquín S. Lavado, QUINO. Toda Mafalda, Penguin Random House, España

Model building is at the heart of system design. This is true in many engineering disciplines and is increasingly the case in software engineering. Model Driven Architecture (MDA) is a methodology for software development. It supports the development of complex software systems by generating software from models.

Software modeling has traditionally been synonymous with producing diagrams, consisting of "arrows and bubbles" with some explanatory text. But a diagram simply can not express the statements that should be part of a detailed specification. To provide the Unified Modeling Language (UML) —the 'de facto' industrial standard for software and system modeling— with the level of conciseness and expressiveness that are needed for certain aspects of system design, the UML language was extended with the Object Constraint Language (OCL).

Experience shows that even when using powerful, high-level modeling languages, like UML/OCL, it is easy to make logical errors and omissions. It is then critical not only that the modeling language has a well-defined semantics, so one can know what one is doing, but also that there is tool support for analyzing the modeled systems' properties. Within the MDA methodology, if the models do not properly specify the system's intended behavior, one should not expect the generated system to do so either.

Our research focuses on providing tool support for building complex system following the MDA methodology. In this line, the doctoral dissertation presented here provides two novel mappings for dealing with UML models (or UML-like models) that use OCL. Moreover, it discusses the applicability and benefits of these mappings with a number of non-trivial benchmarks and case studies. In a nutshell, the first mapping is a code-generator from OCL *queries* to the procedural language extensions of SQL (SQL-PL), which generates code that can be efficiently executed in the target language. The second

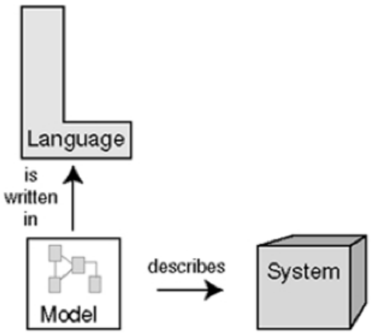


Figure 1.1: MDA: Models and languages

mapping is a translation from OCL *constraints* to many-sorted first-order logic, which generates theories whose satisfiability can be efficiently checked using Satisfiability Module Theories (SMT) solvers.

Next we provide the background and discuss the outline for this doctoral dissertion.

1.1 Model Driven Architecture (MDA)

Model Driven Architecture (MDA) [66] is a methodology for software development, defined by the Object Management Group (OMG) [71]. The key to MDA is the importance given to models in software development. MDA supports the development of complex software systems by generating software from models.

The MDA specification [66, Chapter 2] defines a model of a system as:

- A description or specification of both the system and its environment,
- in a well-defined language (graphic and/or textual),
- for a particular purpose.

Figure 1.1 describes the relationship between a model, the system that it describes, and the language in which this model is written. For MDA, the software development process consists, ultimately, in the successive transformation of models until reaching the final product. Traditionally, this process distinguishes between PIM and PSM models:

- A PIM is an independent platform model: that is, it is a model that describes a system without reference to a final concrete platform for its deployment or implantation.
- A PSM is a platform-specific model: that is, it is a model that describes a system taking into account its concrete final platform of deployment or implantation.

As in the case of models, MDA transformations between models are also written in a well-defined language, typically supported by transformation tools. Figure 1.2 describes the general process of model transformation in this methodology.

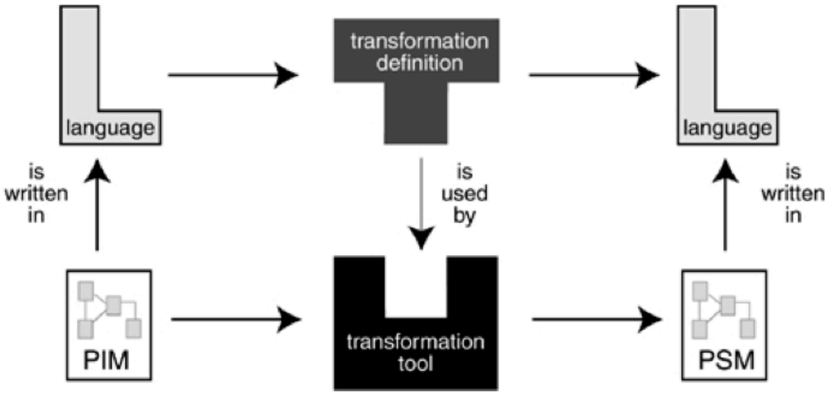


Figure 1.2: Overview of the basic structure of MDA

1.2 Unified Modeling Language (UML)

The Unified Modeling Language (UML) [67] is a visual, general-purpose language for specifying, constructing, and documenting a system's models. UML offers different types of diagrams to model different aspects or views of a system. Here we will only deal with two types of UML diagrams, namely: class diagrams, to specify data models, and object diagrams to specify object models.

1.2.1 Data models

Data models are used to model the structural view of a system. This view is static, that is, it does not describe the behavior of the system. A data model is composed of:

- **Classes.** They are used to model the objects having the same properties, relations and methods. Objects belonging to a class are caller their instances.
- **Attributes.** They are used to model the structural properties of the objects of a class. Each attribute has a name and a type, which specifies the domain of the attribute values.
- **Associations.** They are used to model the structural relationships between classes. Each connection of an association is called an association-end.
- **Multiplicities.** They are used to indicate how many instances of the class connected to an association-end can be related to an instance of the class connected to the other end of the association. In particular, multiplicity * means 0 or more instances; this is the default multiplicity for an association-end. Multiplicities can also be defined by intervals.
- **Generalizations.** They are used to model a taxonomic relationship between two classes. A generalization specializes one general class in a more specific one. Each instance of the specific class is also an instance of the general class. Thus, it has the characteristics of the general class in addition to those of its own class.

More formally,

Definition 1 *A* data model $\mathcal{D}$ *is a tuple* $\langle C, CH, AT, AS, ASO, MU\rangle$ *such that:*

- *C is a set of class identifiers.*
- *CH is a binary relation, $CH \subseteq C \times C$, where $(c, c') \in CH$ represents that c is a subclass of c', also denoted as $c \prec c'$.*
- *AT is a set of triples $\langle at, c, t\rangle$, also denoted as $at_{(c,t)}$, where at is an attribute identifier, $c \in C$, $t \in C \cup \{\mathsf{Integer}, \mathsf{String}, \mathsf{Real}\}$, and c and t are, respectively, the class and the type of the attribute at.*
- *AS is a set of tuples $\langle as, c, c'\rangle$, also denoted as $as_{(c,c')}$, where as is an association-end identifier, $c, c' \in C$, and c and c' are, respectively, the source and the target classes of as.*
- *ASO is a symmetric relation, $ASO \subseteq AS \times AS$, where $(as_{(c,c')}, as'_{(c',c)}) \in ASO$ represents that as' is the association-end opposite to as, and vice versa.*
- *MU is a set of tuples $\langle as_{(c,c')}, mu\rangle$, where $as_{(c,c')} \in AS$, and $mu \in \{\mathtt{0..1}, *\}$ represents the multiplicity of the association-end $as_{(c,c')}$.*

We assume that data models satisfy the following properties: there is no class whose identifier is Integer *or* String*; attributes and associations-ends have different identifiers; there are no cycles in the class hierarchy; and association-ends are related with exactly another association-end and with exactly one multiplicity.*

1.2.2 Object models

An object model specifies the state of a system at a particular time. The object models are mainly used for the analysis and validation of the corresponding data model.

An object model is composed of:

- **Objects.** They are instances of classes. They can have values assigned to their attributes (both their own and "inherited").
- **Links.** They are instances of associations between classes.

More formally,

Definition 2 *Let $\mathcal{D}$ be a data model $\langle C, CH, AT, AS, ASO, MU\rangle$. Then, a $\mathcal{D}$-*object model *is a tuple $\langle O, VA, LK\rangle$, such that:*

- *O is a set of pairs $\langle o, c\rangle$, where o is an object identifier and $c \in C$. Each pair $\langle o, c\rangle$, also represented as o_c, denotes that the object o is of the class c.*
- *VA is a set of triples $\langle o_c, at_{(c,t)}, va\rangle$, where $at_{(c,t)} \in AT$, $o_c \in O$, $t \in C \cup \{\mathsf{Integer}, \mathsf{Real}, \mathsf{String}\}$, and va is a value of type t. Each triple $\langle o_c, at_{(c,t)}, va\rangle$ denotes that va is the value of the attribute at of the object o.*
- *LK is a set of triples $\langle o_c, as_{(c,c')}, o'_{c'}\rangle$, where $as_{(c,c')} \in AS$, and $o_c, o'_{c'} \in O$. Each tuple $\langle o_c, as_{(c,c')}, o'_{c'}\rangle$ denotes that the object o' is among the objects that are linked to the object o through the association-end as.*

Let $\mathcal{D}$ be a data model. In what follows, we denote by $[\![\mathcal{D}]\!]$ the set of all instances of $\mathcal{D}$, i.e., the set of all the objects models of $\mathcal{D}$.

To provide UML with the level of conciseness and expressiveness that are needed for certain aspects of system design, the standard was extended with the specification of the Object Constraint Language (OCL).

1.3 Object Constraint Language (OCL)

Object Constraint Language (OCL) [68] is a textual language with a notational style similar to that of object-oriented languages. In UML 1.1, OCL appears as the standard for specifying invariants, preconditions, and postconditions. However, as of UML 2.0 the use assigned to OCL is much broader: currently, OCL is used, for example, in the definition of specific domain metamodels, model transformations, and testing and validation models.

OCL is a pure specification language: when an expression is evaluated, it simply returns a value without changing anything in the model. It is also a contextualized language: its expressions are written in a context provided by a (data) model, called the contextual model. Finally, OCL is a strongly typed language. Every OCL expression has an associated type that describes the domain of the result of that expression. OCL types can be organized into the following categories:

- **Primitive types.** They are the basic types Integer, Real, String, Boolean.
- **Class types.** They are the classes of the contextual model.
- **Types collection.** They are the parametrized types Set, Bag, OrderedSet and Sequence. Its parameters can be any other type, including collection types.
- **Special types.** They are Invalid, Void and Any types. Invalid conforms to all types except Void: the only instance of type Invalid is the value oclInvalid. Void represents a type that conforms to all types: the only instance of Void is undefined (or null). Any is the type that all other types make up.

OCL provides two constants, null and invalid, to represent *undefinedness*. Intuitively, null represents an unknown or undefined value, whereas invalid represents an error or exception. It also provides predefined operations on its different types. In particular, OCL includes operations to manipulate collections, to check properties and to generate new collections from existing collections.

Let *exp* be an OCL expression and $\mathcal{I}$ be an $\mathcal{D}$-object model. In what follows, we denote by $[\![expr]\!]^{\mathcal{I}}$ the result of evaluating *expr* in $\mathcal{I}$. Also, let Φ be a set of data invariants over $\mathcal{D}$. Then, we denote by $[\![\mathcal{D}, \Phi]\!] \subseteq [\![\mathcal{D}]\!]$ the set of all the *valid* instances of $\mathcal{D}$ with respect to Φ. More formally,

$$[\![\mathcal{D}, \Phi]\!] = \{\mathcal{I} \in [\![\mathcal{D}]\!] \mid [\![\phi]\!]^{\mathcal{I}} = \mathsf{true}, \text{ for every } \phi \in \Phi\}.$$

1.4 Outline

Chapter 2. We introduce a mapping from OCL to stored procedural SQL. In Section 2.1 we explain the basics about the target language of our mapping, namely, SQL and its procedural language extension. In Section 2.2, we provide the definitions of the mappping from OCL to SQL expressions and explain the architecture of the SQL-PL4OCL tool and how syntactic variations among the DBMS are tackled. Finally, in Section 2.3 we introduce a benchmark with the running-times obtained from evaluating examples on the different engines, and we draw conclusions.

Chapter 3. We introduce a mapping from OCL to Many-Sorted First Order Logic. In Section 3.1 we introduce our mapping from UML class diagrams and OCL constraints to MSFOL theories. Also, we discuss how to check the satisfiability of OCL constraints using

SMT solvers, present a tool, called OC2MSFOL, that supports our methodology, and provide a preliminary benchmark. Finally, in Section 3.2 we use an existing benchmark to assess OCL2MSFOL and to compare it with other tools for verifying UML/OCL models, and we draw conclusions.

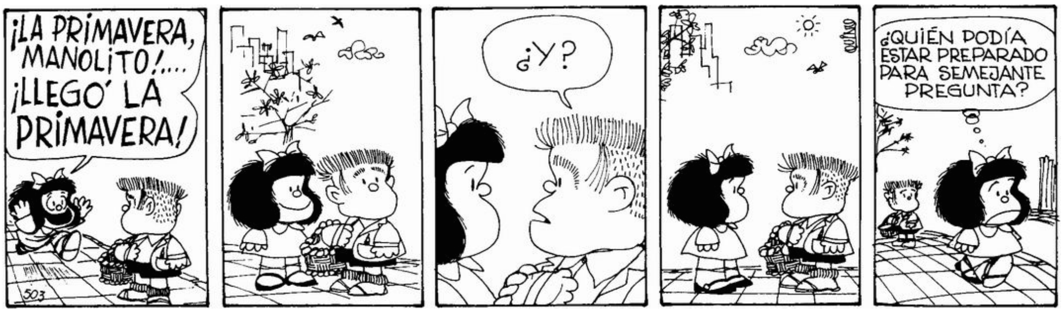
Chapter 4. We propose a set of application domains. In Section 4.1 we check model satisfiability in an eHealth record management system, while in Section 4.2 we validate and instantiate a Core Security Metamodel. In Section 4.3 we analyze security models and, in Section 4.4, privacy models. In the first case we analyze fine-grained access control policies and, in the second, we analyze Facebook posting and tagging privacy policies. Finally, in Section 4.5 we check data invariants preservation, for which we introduce the notions of sequences of states and sequences of data actions.

Chapter 5. We discuss the related work. In Section 5.1 we discuss related work related to OCL as a query language while in Section 5.2 we discuss related work for OCL as a constraint language.

Chapter 6. We draw conclusions and discuss future work.

Chapter 2

Mapping OCL as a query language


¡LA PRIMAVERA, MANOLITO!.... ¡LLEGO' LA PRIMAVERA!
503
¿Y?
QUINO
¿QUIÉN PODÍA ESTAR PREPARADO PARA SEMEJANTE PREGUNTA?

© Joaquín S. Lavado, QUINO. Toda Mafalda, Penguin Random House, España

2.1 Procedural extensions of SQL

The Structured Query Language (SQL) is a special-purpose programming language designed for managing data in relational database management systems (RDBMS). Originally based upon relational algebra and tuple relational calculus, its scope includes data insert, query, update and delete, schema creation and modification, and data access control. Accordingly, SQL commands can be divided into two: the Data Definition Language (DDL) that contains the commands used to create and destroy databases and database objects; and the Data Manipulation Language (DML) that can be used to insert, delete, retrieve and modify the data stored in databases.

Procedural extensions Although SQL is to a great extent a declarative language, it also includes procedural elements. In particular, the procedural extensions to SQL support stored procedures which are routines (like a subprogram in a regular computing language) that are stored in the database.

Currently, SQL corresponds to an ISO standard [37]. However, issues of SQL code portability between major RDBMS products still exist due to lack of full compliance with, or different interpretations of, the standard. Among the reasons mentioned are the large size and incomplete specification of the standard, as well as vendor lock-in. For our current purposes, we use as target language a procedural extension of SQL originally developed by Oracle Corporation in the early 90's, but later adopted by other RDBMS with different realizations: PL/pgSQL in PostgreSQL, stored procedures in MySQL and MariaDB, or TransactSQL (T-SQL) in SQL Server.

Stored procedures Stored procedures provide a special syntax for local variables, error handling, loop control, if-conditions and cursors, and flow control which allow the definition of iterative structures. Within stored programs, `begin-end` blocks are used to enclose multiple SQL statements, namely, to write compound statements. A block consists of various types of declarations (e.g., variables, cursors, handlers) and program code (e.g., assignments, conditional statements, loops). The order in which these can occur in a routine body is the following:

- variable and condition declarations;
- cursor declarations;
- handler declarations;
- program code.

2.2 From OCL to SQL-PL

SQL is an ISO standard [37]. However, SQL full standard is divided into several parts dealing with different aspects of the language or its processing. Also, different RDBMS implement certain syntactic variations to the standard SQL notation. Therefore, we had to adapt the implementation of our mapping to each of them. As implementation targets we selected MariaDB [58], PostgreSQL [75], and MS SQL Server [60]. Also, we kept MySQL [61] which was our first target. MariaDB and PostgreSQL were selected because they are open source and widely used by developers. MS SQL server was selected to be able to compare evaluation time from open source to commercial RDBMS. Yet, it is in our road-map to implement our mapping into other commercial engines like Oracle 12c or the Adaptive Server Enterprise/Anywhere RDBMS by Sybase, among others. Our code generator is defined recursively over the structure of OCL expressions and it is implemented in the SQL-PL4OCL tool that is publicly available at [28].

The seminal work of the mapping presented here can be found in [34, 25]. The key idea that enables the mapping from OCL iterator expressions to iterative stored procedures remains the same, but the work detailed in this chapter introduces a novel mapping from OCL expressions to SQL-PL stored procedures. The most remarkable differences are the following:

i. Each OCL expression, both non-iterator and (nested) iterator expression, is mapped into just one stored procedure.

ii. The evaluation of the source OCL expression once mapped is retrieved by executing exactly one *call-statement*. This call-statement provokes the execution of the procedure and, in particular, the execution of an SQL query written in the last part of the outermost block of the procedure that retrieves the evaluation of the OCL expression.

iii. We use temporary tables for *intermediate* and *final* values' storage. Final values' tables hold the resulting value of a query execution.

iv. We have adapted our mapping to deal with the three-valued semantics of OCL.

Decisions (i) and (ii) have facilitated the recursive definition of the code generator and simplifies its definition. Decision (iii) has significantly decreased the time required for the evaluation of the code generated. Feature (iv) enables to deal properly with the three-valued evaluation semantics of OCL. In addition, our original work and implementation was intended only for the procedural extension of MySQL, while our new definition eased the implementation of the mapping into other relational database management systems. In turn, we can now evaluate the resulting code using different RDBMS, which permits us to widen our discussion regarding efficiency in terms of *evaluation-time* of the code produced by SQL-PL4OCL tool.

2.2.1 Mapping data models

We will introduce first how we map OCL types to SQL-PL types. Second, we will detail the definition of our code generator.

OCL and SQL type systems

OCL is a contextual language which takes syntactic constructs from its contextual model. But, independently of the contextual model, the OCL type system contains the primitive types Boolean, Integer, Real and String. Our code generator maps these types to the following SQL types: `Boolean`, `Int`, `Real`, and `Varchar(250)`, respectively. When the contextual model for the OCL expressions is a structural model, like our data model, the OCL type system also contains one class type for each class specified in the class diagram.

Collection types are also present in OCL, for instance, Set, Bag, OrderedSet, and Sequence that may take as a parameter a primitive type, or a class type, e.g., Set(Integer). These types do not have a direct mapping to SQL since SQL type system does not have collection types. However, the result of an OCL query may be a collection of elements, and the execution of the code generated in SQL to translate this OCL query will also return a collection of elements. Collection of collections are also possible in OCL. These are collection types taking as parameter another collection type, for example, Bag(Set(Car)). We decided not to map collection of collections to SQL since the complexity added to our code generator would increase substantially. Also, we doubt about their utility since they are not commonly used by designers or developers. Like collection types, OCL tuple types cannot be mapped to SQL types, however, we could implement the evaluation semantics of OCL tuples by expanding the strategy that we will apply for sequence types. Namely, we could perform the evaluation of each of the n-tuples separately and ensure the allocation of each tuple evaluation result in a different table's column. Yet, we leave this discussion out of the scope of this work. Last but not least, the OCL special types, i.e., Invalid, Void, and Any do not have a counterpart in SQL either. Yet, the null value which is the unique value of the Void type, is mapped to the `null` value of SQL.

A running example

Let us now introduce a Car-Company model that we will use as our guiding example. The Car-Company model shown in Figure 3 is a data model that contains five classes: the class Car, the class Company, the class Person, and two subclasses of the latter: Employees and Customer, which are used, respectively, to distinguish among employees and customers of the company. The class Company has an association, people, to the

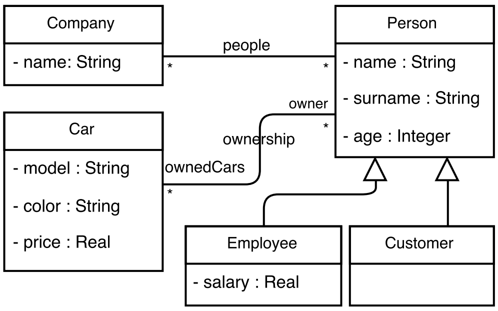


Figure 2.1: Car-Company: A data model for a car company

class Person to indicate that objects of type Company are related to objects of type Person. The classes Car and Person are related by an association to reflect that cars sold by the company may be owned by people, either customers or employees, who may also buy a car. The association is called ownership, and its association ends are, respectively, ownedCars and owner. The class Company has the attribute name of type String. The class Car has the attributes model, and color of type String, and the attribute price, of type Real. The class Person has the attributes name, surname, of type String, and age, of type Int. The class Customer inherits the attributes specified in the class Person. In addition to the attributes inherited from the class Person, the class Employee has the attribute salary of type Real.

Mapping data models and object models

Our code generator maps the underlying data and object models (i.e., the 'context' and the evaluation scenario of the OCL queries) to SQL-PL tables and records (resp.) following the next rules. Let $\mathrm{D} = \langle C, \mathit{CH}, \mathit{AT}, \mathit{AS}, \mathit{ASO}, \mathit{MU} \rangle$ be a data model and let O be a object D-object model. Then,

- **Classes.** Each class c in C is mapped to a table $nm(c)$[1], which contains, by default, a column `pk` of type `Int` as its primary key. Then, each object o in O of class type c is represented by a row in table $nm(c)$ and is identified by a unique value placed automatically in the column `pk` (> 0 and not null). This value is also automatically incremented ($+1$) each time a new row is inserted.

- **Class attribute.** Each attribute $\langle at, c, t \rangle \in \mathit{AT}$ is mapped to a column $nm(at)$ of table $nm(c)$, being the type t, according to the rules for mapping types that we introduced at the beginning of this section. Then, the value of at for an object o, instance of class c, is mapped to the value held by the column $nm(at)$ for the record that is identified by the `pk` value assigned to o in table $nm(c)$.

- **Association.** Each association $\langle as, c, c' \rangle \in \mathit{AS}$ is mapped to a junction table $nm(as)$, which contains two columns $nm(rl\_c)$ and $nm(rl\_c')$, both of type `Int`. Then, each link $\langle o, as, o' \rangle \in \mathit{LK}$ is represented by a row in table $nm(as)$, where $nm(rl\_c)$ holds the key denoting o and $nm(rl\_c')$ holds the key denoting o' as foreign keys' references.

[1] $nm()$ generates unique names for classes, attributes, and associations.

For one-to-many associations, we add a foreign key column on the table corresponding to the class in the *many-side* of the relationship. This column holds the key value referencing the object linked in the *one-side* of the association.

- **Inheritance.** Each class c, subclass of a class c', is mapped to a table $nm(c)$ together with its direct (i.e., not inherited) attributes and associations following the definitions described above. But, in addition, a foreign key column, `fk`, is added to $nm(c)$ referencing the primary key column of the table $nm(c')$ that maps class c'.

  Although it is not completely obvious, this definition is controlling how tables which correspond to classes related by inheritance are populated. We avoid discussing it further here since it would add a complexity that is not of direct value to the presentation of our code generator. Yet, we provide examples next that will help to understand the rationale behind our definition. The interested reader can find the details in [25].

Mapping our running example

From now on we will choose MariaDB (fully compatible with MySQL) syntax to illustrate the code generated by our mapping, both for the definitions and the examples.

The command that is automatically generated to map the class Person to a SQL table is:

```
create table Person (
 pk int not null primary key auto_increment,
 name varchar(250),
 surname varchar(250),
 age int);
```

Similarly, the classes Car and Company are mapped to tables.

The command that is automatically generated to map the class Employee to a SQL table is:

```
create table Employee (
 pk int not null primary key auto_increment,
 salary int,
 fkPerson int,
 foreign key (fkPerson) references Person(pk));
```

Similarly, the class Customer is mapped to a table.

The command that is automatically generated to map the association ownership to a SQL table is:

```
create table ownership (
 owner int,
 ownedCars int,
 foreign key (owner) references Person(pk),
 foreign key (ownedCar) references Car(pk));
```

Similarly, the association people is mapped to a table.

Please, notice that in the structure of the tables that we create for the subclasses Employee (and Customer), the subclasses hold an additional column `fkPerson` as a foreign key to the primary key of the table `Person` that corresponds to their parent class.

Car
–model : String
pk
model

(a) (b)

Figure 2.2: (a) Simple Car company model. (b) Car company table.

2.2.2 Mapping OCL expressions

In what follow, we briefly introduce the novel structure of the code produced by our SQL-PL generator for OCL expressions. This section is intended to help the understanding of our mapping definition in the following section. For any input OCL expression, our code generator always produces a stored procedure that can be invoked using a call statement, as we explain next.

Given an OCL expression *exp*, our code generator *patternproc*(*exp*) generates the following pattern.

```
create procedure nm(exp)()    1
begin                         2
  codegen_b(exp)              3
  codegen_q(exp);             4
end;//                        5
call nm(exp)//                6
```

The generated code contains the declaration of the stored procedure (lines 2-5), headed by its creation command and name (line 1). The main block is enclosed by the delimiters `begin-end`. The code contained by the main block is generated by the auxiliary functions *codegen*$_b$(*exp*) and *codegen*$_q$(*exp*) (lines 3-4). These functions generate code that mirrors the structure of the OCL expressions. Finally, the function *patternproc*(*exp*) also generates a `call`-statement to execute the stored procedure (line 6).[2]

In what follow, we will explain two kind of expressions: Simple and Complex. Simple expressions are the expressions that does not need any auxiliary block definition within the stored procedure to be mapped. While complex expressions need an auxiliary block definition within the stored procedure to be mapped.

In particular, `begin-end` blocks have the features that are particularly useful for our work:

- `begin-end` blocks can be nested;
- variables declared in outer `begin-end` blocks are visible in the inner blocks at any level of depth.

Both of these features are crucial in our mapping to easily and recursively map OCL expressions that contain nested operators expressions. Figure 2.3 gives an idea of the structure that nested blocks adopt within stored procedures. Another case is OCL sequential operators; in such case, these are mapped into sequential blocks. Figure 2.4 gives an idea of the structure that sequential blocks adopt within stored procedures. Furthermore, we can have a combination of sequential and nested operators, in that case, the stored procedure will have a combination of sequential and nested blocks. Finally, to invoke a stored procedure, we use the `call` statement; i.e. the routines showed in the Figure 2.3 or Figure 2.4, are invoked by the following statement:

[2] Please, note that our delimiter in SQL-PL is set to '//'.

```
call program_name
```

Simple expressions. In this case, the function $codegen_b(exp)$ does not generate any code. Examples of this kind of expressions are operators over classes, operators between sets or bags, math operators, etc.

Example 1 *The code generated by* $patternproc(exp)$ *for the expression* $exp =$ Car.allInstances() *is:*

```
create procedure carallinstances()
begin
  codegen_q(exp);
end;//
call carallinstances//
```

Where $codegen_q(exp)$ *generates the following specific code:*

```
select Car.pk as val from Car
```

Note that when the stored procedure is executed, the result is a table containing a column called `val`*, which holds all the values of the column* `pk` *(primary key) from the records of table* `Car`*.* □

Example 2 *Consider now the expression* $exp =$ Car.allInstances().model*. The code generated by* $patternproc(exp)$ *is:*

```
create procedure modelallinstances()
begin
 codegen_q(exp);
end;//
call modelallinstances//
```

Where $codegen_q(exp)$ *generates the following specific code:*

```
select Car.model as val
from (select pk as val from Car) as t0
left join Car on Car.pk = t0.val
```

```
create procedure program_name()
  begin
    begin
      begin
        ...
      end;
    end;
  ...
end;
```

Figure 2.3: Nested blocks structure in Stored Procedures

Note that when the stored procedure is executed, the result is a table containing a column called `val`*, which holds all values of the column model from the records of the* Car *table.* □

Example 3 *Consider the following OCL expression exp, exp = exp_1*->notEmpty()*, where exp_1 is an expression which does not contain any operator subexpression that requires a block definition, then patternproc(exp) generates the following code:*

```
create procedure exp1notEmpty()
begin
  select count(*) > 0 as val from ($codegen_q(exp_1)$) as t1;
end;//
call exp1notEmpty//
```

□

In what follows, we will see how our code generator can recursively deal with the recursive structure of OCL expressions.

Complex expressions. For the other cases which the function $codegen_b(exp)$ does generate code because mapping a given expression, *exp*, needs of an auxiliary block definition. This auxiliary block is required either for the expression to be properly mapped or because we have noticed that it brings efficiency to the execution. For example, in some cases we noticed that executing a given sequence of operations within a block required less time than executing a given SQL query, and we tailored our mapping accordingly. We consider occurrences of complex expressions to operators over sequences, iterators, etc. Next, we sketch the idea of our mapping in these cases and provide examples.

Sequence Operators. Let *exp* be a sequence expression. Let the shape of this expression be $op(exp_1, \ldots, exp_n)$ and consider that the subexpressions $exp_1, \ldots, exp_n$ need to be mapped into blocks too. Then, $codegen_b(exp)$ generates the SQL-PL blocks:

```
begin
$codegen_b(exp_1)$
...
$codegen_b(exp_n)$

create procedure program_name()
  begin
    begin
      ...
    end;
    ...
    begin
      ...
    end;
    ...
end;
```

Figure 2.4: Sequential blocks structure in Stored Procedures

```
drop table if exists nm(codegen_b(exp));
create temporary table nm(codegen_b(exp))
 (pos int not null auto_increment, val basictype(exp), primary key(pos));
insert into nm(codegen_b(exp))(val) (codegen_q(exp_1));
...
insert into nm(codegen_b(exp))(val) (codegen_q(exp_n));
end;
```

while, $codegen_q(exp)$ generates:

```
select * from nm(codegen_b(exp));
```

Note that $basictype(tp)$ is the SQL type associated to the UML type tp.

Example 4 *Consider now the expression exp =* 'hi'.characters().union('ho'.characters())*. Then, the code generated by $patternproc(exp)$ is:*

```
create procedure unionLits()
begin
 codegen_b(exp)
 codegen_q(exp);
end;//
call unionLits//
```

Where $codegen_b(exp)$ generates the following specific code:

```
begin
-- sub-block 'hi'.sequence()
begin
drop table if exists wchars;
create temporary table wchars
  (pos int not null auto_increment, val varchar(250), primary key(pos));
insert into wchars(val) (select 'h' as val);
insert into wchars(val) (select 'i' as val);
end;
-- sub-block 'ho'.sequence()
begin
 drop table if exists w1chars;
 create temporary table w1chars
   (pos int not null auto_increment, val varchar(250), primary key(pos));
 insert into w1chars(val) (select 'h' as val);
 insert into w1chars(val) (select 'o' as val);
end;
-- code for operator union
drop table if exists unionLits;
create temporary table unionLits(val varchar(250));
insert into unionLits(val)
  (select wchars.val as val from wchars as t1 order by wchars.pos asc);
insert into unionLits(val)
  (select w1chars.val as val from w1chars as t2 order by w1chars.pos asc);
end;
```

While $codegen_q(exp)$ generates the following specific code:

```
select * from unionLits
```

□

Note that when a stored procedure is executed to evaluate an expression of Sequence type, the result is stored in a table containing two columns called `pos` and `val`, which holds all values (in the column val) ordered by the position given in the column pos.

Iterator expressions. These expressions are of the form *src*->*iterOp*(*v*|*body*) whose top-operator is an iterator operator.[3] For each iterator expression *exp*, our code generator produces a stored procedure composed of an iterative *block* and a *query* following the structure introduced at the beginning of the section.

When the stored procedure is called, it

1. creates a temporary table;
2. executes, for each element in the *src*-collection that is instantiating the iterator variable *v* the *body* of the iterator expression;
3. processes and stores in the temporary table, created in Step 1, the result of the query $codegen_q(body)$, according to the semantics of the iterator operator.

The function $codegen_q(exp)$ generates a query that retrieves the values corresponding to the evaluation of *exp* from the table that has been created and filled in during the execution of the iterative block of the stored procedure. Finally, as we shown before, the function *patternproc*(*exp*) also generates a `call`-statement to actually execute the procedure *patternproc*(*exp*).

Example 5 *Iterator expressions. Consider the expression exp =* Car.allInstances()->select(u|u.model='BMW'). *The code generated by patternproc(exp) is:*

```
create procedure selectproc()
begin
 codegen_b(exp)
 codegen_q(exp);
end;//
call selectproc//
```

Where $codegen_b(exp)$*, generates the following specific code:*

```
begin
 declare done int default 0;
 declare var int;
 declare crs cursor for (select pk as val from Car);
 declare continue handler for sqlstate '02000' set done = 1;
 drop table if exists selectproc;
 create temporary table selectproc(val int);
 open crs;
 repeat
```

[3] For the sake of simplicity, we will consider here that the top-operator of *src* is a *simple* expression. The case when the iterator expressions are nested deserve, however, a particular attention.

```
  fetch crs into var;                                    10
  if not done then
    if exists                                            12
      (select True from
        (select model = 'BMW' as val                     14
        from Car where pk = var) as t1) as t2
    then                                                 16
     insert into selectproc(val) values (var);
  end if;                                                18
  until done end repeat;
 close crs;                                              20
end;
```

The definition of the block (line 1-21) contains the following declarations: first some variables are declared (lines 2-5); following Step 1, a new temporary table is created (note that it is deleted if it exists) (lines 6-7); following Step 2, for each element of the source (lines 9-10), the value of the result of the execution of the body is calculated; however, following Step 3, this value is only inserted into the new table (line 17) if the condition of the body is satisfied (lines 11-20), according to the semantics of the iterator operation.

Finally, $codegen_q(exp)$ *generates the following specific code:*

```
select val from selectproc
```

Note that, as it happened for Example 1, the result of the execution of the stored procedure is a table containing a column called `val`*, which holds all records of the table Car whose model is 'BMW'.* □

To conclude, let us say that the potential complexity of the OCL expression is mirrored within the stored procedure by using the function $codegen_b(exp)$. Within such procedure, the general idea that drives the mapping of OCL complex expressions is that OCL sequential operators are mapped to sequential blocks, and OCL nested operators are mapped to nested blocks. In addition, there will always be an outermost `begin-end` enclosing block that contains the query to retrieve the evaluation result when the procedure is invoked.

Scope

We do not cover yet completely the whole OCL language. However, we cover most of the operators listed in the OCL standard library [68, Chapter 11]. More concretely, we cover operators on primitive types String, Boolean, Integer and Real; operators on Set, Bag and Sequence types; and all iterator operators except orderBy and closure. Last but not least, we do cover nested iterator expressions, i.e., iterator expressions whose body also contains iterator expressions, for example, Person.allInstances()–>forAll(p|Car.allInstances()–>exists(c|p.ownedCars–>includes(c))). We will deal in detail with this type of expression in the following section. Yet, we do not support tuples or nested collections. Finally, we neither support static collections of AnyType, and we have to refer the null value explicitly, i.e. null::String.

In the following two subsections, we take advantage of the explanation about the structure of the code generated in this subsection that will allow the reader to understand more easily the definition of our mapping. Below, we provide the mapping definition for those operations from the OCL standard library [68, Chapter 11] that we have considered more illustrative. The exhaustive definition of the mapping for all the operations of the

OCL standard library is provided in [28]. We start each definition with the name of the operator, followed by a brief description of its semantics, and the definition of its mapping.

Mapping simple OCL expressions

In this section we show how we define our mapping for simple expressions. Recall from the previous section that these are expressions for which the top operator is mapped directly to a SQL query without the need of declaring auxiliary SQL-PL blocks. Fall within this category model specific operators, boolean, numeric, and collection operators for sets and bags.

Model specific operators There are operations in OCL that the language 'borrows' from the contextual model. These operations vary when the contextual model changes and they refer to association ends, classes' attributes and classes' identifiers.
In the following, we consider $\mathit{exp1}$ to be an OCL expression of type class, or (not ordered) set or bag.

allInstances(). It returns all the instances of the class that it receives as argument. Let exp be an expression of the form C.allInstances(), where C is a class of the contextual model. Then, $codegen_q(exp)$ is the following SQL query:

select $nm(C)$.pk **as** val **from** $nm(C)$

Attribute Expression. It retrieves an attribute's values of the instances returned by the source expression.

Let exp be an expression of the form $\mathit{exp1}.attr$ where $attr$ is an attribute of a class A. Then, $codegen_q(exp)$ is the following SQL query:

select $nm(A).nm(attr)$ **as** val
from $(codegen_q(\mathit{exp1}))$ **as** $al(codegen_q(\mathit{exp1}))$
left join $nm(A)$ **on** $al(codegen_q(\mathit{exp1}))$.val = $nm(A)$.pk

Note that $al()$ generates a unique alias names for tables.

Association-End Expression. It retrieves the instances linked to the objects returned by the source expression through the association end.

Let exp be an expression of the form $\mathit{exp1}\,.\,rl\_A$ (resp. $\mathit{exp1}\,.\,rl\_B$), where rl_A (resp. rl_B) is the A-end (resp. B-end) of an association as between two classes A and B. Then, $codegen_q(exp)$ is the following SQL query:

select $nm(at).nm(rl\_A)$ **as** val
from $(codegen_q(\mathit{exp1}))$ **as** $al(codegen_q(\mathit{exp1}))$
left join $nm(as)$ **on** $al(codegen_q(\mathit{exp1}))$.val = $nm(as).nm(rl\_B)$
where $nm(as).nm(rl\_A)$ **is not null**

In all cases previously described, the top expression exp does not require any block definition. Thus $codegen_b(exp)$ consists only of the blocks that might be required by its subexpression:

$codegen_b(\mathit{exp1})$

Example 6 *Model specific operators The following examples do only generate SQL queries. None of them need blocks for their definition, i.e.,* $codegen_b(exp)$ *is empty in all cases.*

Q1. *Query the ages of all employees.* Employee.allInstances().age

```
select Person.age as val
from (
 select fkEmployee as val
 from (select pk as val from Employee) as t0
 left join Employee on t0.val = Employee.pk) as t1
left join Person on t1.val = Person.pk
```

Notice that since Employee *is a subclass of* Person *that inherits from it the attribute* age*, we recover with the SQL query the column age of the table* `Person`*, but only for the rows contained by the table* `Employee`*. This is enforced by the left join used to align the foreign keys contained by the table* `Employee` *with the keys contained by the table* `Person`*.*

Q2. *Query the cars owned by all persons.* Person.allInstances().ownedCars

```
select ownership.ownedCars as val
from (select pk as val from Person) as t0
left join ownership on t0.val = ownership.owner
where ownership.ownedCars is not null
```

□

Boolean Operators In all cases described below, the top expression *exp* does not require any block definition. Thus $codegen_b(exp)$ consists only of the blocks that might be required by its sub-expression:

$codegen_b(exp_1)$

isEmpty(). It returns 'true' if the source collection is empty, and 'false' otherwise. Let *exp* be an expression of the form exp_1->isEmpty(). Then, $codegen_q(exp)$ is the following SQL query:

select count(*) = 0 **as** val
from ($codegen_q(exp_1)$) **as** $al(codegen_q(exp_1))$

The operator isEmpty does not require any block definition, thus $codegen_b(exp)$ is composed by the blocks of its subexpression (if any):

$codegen_b(exp_1)$

For the operator notEmpty, '>' replaces '=' in the above SQL query.

includes. It returns 'true' if the source collection exp_1 contains the element *exp*.

Let *exp* be an expression of the form exp_1->includes(exp_2). Then, $codegen_q(exp)$ is the following SQL query:

select $codegen_q(exp_2)$ **in** $codegen_q(exp_1)$ **as** val

The operator includes does not require any block definition, thus $codegen_b(exp)$ is composed by the blocks of its subexpressions (if any):

$codegen_b(exp_1)$
$codegen_b(exp_2)$

For the operator excludes, '`not in`' replaces '`in`' in the above SQL query.

Example 7 *Boolean operators. The following examples only need to generate SQL queries. None of them require any block definition, i.e.,* $codegen_b(exp)$*, in all cases, is empty.*

Q3. *Query whether there are 'BMW' cars in the company.*
Car.allInstances().model->includes('BMW')

```
select (select 'BMW' as val) in
 (select Car.model as val
 from (select Car.pk as val from Car) as t0
 left join Car on t0.val = Car.pk) as val
```

□

Numeric operators Again, for all cases described below, the top expression *exp* does not require any block definition. Thus $codegen_b(exp)$ consists only of the blocks that might be required by its sub-expression:

$codegen_b(exp_1)$

size. It returns the size of the source collection. Let *exp* be an expression of the form exp_1->size(). Then, $codegen_q(exp)$ is the following SQL query:

```
select count(*) as val
from (codegen_q(exp_1)) as al(codegen_q(exp_1))
```

sum. It returns the sum of the elements in the source collection that must be of numeric type. Let *exp* be an expression of the form exp_1->sum(). Then, $codegen_q(exp)$ is the following SQL query:

```
select sum(*) as val
from (codegen_q(exp_1)) as al(codegen_q(exp_1))
```

Example 8 *Numeric operators. The following examples do only generate SQL queries. None of them need blocks for their definition, i.e.,* $codegen_b(exp)$ *is empty in all cases.*

Q4. *Count the number of customers.* Customer.allInstances()->size()

```
select count(*) as val
from (select Customer.pk as val from Customer) as t0
```

□

Collection operators for Set and Bag types asSet. The set containing all the elements from the source collection, with duplicates removed (if any). Let *exp* be an expression of the form exp_1->asSet(). Then, $codegen_q(exp)$ is the following SQL query:

```
select distinct al(codegen_q(exp_1)).val as val
from (codegen_q(exp_1)) as al(codegen_q(exp_1))
```

union. It returns the set union (resp. multiset union) of both sets (resp. bags) passed as arguments to the operation. Let *exp* be an expression of the form exp_1->union(exp_2), where both exp_1 and exp_2 are sets. Then, $codegen_q(exp)$ is the following SQL query:

select $al(codegen_q(exp1))$.val
from ($codegen_q(exp2)$ union $codegen_q(exp1)$) **as** $al(codegen_q(exp1))$

When *exp1* or *exp2* are bags, then '`union all`' will replace '`union`' in the above SQL query. The operator including that returns the bag containing all elements of the source collection *exp1* plus the element *exp2* passed as argument is mapped exactly as the operator union is.

excluding. It returns the bag that results from removing the element *exp2* from the source collection *exp1*. Let *exp* be an expression of the form *exp1*–>excluding(*exp2*). Then, $codegen_q(exp)$ is the following SQL query:

select $al(codegen_q(exp1))$.val
from ($codegen_q(exp1)$) **as** $al(codegen_q(exp1))$
where $al(codegen_q(exp1))$.val **not in** $codegen_q(exp2)$

includesAll. It returns 'true' if the collection *exp1* contains all the elements in the collection *exp2*, and 'false' otherwise. Let *exp* be an expression of the form *exp1*–>includesAll(*exp2*). Then, $codegen_q(exp)$ is the following SQL query:

select count($al(codegen_q(exp2))$.val) = 0 **as** val
from ($codegen_q(exp2)$) **as** $al(codegen_q(exp2))$
where $al(codegen_q(exp2))$.val **in** ($codegen_q(exp1)$)

The operator excludesAll returns 'true' if the collection *exp1* does not contain all the elements in the collection *exp2*, and 'false' otherwise. For the operator excludesAll, '`not in`' replaces '`in`' in the above SQL-PL statement.

In all cases previously described, the expression *exp* does not require any block definition. Thus $codegen_b(exp)$ consists only of the blocks that might be required by its subexpressions:

$codegen_b(exp1)$
$codegen_b(exp2)$

Example 9 *Collection Operators. The following examples do only generate SQL queries. None of them need blocks for their definition, i.e.,* $codegen_b(exp)$ *is empty in all cases.*

Q5. *Query the surnames of all customers but those whose surname is 'Smith'.*
Customer.allInstances().surname–>excluding('Smith')

```
select t2.val
from
 (select Person.surname as val
 from
  (select fkCustomer as val
  from (select pk as val from Customer) as t0
  left join Customer on t0.val = Customer.pk) as t1
 left join Person on t1.val = Person.pk) as t2
where t2.val not in (select 'Smith' as val)
```

□

Mapping complex OCL expressions.

In this section we introduce the mapping definition for those top operators whose definition needs to generate both SQL queries and blocks. Namely, sequence and iterator operators.

Sequence Operators In OCL there is an operation for building a sequence from a set or a bag of elements. This operation is asSequence(). Remember that, when we talk about a sequence in OCL we talk about a collection of elements that are assigned a position in a list. Sequences allow for duplicated elements.

asSequence(). Let *exp* be an expression of the form *exp1*.asSequence().
Then, $codegen_b(exp)$ generates the SQL-PL blocks:

```
begin
 drop table if exists $nm(codegen_b(exp))$;
 create temporary table $nm(codegen_b(exp))$ (val varchar(250));
 insert into $nm(codegen_b(exp))$(val)
 select $al(codegen_q(exp_1))$.val as val as
  from ($codegen_q(exp_1)$) as $al(codegen_q(exp_1))$;
end;
```

while, $codegen_q(exp)$ generates:

```
select pos, val from $nm(codegen_b(exp))$
```

Example 10 *Sequence Operators.*

Q6. *Query the length of a sequence that contains all instances of Person.* Person.allInstances()->asSequence()->size()

```
begin
drop table if exists personAsSequence;
create temporary table personAsSequence
 (pos int not null auto_increment, val int, primary key(pos));
insert into personAsSequence(val)
 select t0.val as val as from (select pk as val from Person) as t0;
end;
select count(*) as val from (select * from personAsSequence) as t1;
```

□

Mapping OCL iterator expressions Since the semantics of each OCL iterator operator can be defined through a mapping from the iterator to the iterate construct, we could have decided to translate the iterate expressions resulting from those mappings in order to generate code for the iterator operations like reject, select, forAll, exists, collect, one, sortedBy, isUnique and any by applying the iterate pattern. In fact, this was the decision made for the definition of the SQL-PL4OCL code generator in [87], however they did not succeed in finding a pattern to map the iterate expressions and therefore the iterator expressions were not mapped either. Instead, we decided to generate code specifically for each iterator operator according to its semantics. In this way, we can generate code that is less complex and more tailored to the semantics of each iterator operator. Also this decision allows us, as we explain below, to end a block at an intermediate iteration step once the evaluation result of the translated iterator is clear. For instance, when the execution of the code generated to map the body of a forAll expression returns false at one iteration step, the procedure is terminated returning false.

The basic idea is therefore that, for each iterator expression *exp*, our code generator produces a SQL-PL block that, when it is called creates a temporary table, denoted by $nm(codegen_b(exp))$, from which we obtain using a simple `select`-statement the values corresponding to the evaluation of *exp*. For now, we assume that the types of the *src*-subexpressions are either sets or bags of primitive or class types.

Let $exp = src \rightarrow iter\_op(var|body)$ be an iterator expression. Then, $codegen_q(exp)$ is the following SQL query:

```
select * from $nm(codegen_b(exp))$;
```

While, $codegen_b(exp)$ generates the following scheme of SQL-PL blocks:

```
$codegen_b(src)$
begin                                                                  2
declare done int default 0;
declare $var$ cursor-specific type ;                                   4
declare crs cursor for ($codegen_q(src)$);
declare continue handler for sqlstate '02000' set done = 1;            6
drop table if exists $nm(codegen_b(exp))$;
create temporary table $nm(codegen_b(exp))$ (val value-specific type );   8
Initialization-specific code (only for forAll, one, exists and sortedBy)
open crs;                                                              10
repeat
 fetch crs into $var$;                                                 12
 $codegen_b(body)$
 if not done then                                                      14
  Iterator-specific processing code
 end if;                                                               16
until done end repeat;
close crs;                                                             18
End-specific code (only for isUnique)
end;                                                                   20
```

Basically, $codegen_b(exp)$ generates a block [lines 2–20] which creates the temporary table $nm(codegen_b(exp))$ [line 8] and execute, for each element in the *src*-collection [lines 5,10-12], the *body* [line 13] of the iterator expression *exp*. More concretely, until all elements in the *src*-collection have been considered, $codegen_b(exp)$ repeats the following process: (*i*) it instantiates the iterator variable *var* in the *body*-subexpression, each time with a different element of the *src*-collection, which it fetches from $codegen_q(src)$ using a cursor [lines 12–14]; and (*ii*) using the so called "iterator-specific processing code", it processes in $nm(codegen_b(exp))$ the result of the query $codegen_q(body)$, according to the semantics of the iterator *iter_op* [line 15]. In addition, in the case of the four iterators: forAll, one, exists and sortedBy, the table $nm(codegen_b(exp))$ is initialized, using the so called "initialization-specific code" [line 9], and in the case of the iterator isUnique, an "end-specific code" is required. Moreover, for the iterators forAll and exists, the process described above will also be finished when, for any element in the *src*-collection, the result of the query *codegen*$_q$(*body*) contains the corresponding value, in the case of the iterator forAll, to False or, in the case of the iterator exists, to True.

In the remaining of this subsection, we specify, for each case of iterator expression, the corresponding "value-specific type", "initialization-specific code", "iterator-specific processing code", and "end-specific code" produced by our code generator when instantiating the general schema. Again, for all cases, the "cursor-specific type" is the SQL-PL type which represents, according to our mapping (see section 2.2.1), the type of the elements in the *src*.

forAll-iterator. Let *exp* be an expression of the form *src*–>`forAll`(*var*|*body*). This operation returns 'true' if *body* is 'true' for all elements in the source collection *src*. The "holes" in the scheme *codegen*$_b$(*exp*) will be filled as follows:

- *value-specific type:* boolean.
- *Initialization code:*

```
insert into nm(codegen_b(exp)) (val) values (True);
```

- *Iteration-processing code:*

```
update nm(codegen_b(exp)) set val = False
where (codegen_q(body)) = False;
if exists
  (select True from nm(codegen_b(exp)) where val = False)
then set done = 1;
end if;
```

exists-iterator. Let *exp* be an expression of the form *src*–>`exists`(*var*|*body*). This operation returns 'true' if *body* is 'true' for at least one element in the source collection *src*. The "holes" in the scheme *codegen*$_b$(*exp*) will be filled as follows:

- *value-specific type:* boolean.
- *Initialization code:*

```
insert into nm(codegen_b(exp)) (val) values (False);
```

- *Iteration-processing code:*

```
update nm(codegen_b(exp)) set val = True where (codegen_q(body)) = True;
if exists (select True from nm(codegen_b(exp)) where val = True)
  then set done = 1;
end if;
```

one-iterator. Let *exp* be an expression of the form *src* –>`one`(*var*|*body*). This operation returns 'true' if *body* is 'true' for exactly one element in the source collection *src*. The "holes" in the scheme *codegen*$_b$(*exp*) will be filled as follows:

- *value-specific type:* boolean.
- *Initialization code:*

```
insert into $nm(codegen_b(exp))$(val) values (False);
set @counter = 0;
```

- *Iteration-processing code:*

```
if exists
(select $nm(codegen_b(body))$.val
 from ($codegen_q(body)$) as $nm(codegen_b(body))$
 where $nm(codegen_b(body))$.val = True)
then
 set @counter = @counter+1;
 update $nm(codegen_b(exp))$ set val = True;
end if;
if @counter = 2 then
 update $nm(codegen_b(exp))$ set val = False;
 set done = 1;
end if;
```

sortedBy-iterator. According to [68], it results in the OrderedSet containing all elements of the source collection ordered in descending order according to the values returned by the evaluation of the body expression. The order considered is given by the operation $<$ that should be defined on the type of the body expression. We consider instead the order given by the operation $\leq$ in order to be able to include in the resulting ordered set those elements for which the evaluation of the body returns exactly the same value.

Let *exp* be an expression of the form *src*->sortedBy (*var*|*body*). This operation returns the collection of elements in the *src* expression ordered by the criterion specified by *body*. The "holes" in the scheme $codegen_b(exp)$ will be filled as follows:

- *value-specific type:* the SQL type which represents, according to our mapping, the type of the *body*.
- *Initialization code:*

```
create temporary table $nm_{seq}(codegen_b(exp))$
 (pos int not null auto_increment, val $\boxed{\textit{value-specific type}}$);
```

- *Iteration-processing code:*

```
insert into $nm(codegen_b(exp))$(val) $codegen_q(body)$;
insert into $nm_{seq}(codegen_b(exp))$(val)
 (select val from $nm(codegen_b(exp))$ order by val desc);
```

collect-iterator. Let *exp* be an expression of the form *src*->collect(*var*|*body*). This expression returns the collection of objects that result from evaluating *body* for each element in the source collection *src*. The "holes" in the scheme $codegen_b(exp)$ will be filled as follows:

- *value-specific type:* the SQL-PL type which represents, according to our mapping, the type of the *body*.
- *Iteration-processing code:*

```
insert into $nm(codegen_b(exp))$(val) $codegen_q(body)$;
```

select-iterator. Let exp be an expression of the form src->select($var|body$). This expression returns a subcollection of the source collection src containing all elements for which $body$ evaluates to 'true'. The "holes" in the scheme $codegen_b(exp)$ will be filled as follows:

- *value-specific type:* the SQL-PL type which represents, according to our mapping, the type of the elements in the src.
- *Iteration-processing code:*

```
if exists
  (select $al(codegen_q(body))$.val
   from ($codegen_q(body)$) as $al(codegen_q(body))$
   where $al(codegen_q(body))$.val = True)
then
  insert into $nm(codegen_b(exp))$(val) values ($var$);
end if;
```

reject-iterator. Let exp be an expression of the form $source$->reject($var|body$). This expression returns a subcollection of the source collection src containing all elements for which $body$ evaluates to $false$. The "holes" in the scheme $codegen_b(exp)$ will be filled as follows:

- *value-specific type:* the SQL-PL type which represents, according to our mapping, the type of the elements in the src.
- *Iteration-processing code:*

```
if exists
  (select True
   from ($codegen_q(body)$) as $al(codegen_q(body))$
   where val = False)
then
  insert into $nm(codegen_b(exp))$(val) values ($var$);
end if;
```

isUnique-iterator. Let exp be an expression of the form $source$->isUnique($var|$ $body$). This expression returns True if all elements of the collection of objects that result from evaluating $body$ for each element in the source collection src, are different. The "holes" in the scheme $codegen_b(exp)$ will be filled as follows:

- *value-specific type:* boolean
- *Initialization code:*

```
create temporary table $nm_{acc}(codegen_b(exp))$ (val $\boxed{\textit{value-specific type}}$);
```

  where *value-specific type:* the SQL-PL type which represents, according to our mapping, the type of the elements in the $body$.

- *Iteration-processing code:*

```
insert into $nm_{acc}(codegen_b(exp))$(val) $codegen_q(body)$;
```

- *End code:*

```
insert into $nm(codegen_b(exp))$(val)
(select $al_1(codegen_q(exp))$.val = $al(codegen_q(exp))$.val
from
  (select count(*) as val
  from
    (select distinct val from $nm_{acc}(codegen_q(exp))$) as $al(codegen_q(body))$)
        as $al_1(codegen_q(body))$,
  (select count(*) as val
  from $nm_{acc}(codegen_b(exp))$) as $al(codegen_q(body))$);
cl2sql
```

Example 11 *Nested and sequential iterator expressions*

Q7. *Check whether there is a car owner whose surname is Perez.*
Car.allInstances()->select(c|c.owner->exists(p|p.surname='Perez'))

```
begin
declare done int default 0;
declare body Boolean default false;
declare var0 int;
declare crs cursor for select pk as val from Car;
declare continue handler for sqlstate '02000' set done = 1;
drop table if exists select0;
create temporary table select0(val int);
open crs;
repeat
  fetch crs into var0;
  begin
  declare done int default 0;
  declare result boolean default false;
  declare tResult int default 0;
  declare var01 int;
  declare crs cursor for
   (select ownership.owner as val
   from (select var0 as val) as t0
   left join ownership on t0.val = ownership.ownedCars
   where ownership.owner is not null);
  declare continue handler for sqlstate '02000' set done = 1;
  drop table if exists exists01;
  create temporary table exists01(val int);
  open crs;
  repeat
   fetch crs into var01;
   if not done then
    select val into tResult
    from
     (select (select Person.name as val
      from (select var01 as val) as t1
      left join Person on t1.val = Person.pk) = (select 'Perez' as val) as val) as t;
    if tResult then
     set done = 1;
     set result = 1;
    end if;
   end if;
   until done end repeat;
   insert into exists01(val) values (result);
   close crs;
  end;
  if not done then
 select val into body from (select * from exists01) as t;
 if body then
    insert into select0(val) values (var0);
end if;
end if;
until done end repeat;
close crs;
end;
select * from select0;
```

Q8. *Check whether exists a person, who owner a car, with surname Perez.*
Car.allInstances()->collect(p|p.owner)->exists(q|q.surname='Perez')

```
begin
  begin
  declare done int default 0;
  declare var1 int;
  declare crs cursor for select pk as val from Car;
  declare continue handler for sqlstate '02000' set done = 1;
  drop table if exists collect0;
  create temporary table collect0(val boolean);
  open crs;
  repeat
    fetch crs into var1;
    if not done then
      insert into collect0(val)
        (select ownership.owner as val
        from (select var1 as val) as tbl1
        left join ownership on tbl1.val = ownership.ownedCars
        where ownership.owner is not null or tbl1.val is null);
    end if;
    until done end repeat;
    close crs;
  end;
  begin
    declare done int default 0 ;
    declare result boolean default false;
    declare tempResult boolean default false;
    declare var2 int;
    declare crs cursor for select val from collect0;
    declare continue handler for sqlstate '02000' set done = 1;
  drop table if exists exists0;
  create temporary table exists0(val bool);
  open crs;
  repeat
    fetch crs into var2;
    if not done then
      select val into tempResult
      from
        (select tbl5.val = tbl6.val as val
        from
          (select Person.surname as val
          from Person, (select var2 as val) as tbl4
          where pk = tbl4.val) as tbl5,
          (select 'Perez' as val) as tbl6) as tbl8;
      if tempResult then
        set done = 1;
        set result = True;
      end if;
    end if;
    until done end repeat;
    insert into exists0(val) (select result as val);
    close crs;
end;
select val from exists0;
end;
```

□

To conclude this section, we would like to remark, some general invariants in our mappings:

- nested operators, which requires blocks definitions, are mapped into nested blocks, while sequential operators are mapped into sequential blocks.
- the results of expressions with simple types and sets are mapped into tables with a column called val; while expressions with sequence types are mapped into tables with two columns, one for the values (i.e. `val`) and the another for the positions (i.e. `pos`).
- when we talk about iterators, the statement:

  ```
  declare crs cursor for (codegen_q(src));
  ```

  defined when the *src*-collection is a set or bag, is changed to:

  ```
  declare crs cursor for
    (select al(codegen_q(src)).val
    from (codegen_q(src))) as al(codegen_q(src))
    order by al(codegen_q(src)).pos;
  ```

  to deal with *src*-collection ordered.

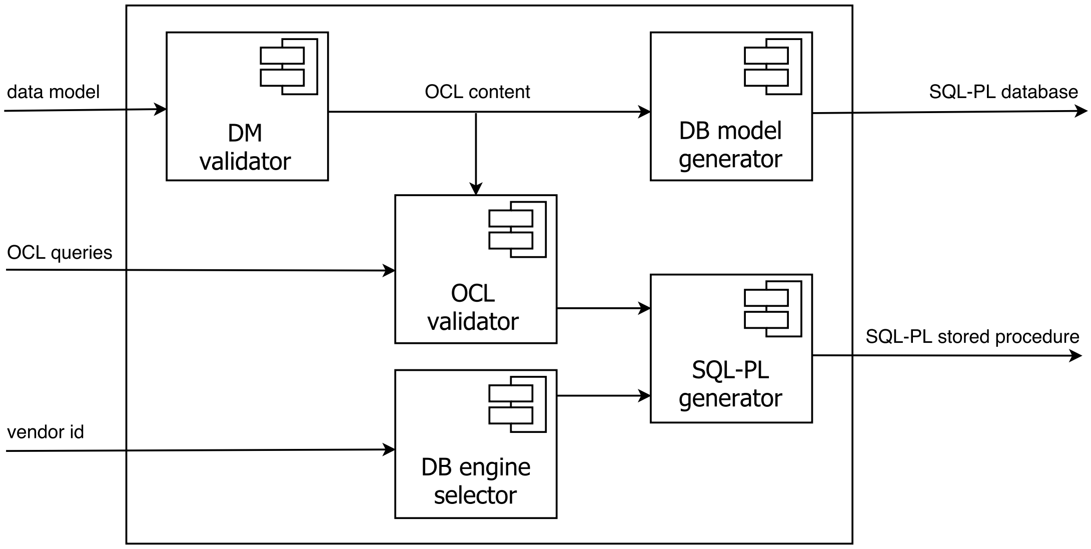


Figure 2.5: SQL-PL4OCL tool component architecture

2.2.3 The SQL-PL4OCL tool

The SQL-PL4OCL tool rewrites the tool introduced in [34] to target not just MySQL (or MariaDB) but also PostgreSQL and SQL Server DBMS. The new implementation does not comply to the mapping we introduced in [25, 34] but to the one defined in section 2.2.2.

Essentially, SQL-PL4OCL is a Java Web Application tool that using as input a data model (as specified in Section 2.2.1), a list of OCL queries, and a vendor identifier, it generates a set of statements ready to create the database with the tables that correspond to the data model (following the mapping introduced in Section 2.2.1), and a list of stored procedures (one per OCL query, following the definition specified in Section 2.2.2). Figure 2.6 shows two screen-shots of the tool interface. Of course, the resulting code is produced adapted to the syntax of each target RDBMS. ec Figure 2.5 shows the main components of the tool architecture. These are:

- DM validator: This component checks whether the input data model fulfills the restrictions about well-formedness that we explain in Section 2.2.1, so as to serve as a valid context for OCL queries.

- OCL validator: This component parses each OCL query of input in the context of the data model. Only if a query parses correctly (and our mapping covers it), it is used as input to produce code.

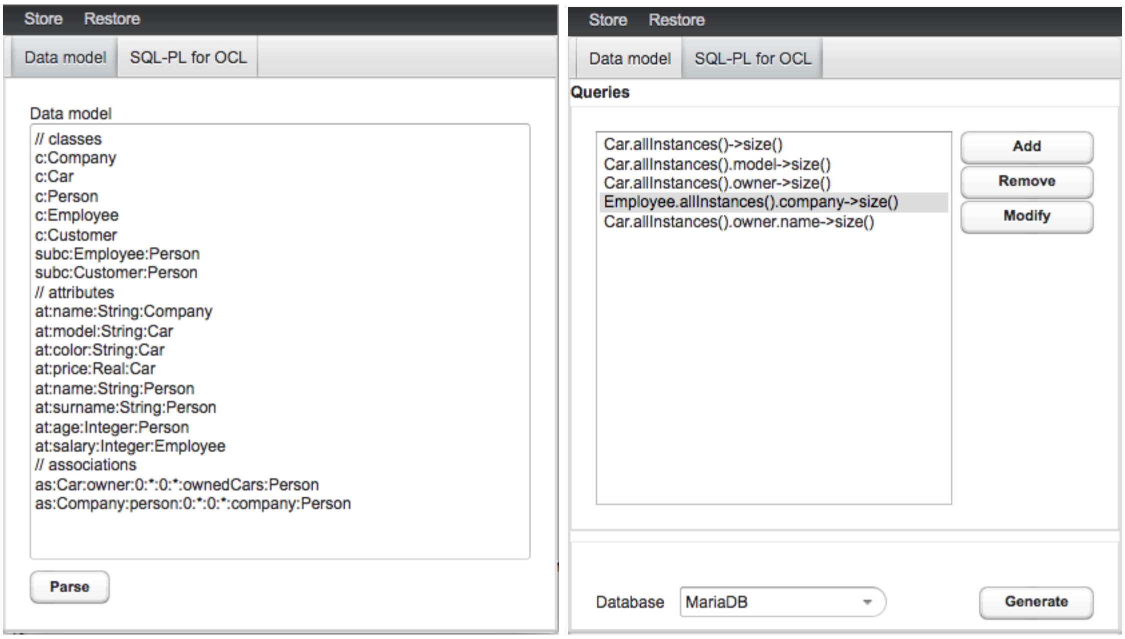
- DB engine selector: This component receives as input the vendor identifier so as the code generated is syntactically adapted to the selected RDBMS.

- DB model generator: This component generates the *engine-specific* statements to create the database and corresponding tables.

- SQL-PL generator: This component generates the *engine-specific* statements to create the SQL-PL stored procedures corresponding to the input OCL queries.

The complexity of supporting multiple RDBMS is brought by their implementation differences. Perhaps the most noticeable difference is the language they parse. Even though all engines use some flavor of SQL, these all differ in how variables, stored procedures, and built-in functions are declared in their procedural extensions. Also, PostgreSQL supports different procedural languages (we targeted at PL/pgSQL), MS SQL Server uses Transact SQL and MySQL uses yet another dialect (fully compatible with MariaDB's).

As implementation strategy, we avoided the burden of dealing with the subtleties of each SQL dialect within the mapping algorithm by defining a plugin-based architecture. In this architecture, each plugin component is responsible for performing the appropriate translation for the RDBMS it targets. In [98], the reader can find a comparison that gives idea of the variations among the different SQL dialects. We do not discuss the differences here since, in our view, they do not add to our discussion. Instead, we encourage the interested reader to use our tool, which is available at [28].



Data model

```
// classes
c:Company
c:Car
c:Person
c:Employee
c:Customer
subc:Employee:Person
subc:Customer:Person
// attributes
at:name:String:Company
at:model:String:Car
at:color:String:Car
at:price:Real:Car
at:name:String:Person
at:surname:String:Person
at:age:Integer:Person
at:salary:Integer:Employee
// associations
as:Car:owner:0:*:0:*:ownedCars:Person
as:Company:person:0:*:0:*:company:Person
```

Queries

```
Car.allInstances()->size()
Car.allInstances().model->size()
Car.allInstances().owner->size()
Employee.allInstances().company->size()
Car.allInstances().owner.name->size()
```

Database MariaDB

Figure 2.6: SQL-PL4OCL tool: screen-shots

2.3 Benchmark

2.3.1 Description

The data model for our benchmark is the Car-Company model shown in Figure 2.1. The expressions that we consider in our benchmark are the following:

Q1. Car.allInstances()−>size()
Q2. Car.allInstances().model−>size()
Q3. Car.allInstances().owner−>size()
Q4. Employee.allInstances().company−>size()
Q5. Car.allInstances().owner.name−>size()

	MySQL	MariaDB	PostgreSQL	MSSQL
Q1	0.19s	0.13s	**0.10s**	0.12s
Q2	0.25s	**0.20s**	0.33s	0.28s
Q3	0.36s	0.35s	0.27s	**0.26s**
Q4	**0.04s**	**0.04s**	**0.04s**	0.05s
Q5	0.55s	**0.40s**	**0.40s**	0.42s
Q6	1.05s	**0.55s**	1.06s	1.03s
Q7	2.07s	**1.56s**	1.99s	2.08s
Q8	50.02s	**43.08s**	57.04s	53.47s
Q9	9.14s	**8.00s**	8.18s	8.89s
Q10	0.05s	**0.04s**	0.07s	0.05s
Q11	49.56s	**40.02s**	40.10s	43.46s
Q12	59.58s	**51.23s**	51.25s	54.82s
Q13	**1.67s**	1.98s	2.35s	1.90s
Q14	59.52s	**54.33s**	63.35s	58.33s

Table 2.1: SQL-PL4OCL. Evaluation times.

Q6. Car.allInstances().owner–>oclAsType(Employee).salary–>size()
Q7. Car.allInstances().owner–>oclAsType(Employee).ownedCars–>size()
Q8. Car.allInstances()–>select(c|c.color<>”black”)–>size()
Q9. Car.allInstances()–>forAll(c|c.color<>”black”)
Q10. Car.allInstances()–>exists(c|c.color<>’black”)
Q11. Car.allInstances()–>collect(x|x.color)–>size()
Q12. Car.allInstances()–>collect(x|x.owner.ownedCars)–>size()
Q13. Car.allInstances().model–>asSequence()–>size()
Q14. Car.allInstances()–>asSequence()–>select(c|c.color<>”black”)–>size()

2.3.2 Results

Table 2.1 shows a benchmark to test the performance (in terms of the evaluation time) of a sample of OCL queries mapped into the different DBMS. In this sample, we included both simple expressions (Q1-Q7), and complex expressions (Q8-Q14), including iterator and sequence operators. All the expressions in the benchmark were evaluated on an artificial scenario that we created. The scenario is an instance of the Car-Company data model depicted in 2.1. This instance contains 10^6 instances of class Car, 10^5 instances of class Person (all of them are Employees), and 10^2 instances of class Company, where each company is associated to 10^2 instances of Person, and each person owns 10 different cars. All car instances have a color different from black.

We used bold font to highlight the lowest evaluation time of each query in Table 2.1. By just taking a look, it is clear that MariaDB, an open source database, achieves the fastest evaluation times for the majority of the queries and, most importantly, for almost the totality of complex expressions.[4]

Based on our experiments, we can identify three parameters which seem to correlate

[4] We ran the benchmark in a laptop with an Intel Core m7, 1.3 GHz, 8 GB RAM, and 500 GB Flash Storage. The RDBMS versions used were MySQL 5.7, MariaDB 10.1, SQL Server 2016 Express, and PostgreSQL 9.6.1.

directly to the increase in the evaluation time of an expression translated by our mapping. More concretely,

i. The OCL expression contains access to attributes or association-ends. Their translation into left joins (of size n × m) makes them expensive in time. Also, the materialization of a left join performed between different tables (i.e., for translating an association, as in Q3 and Q7) is more expensive than one performed by a table with itself (i.e., for translating access to an attribute, as in Q2 and Q6). The time gets worse when the source table is larger, i.e., with a high n. For example, compare evaluation times for queries Q3 and Q4 where the size of the source collection is 10^6 and 10^5 (resp.), or queries Q2 and Q12 for which the size of the left join (owner.ownedCars) is $10^6 \times 10$ and 1×10 (resp.).

ii. The size of the outermost source collection in an OCL iterator expression (if there is no stop criterion applied). For example, to evaluate Q9 the cursor has to fetch values from a table of size 10^6, however, to evaluate Q10 the cursor only fetches one value and the procedure stops. Notice also the different evaluation time between Q2 and Q11 (which are similar expressions in semantics) since the last is shaped as an iterator expression.

iii. The number of insertions to a table when this is required by the mapping to translate a query. In particular, insertions to a table are always required for evaluating sequence expressions. As an example we compare queries Q8 and Q9. The size of the source expression for both queries is the same (10^6). However, the evaluation of Q8 requires the insertion of intermediate values into a table while Q9 evaluation does not. Similarly happens with Q2 and Q13. The different evaluation time between Q8 and Q14 seems to be due to the generation of the auto-incremented *position* value for the latter.

Chapter 3

Mapping OCL as a constraint language

© Joaquín S. Lavado, QUINO. Toda
Mafalda, Penguin Random House, España

3.1 From OCL to many-sorted first-order logic

We introduce here a mapping from OCL to MSFOL, which supports the use of Satisfiability Modulo Theories (SMT) solvers for checking UML/OCL satisfiability problems.[1] This mapping is the result of an evolution of different mappings, which we briefly discussed below.

In [20] we proposed a first mapping from UML/OCL to first-order logic, called OCL2-FOL, which did not support UML generalization or OCL undefinedness. In [26] we proposed a second mapping, called OCL2FOL$^+$, which did take into account OCL undefinedness, but did not support UML generalization. Moreover, OCL2FOL$^+$ turned out to be rather inefficient in practice, since SMT solvers would often return *unknown*, as a consequence of two facts: first, that non-trivial OCL constraints contain expressions that are naturally mapped to quantified formulas (since they refer to *all* the objects in a class, for example), and, secondly, that techniques for dealing with quantified formulas in SMT are generally incomplete.

To overcome this limitation, we decided to use SMT solvers along with finite model finding methods for checking the satisfiability of the formulas resulting from our mapping. In particular, we opted for using the SMT solver CVC4 [10], which has a finite model finding method [81] fully integrated with its SMT solver architecture. The finite model finding

[1] SMT solvers generalize Boolean satisfiability (SAT) by incorporating equality reasoning, arithmetic, fixed-size bit-vectors, arrays, quantifiers, and other first-order theories.

method implemented in CVC4 requires, however, that quantified variables in the input problem always range over finite domains. OCL2FOL$^+$ could not satisfy this requirement, since its target formalism was *unsorted* FOL: variables in quantified formulas generated by OCL2FOL$^+$ range over a single, infinite domain that includes the integer numbers. By switching to *many-sorted* FOL (MSFOL), we were able to satisfy the aforementioned requirement: variables in quantified formulas range now over the domain of a distinguished sort, called *Classifier*, which essentially contains the objects in an object diagram and the undefined values (but not the integer numbers or the strings), and which, for the purpose of UML/OCL verification, can be considered as finite (object diagrams can be assumed to contain only a finite number of objects). Finally, many-sorted FOL provides a more adequate target formalism than unsorted FOL for mapping UML generalization and generalization-related OCL operations.

Hence, we propose here a mapping from OCL to many-sorted first-order logic [27] which successfully overcomes the limitations of our previous mappings. First, it accepts as input a significantly larger subset of the UML/OCL language; in particular, it supports UML generalization, along with the generalization-related OCL operators. Secondly, it generates as output a class of satisfiability problems that are amenable to checking by using SMT solvers with finite model finding capabilities.

3.1.1 Mapping data models

Our mapping from OCL to MSFOL builds upon a base mapping from data models to MSFOL theories, called $\mathrm{o2f_{data}}$.

Let $\mathcal{D} = \langle C, CH, AT, AS, ASO, MU\rangle$ be a data model. In a nutshell, $\mathrm{o2f_{data}}(\mathcal{D})$ contains:

- The sorts *Int* and *String*, whose intended meaning is to represent the integer numbers and the strings.
- The constants *nullInt*, *nullString*, *invalInt*, and *invalString*, whose intended meaning is to represent null and invalid for integers and strings.
- The sort *Classifier*, whose intended meaning is to represent *all* the objects in an instance of $\mathcal{D}$, as well as null and invalid for objects.
- The sort *Type*, whose intended meaning is to represent the type identifiers declared in $\mathcal{D}$.
- For each class $\mathsf{c} \in C$, a unary predicate c, whose intended meaning is to define the objects of the class c in an instance of $\mathcal{D}$
- For each attribute $\langle at, c, t\rangle \in AT$, a function at, whose intended meaning is to define the values of the attribute at in the objects in an instance of $\mathcal{D}$.
- For each binary association, $(as_{(c,c')}, as_{(c',c)}) \in ASO$, with multiplicity, $\langle as_{(c,c')}, *\rangle$, $\langle as_{(c',c)}, *\rangle \in ASO$. A binary predicate as_as', whose intended meaning is to define the links through the association $\langle as_{(c,c')}, as_{(c',c)}\rangle$ between the objects.[2]

[2] For associations with both association-ends with multiplicities 0..1, our mapping declares a function for each association-end, instead of a predicate for the association. Then, for associations with one association-end with multiplicity * and the other with multiplicity 0..1, our mapping declares a binary predicate for the association-end with multiplicity * and a function for the one with multiplicity 0..1.

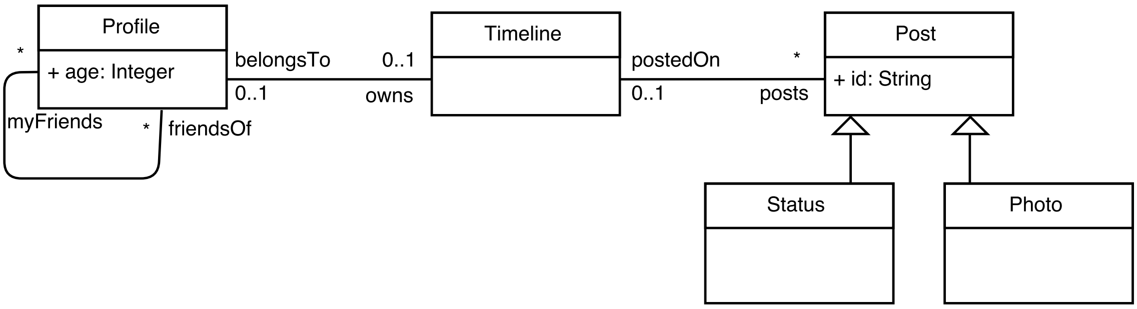


Figure 3.1: BasicSocNet: A data model for a basic social network.

- The axioms that constrain the meaning of the aforementioned sorts, constants, predicates, and functions.

Formally, $\mathrm{o2f}_{\mathrm{data}}$ is defined as follow:

Definition 3 *Let $\mathcal{D} = \langle C, CH, AT, AS, ASO, MU\rangle$ be a data model. Then,* $\mathrm{o2f}_{\mathrm{data}}(\mathcal{D})$ *is an MSFOL theory, which is defined below.*

- *It declares two sorts, Classifier and Type, to represent the OCL types* Classifier *and* Type*. It also declares two sorts, Int and String, to represent the integer numbers and the strings.* [3]

- *It declares two constants of sort Classifier, nullClassifier and invalClassifier, to represent, the values* null *and* invalid *of type* Classifier*. In addition, it includes the following axiom: ysql*

  $$\neg(\mathit{nullClassifier} = \mathit{invalClassifier}).$$

  Similarly for the type Type*.*

- *It declares two constants of sort Int, nullInt and invalInt, to represent, respectively, the values* null *and* invalid *of the primitive data-type* Integer*. In addition, it includes the following axiom:*

  $$\neg(\mathit{nullInt} = \mathit{invalInt}).$$

  Similarly for the primitive data-type String*.*

- *For each class $c \in C$, it declares a predicate c: Classifier → Bool, to represent the objects of type c. In addition, it includes the following axioms:*

  $$\forall(x{:}\mathit{Classifier})(c(x) \Rightarrow \neg(\textstyle\bigvee_{c' \in (C \setminus \{c\})} c'(x))).$$
  $$\neg(c(\mathit{nullClassifier}) \vee c(\mathit{invalClassifier})).$$

- *For each attribute $at_{(c,\mathsf{Integer})}$, it declares a function at: Classifier → Int. In addition, it includes the following axioms:*

[3] We assume that *Int* and *String* are declared with the standard operations and semantics.

$$\begin{array}{l}
\forall(x{:}\mathit{Classifier})((\bigvee_{s\preceq c}(s(x))) \Rightarrow at(x) \neq \mathit{InvalInt}).\\
at(\mathit{nullClassifier}) = \mathit{InvalInt}.\\
at(\mathit{invalClassifier}) = \mathit{InvalInt}.
\end{array}$$

- *For each association between two classes c and c', with association-ends $as_{(c,c')}$ and $as'_{(c',c)}$, such that $\langle as_{(c,c')},\ *\rangle$, $\langle as'_{(c',c)},\ *\rangle \in MU$, it declares a predicate as_as': Classifier $\times$ Classifier $\rightarrow$ Bool. In addition, it includes the following axiom:*

$$\begin{array}{l}
\forall(x{:}\mathit{Classifier}, y{:}\mathit{Classifier})\\
\quad(as\_as'(x,y) \Rightarrow ((\bigvee_{s\preceq c}(s(x))) \wedge (\bigvee_{s'\preceq c'}(s'(y))))).
\end{array}$$

- *For each association as between two classes c and c', with association-ends $as_{(c,c')}$ and $as'_{(c',c)}$, such that $\langle as, \texttt{0..1}\rangle$, $\langle as', \texttt{0..1}\rangle \in MU$ it declares two functions, as, as':* Classifier $\rightarrow$ Classifier. *In addition, it includes the following axioms:*

$$\begin{array}{l}
\forall(x{:}\mathrm{Classifier}, y{:}\mathrm{Classifier})\\
\quad(((as(x) = y) \wedge (\bigvee_{s\preceq c}(s(x))) \wedge (\bigvee_{s'\preceq c'}(s'(y)))) \Rightarrow as'(y) = x).\\
\forall(x{:}\mathrm{Classifier})((\bigvee_{s\preceq c}(s(x))) \Rightarrow (as(x) = \mathrm{nullClassifier} \vee (\bigvee_{s'\preceq c'}(s'(as(x)))))).\\
as(\mathrm{nullClassifier}) = \mathrm{invalClassifier}.\\
as(\mathrm{invalClassifier}) = \mathrm{invalClassifier}.
\end{array}$$

$$\begin{array}{l}
\forall(x{:}\mathrm{Classifier}, y{:}\mathrm{Classifier})\\
\quad(((as'(y) = x) \wedge (\bigvee_{s\preceq c}(s(x))) \wedge (\bigvee_{s'\preceq c'}(s'(y)))) \Rightarrow as(x) = y).\\
\forall(y{:}\mathrm{Classifier})((\bigvee_{s'\preceq c'}(s'(y))) \Rightarrow (as'(y) = \mathrm{nullClassifier} \vee (\bigvee_{s\preceq c}(s(as'(y)))))).\\
as'(\mathrm{nullClassifier}) = \mathrm{invalClassifier}.\\
as'(\mathrm{invalClassifier}) = \mathrm{invalClassifier}.
\end{array}$$

- *For each association as between two classes c and c', with association-ends $as_{(c,c')}$ and $as'_{(c',c)}$, such that $\langle as, \texttt{0..1}\rangle$, $\langle as', *\rangle \in MU$, it declares a function as:* Classifier $\rightarrow$ Classifier *and a predicate as':* Classifier $\times$Classifier $\rightarrow$ Bool. *In addition, it includes the following axioms:*

$$\begin{array}{l}
\forall(x{:}\mathrm{Classifier}, y{:}\mathrm{Classifier})\\
\quad(((as(x) = y) \wedge (\bigvee_{s\preceq c}(s(x))) \wedge (\bigvee_{s'\preceq c'}(s'(y)))) \Rightarrow as'(y,x)).\\
\forall(x{:}\mathrm{Classifier})((\bigvee_{s\preceq c}(s(x))) \Rightarrow (as(x) = \mathrm{nullClassifier} \vee (\bigvee_{s'\preceq c'}(s'(as(x)))))).\\
as(\mathrm{nullClassifier}) = \mathrm{invalClassifier}.\\
as(\mathrm{invalClassifier}) = \mathrm{invalClassifier}.
\end{array}$$

$$\begin{array}{l}
\forall(x{:}\mathrm{Classifier}, y{:}\mathrm{Classifier})(as'(y,x) \Rightarrow as(x) = y).\\
\forall(x{:}\mathrm{Classifier}, y{:}\mathrm{Classifier})(as'(y,x) \Rightarrow ((\bigvee_{s\preceq c}(s(x))) \wedge (\bigvee_{s'\preceq c'}(s'(y))))).
\end{array}$$

- *For each class $c \in C$, it declares a constant c_{type} of sort Type. In addition, it includes the following axiom:*

$$\bigwedge_{c'\in C\setminus\{c\}} \neg(c_{\mathrm{type}} = c'_{\mathrm{type}}).$$

- *It declares two predicates OclIsTypeOf, OclIsKindOf: Classifier $\times$ Type–>Bool. In addition, for each class $c \in C$, it includes the following axioms:*

$\forall(x{:}\mathit{Classifier})(\mathit{OclIsTypeOf}(x, c_{\mathrm{type}}) \Leftrightarrow c(x))$.
$\forall(x{:}\mathit{Classifier})(\mathit{OclIsKindOf}(x, c_{\mathrm{type}}) \Leftrightarrow \bigvee_{s \preceq c}(s(x)))$.

In the following example we illustrate the mapping $\mathrm{o2f}_{\mathrm{data}}$.

Example 12 *Consider the data model* SSN *shown in Figure 3.1, which models a basic social network. Then, the MSFOL theory* $\mathrm{o2f}_{\mathrm{data}}$(SSN) *contains, among other elements:*

- *The constants nullClassifier and invalClassifier of sort* Classifier*, along with the axiom:*

  $\neg(\mathit{nullClassifier} = \mathit{invalClassifier})$.

- *The constants nullInt and invalInt of sort Int, along with the axiom:*

  $\neg(\mathit{nullInt} = \mathit{invalInt})$.

- *The predicate* Profile:*Classifier* → *Bool, along with the axioms:*

  $\forall(x)(\mathit{Profile}(x) \Rightarrow \neg(\mathit{Photo}(x) \vee \mathit{Status}(x) \vee \mathit{Timeline}(x) \vee \mathit{Post}(x)))$.
  $\neg(\mathit{Profile}(\mathit{nullClassifier}) \vee \mathit{Profile}(\mathit{invalClassifier}))$.

- *The function age: Classifier* → *Int, along with the axioms:*

  $\mathit{age}(\mathit{nullClassifier}) = \mathit{invalInt}$.
  $\mathit{age}(\mathit{invalClassifier}) = \mathit{invalInt}$.
  $\forall(x)(\mathit{Profile}(x) \Rightarrow \neg(\mathit{age}(x) = \mathit{invalInt}))$.

- *The predicate myFriends_friendsOf:Classifier* × *Classifier* → *Bool, along with the axioms:*

  $\forall(x, y)(\mathit{myFriends\_friendsOf}(x, y) \Leftrightarrow (\mathit{Profile}(x) \wedge \mathit{Profile}(y)))$.

- *The constants* $\mathit{Post}_{\mathrm{type}}$*,* $\mathit{Photo}_{\mathrm{type}}$*, and* $\mathit{Status}_{\mathrm{type}}$ *of sort Type, along with the axioms:*

  $\neg(\mathit{Post}_{\mathrm{type}} = \mathit{Photo}_{\mathrm{type}})$.
  $\neg(\mathit{Post}_{\mathrm{type}} = \mathit{Status}_{\mathrm{type}})$.
  $\neg(\mathit{Photo}_{\mathrm{type}} = \mathit{Status}_{\mathrm{type}})$.

- *The predicate oclIsKindOf : Classifier* × *Type* → *Bool, along with the axioms:*

  $\forall(x)(\mathit{oclIsKindOf}(x, \mathit{Post}_{\mathrm{type}}) \Leftrightarrow (\mathit{Post}(x) \vee \mathit{Photo}(x) \vee \mathit{Status}(x)))$.
  $\forall(x)(\mathit{oclIsKindOf}(x, \mathit{Photo}_{\mathrm{type}}) \Leftrightarrow \mathit{Photo}(x))$.
  $\forall(x)(\mathit{oclIsKindOf}(x, \mathit{Status}_{\mathrm{type}}) \Leftrightarrow \mathit{Status}(x))$.

3.1.2 Mapping OCL expressions

OCL2MSFOL is designed for checking the satisfiability of OCL constraints: it accepts as input OCL Boolean expressions, and only deals with non-Boolean expressions inasmuch as they appear as subexpressions of Boolean expressions.

The mappings $\mathrm{o2f}_{\mathrm{true}}$, $\mathrm{o2f}_{\mathrm{false}}$, $\mathrm{o2f}_{\mathrm{null}}$, and $\mathrm{o2f}_{\mathrm{inval}}$

In the presence of undefinedness, OCL Boolean expressions can evaluate not only to true and false but also to null or invalid. To cope with four Boolean values in a two-valued logic like MSFOL, we define four mappings, namely, $\mathrm{o2f}_{\mathrm{true}}$, $\mathrm{o2f}_{\mathrm{false}}$, $\mathrm{o2f}_{\mathrm{null}}$, and $\mathrm{o2f}_{\mathrm{inval}}$, which formalize when a Boolean expression evaluates to true, when to false, when to null, and when to invalid. We define these mappings by *structural recursion.* In the recursive case, when the subexpression is a non-Boolean type, we call an auxiliary mapping, $\mathrm{o2f}_{\mathrm{eval}}$, which we will discuss below. For now, it is sufficient to know that $\mathrm{o2f}_{\mathrm{eval}}$ returns a term when its argument is an expression of a class type or of type Integer or String, and that it returns a predicate when its argument is an expression of a set type.[4]

Let *expr* be an expression we assume, without loss of generality, that each iterator in *expr* introduces a different iterator variable. Moreover, we denote by fVars(*expr*) the sequence formed by the free variables in *expr*, sorted alphabetically. Finally, we denote by $\mathrm{App}(P,(x_1,\ldots,x_n),y)$ the atomic formula $P(x_1,...,x_n,y)$, and we denote by $\mathrm{App}(f,(x_1,\ldots,x_n))$ the term $f(x_1,...,x_n)$.

Definition 4 *Let expr, $expr_1$, $expr_2$ be boolean expressions, and src be expression of the appropriate type, we define the mappings* $\mathrm{o2f}_{\mathrm{true}}$, $\mathrm{o2f}_{\mathrm{false}}$, $\mathrm{o2f}_{\mathrm{null}}$, *and* $\mathrm{o2f}_{\mathrm{inval}}$ *as follow:*

oclIsUndefined-expressions:

$\mathrm{o2f}_{\mathrm{true}}(expr.\mathsf{oclIsUndefined()}) = \mathrm{o2f}_{\mathrm{null}}(expr) \vee \mathrm{o2f}_{\mathrm{inval}}(expr).$

$\mathrm{o2f}_{\mathrm{false}}(expr.\mathsf{oclIsUndefined()}) = \neg(\mathrm{o2f}_{\mathrm{null}}(expr) \vee \mathrm{o2f}_{\mathrm{inval}}(expr)).$

$\mathrm{o2f}_{\mathrm{null}}(expr.\mathsf{oclIsUndefined()}) = \bot.$

$\mathrm{o2f}_{\mathrm{inval}}(expr.\mathsf{oclIsUndefined()}) = \bot.$

oclIsInvalid-expressions:

$\mathrm{o2f}_{\mathrm{true}}(expr.\mathsf{oclIsInvalid()}) = \mathrm{o2f}_{\mathrm{inval}}(expr).$

$\mathrm{o2f}_{\mathrm{false}}(expr.\mathsf{oclIsInvalid()}) = \neg(\mathrm{o2f}_{\mathrm{inval}}(expr)).$

$\mathrm{o2f}_{\mathrm{null}}(expr.\mathsf{oclIsInvalid()}) = \bot.$

$\mathrm{o2f}_{\mathrm{inval}}(expr.\mathsf{oclIsInvalid()}) = \bot.$

oclIsTypeOf-expressions:

$\mathrm{o2f}_{\mathrm{true}}(expr.\mathsf{oclIsTypeOf}(c)) = \mathrm{OclIsTypeOf}(\mathrm{o2f}_{\mathrm{eval}}(expr), c).$

$\mathrm{o2f}_{\mathrm{false}}(expr.\mathsf{oclIsTypeOf}(c)) = \neg(\mathrm{OclIsTypeOf}(\mathrm{o2f}_{\mathrm{eval}}(expr), c)).$

$\mathrm{o2f}_{\mathrm{null}}(expr.\mathsf{oclIsTypeOf}(c)) = \bot.$

$\mathrm{o2f}_{\mathrm{inval}}(expr.\mathsf{oclIsTypeOf}(c)) = \bot.$

[4] We assume that all non-Boolean subexpressions have either a class type, a primitive type (either Integer or String), or a set type.

oclIsKindOf-expressions:

$\mathrm{o2f}_{\mathrm{true}}(expr.\mathsf{OclIsKindOf}(c)) = \mathrm{OclIsKindOf}(\mathrm{o2f}_{\mathrm{eval}}(expr), c).$

$\mathrm{o2f}_{\mathrm{false}}(expr.\mathsf{OclIsKindOf}(c)) = \neg(\mathrm{OclIsKindOf}(\mathrm{o2f}_{\mathrm{eval}}(expr), c)).$

$\mathrm{o2f}_{\mathrm{null}}(expr.\mathsf{OclIsKindOf}(c)) = \bot.$

$\mathrm{o2f}_{\mathrm{inval}}(expr.\mathsf{OclIsKindOf}(c)) = \bot.$

equality-expressions:

$$\begin{aligned}&\mathrm{o2f}_{\mathrm{true}}(expr_1 = expr_2) = (\mathrm{o2f}_{\mathrm{null}}(expr_1) \wedge \mathrm{o2f}_{\mathrm{null}}(expr_2)) \vee \\ &\quad (\mathrm{o2f}_{\mathrm{eval}}(expr_1) = \mathrm{o2f}_{\mathrm{eval}}(expr_2) \\ &\qquad \wedge \neg(\mathrm{o2f}_{\mathrm{null}}(expr_1) \vee \mathrm{o2f}_{\mathrm{inval}}(expr_1) \vee \mathrm{o2f}_{\mathrm{null}}(expr_2) \vee \mathrm{o2f}_{\mathrm{inval}}(expr_2))).\end{aligned}$$

$$\begin{aligned}&\mathrm{o2f}_{\mathrm{false}}(expr_1 = expr_2) = (\neg(\mathrm{o2f}_{\mathrm{eval}}(expr_1) = \mathrm{o2f}_{\mathrm{eval}}(expr_2)) \\ &\qquad \wedge \neg(\mathrm{o2f}_{\mathrm{null}}(expr_1) \vee \mathrm{o2f}_{\mathrm{inval}}(expr_1) \vee \mathrm{o2f}_{\mathrm{null}}(expr_2) \vee \mathrm{o2f}_{\mathrm{inval}}(expr_2))).\end{aligned}$$

$\mathrm{o2f}_{\mathrm{null}}(expr_1\texttt{=}expr_2) = \bot.$

$$\begin{aligned}&\mathrm{o2f}_{\mathrm{inval}}(expr_1\texttt{=}expr_2) = \mathrm{o2f}_{\mathrm{inval}}(expr_1) \vee \mathrm{o2f}_{\mathrm{inval}}(expr_2) \\ &\quad \vee (\mathrm{o2f}_{\mathrm{null}}(expr_1) \wedge \neg\mathrm{o2f}_{\mathrm{null}}(expr_2)) \vee (\neg\mathrm{o2f}_{\mathrm{null}}(expr_1) \wedge \mathrm{o2f}_{\mathrm{null}}(expr_2)).\end{aligned}$$

inequality-expressions:

$$\begin{aligned}&\mathrm{o2f}_{\mathrm{true}}(expr_1 <> expr_2) = (\neg(\mathrm{o2f}_{\mathrm{eval}}(expr_1) = \mathrm{o2f}_{\mathrm{eval}}(expr_2)) \\ &\quad \wedge \neg(\mathrm{o2f}_{\mathrm{null}}(expr_1) \vee \mathrm{o2f}_{\mathrm{inval}}(expr_1) \vee \mathrm{o2f}_{\mathrm{null}}(expr_2) \vee \mathrm{o2f}_{\mathrm{inval}}(expr_2))).\end{aligned}$$

$$\begin{aligned}&\mathrm{o2f}_{\mathrm{false}}(expr_1 <> expr_2) = (\mathrm{o2f}_{\mathrm{null}}(expr_1) \wedge \mathrm{o2f}_{\mathrm{null}}(expr_2)) \vee \\ &\quad (\mathrm{o2f}_{\mathrm{eval}}(expr_1) = \mathrm{o2f}_{\mathrm{eval}}(expr_2) \\ &\qquad \wedge \neg(\mathrm{o2f}_{\mathrm{null}}(expr_1) \vee \mathrm{o2f}_{\mathrm{inval}}(expr_1) \vee \mathrm{o2f}_{\mathrm{null}}(expr_2) \vee \mathrm{o2f}_{\mathrm{inval}}(expr_2))).\end{aligned}$$

$\mathrm{o2f}_{\mathrm{null}}(expr_1 <> expr_2) = \bot.$

$$\begin{aligned}&\mathrm{o2f}_{\mathrm{inval}}(expr_1 <> expr_2) = \mathrm{o2f}_{\mathrm{inval}}(expr_1) \vee \mathrm{o2f}_{\mathrm{inval}}(expr_2) \\ &\quad \vee (\mathrm{o2f}_{\mathrm{null}}(expr_1) \wedge \neg\mathrm{o2f}_{\mathrm{null}}(expr_2)) \vee (\neg\mathrm{o2f}_{\mathrm{null}}(expr_1) \wedge \mathrm{o2f}_{\mathrm{null}}(expr_2)).\end{aligned}$$

not-expressions:

$\mathrm{o2f}_{\mathrm{true}}(\mathsf{not}(expr)) = \mathrm{o2f}_{\mathrm{false}}(expr).$

$\mathrm{o2f}_{\mathrm{false}}(\mathsf{not}(expr)) = \mathrm{o2f}_{\mathrm{true}}(expr).$

$\mathrm{o2f}_{\mathrm{null}}(\mathsf{not}(expr)) = \mathrm{o2f}_{\mathrm{null}}(expr).$

$\mathrm{o2f}_{\mathrm{inval}}(\mathsf{not}(expr)) = \mathrm{o2f}_{\mathrm{inval}}(expr).$

and-expressions:

$\mathrm{o2f}_{\mathrm{true}}((\mathit{expr}_1 \text{ and } \mathit{expr}_2)) = \mathrm{o2f}_{\mathrm{true}}(\mathit{expr}_1) \wedge \mathrm{o2f}_{\mathrm{true}}(\mathit{expr}_2)$.

$\mathrm{o2f}_{\mathrm{false}}((\mathit{expr}_1 \text{ and } \mathit{expr}_2)) = \mathrm{o2f}_{\mathrm{false}}(\mathit{expr}_1) \vee \mathrm{o2f}_{\mathrm{false}}(\mathit{expr}_2)$.

$\mathrm{o2f}_{\mathrm{null}}(\mathit{expr}_1 \text{ and } \mathit{expr}_2) = \mathrm{o2f}_{\mathrm{null}}(\mathit{expr}_1) \wedge \mathrm{o2f}_{\mathrm{null}}(\mathit{expr}_2)$
$\quad \vee\ (\mathrm{o2f}_{\mathrm{null}}(\mathit{expr}_1) \wedge \mathrm{o2f}_{\mathrm{true}}(\mathit{expr}_2)) \vee\ (\mathrm{o2f}_{\mathrm{true}}(\mathit{expr}_1) \wedge \mathrm{o2f}_{\mathrm{null}}(\mathit{expr}_2))$.

$\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_1 \text{ and } \mathit{expr}_2) =$
$\quad (\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_1)\ \wedge (\mathrm{o2f}_{\mathrm{true}}(\mathit{expr}_2) \vee \mathrm{o2f}_{\mathrm{null}}(\mathit{expr}_2) \vee\ \mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_2)))$
$\quad \vee\ (\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_2)\ \wedge (\mathrm{o2f}_{\mathrm{true}}(\mathit{expr}_1) \vee \mathrm{o2f}_{\mathrm{null}}(\mathit{expr}_1) \vee\ \mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_1)))$.

or-expressions:

$\mathrm{o2f}_{\mathrm{true}}((\mathit{expr}_1 \text{ or } \mathit{expr}_2)) = \mathrm{o2f}_{\mathrm{true}}(\mathit{expr}_1) \vee \mathrm{o2f}_{\mathrm{true}}(\mathit{expr}_2)$.

$\mathrm{o2f}_{\mathrm{false}}((\mathit{expr}_1 \text{ or } \mathit{expr}_2)) = \mathrm{o2f}_{\mathrm{false}}(\mathit{expr}_1) \wedge \mathrm{o2f}_{\mathrm{false}}(\mathit{expr}_2)$.

$\mathrm{o2f}_{\mathrm{null}}((\mathit{expr}_1 \text{ or } \mathit{expr}_2) = \mathrm{o2f}_{\mathrm{null}}(\mathit{expr}_1)) \wedge \mathrm{o2f}_{\mathrm{null}}(\mathit{expr}_2)$
$\quad \vee\ (\mathrm{o2f}_{\mathrm{null}}(\mathit{expr}_1) \wedge \mathrm{o2f}_{\mathrm{false}}(\mathit{expr}_2)) \vee\ (\mathrm{o2f}_{\mathrm{false}}(\mathit{expr}_1) \wedge \mathrm{o2f}_{\mathrm{null}}(\mathit{expr}_2))$.

$\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_1 \text{ or } \mathit{expr}_2) =$
$\quad (\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_1)\ \wedge (\mathrm{o2f}_{\mathrm{false}}(\mathit{expr}_2) \vee \mathrm{o2f}_{\mathrm{null}}(\mathit{expr}_2) \vee\ \mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_2)))$
$\quad \vee\ (\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_2)\ \wedge (\mathrm{o2f}_{\mathrm{false}}(\mathit{expr}_1) \vee \mathrm{o2f}_{\mathrm{null}}(\mathit{expr}_1) \vee\ \mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_1)))$.

implies-expressions:

$\mathrm{o2f}_{\mathrm{true}}((\mathit{expr}_1 \text{ implies } \mathit{expr}_2)) = \mathrm{o2f}_{\mathrm{false}}(\mathit{expr}_1) \vee \mathrm{o2f}_{\mathrm{true}}(\mathit{expr}_2)$.

$\mathrm{o2f}_{\mathrm{false}}((\mathit{expr}_1 \text{ implies } \mathit{expr}_2)) = \mathrm{o2f}_{\mathrm{true}}(\mathit{expr}_1) \wedge \mathrm{o2f}_{\mathrm{false}}(\mathit{expr}_2)$.

$\mathrm{o2f}_{\mathrm{null}}(\mathit{expr}_1 \text{ implies } \mathit{expr}_2) =$
$\quad (\mathrm{o2f}_{\mathrm{null}}(\mathit{expr}_1)\ \wedge (\mathrm{o2f}_{\mathrm{true}}(\mathit{expr}_2) \vee \mathrm{o2f}_{\mathrm{null}}(\mathit{expr}_2) \vee\ \mathrm{o2f}_{\mathrm{false}}(\mathit{expr}_2)))$
$\quad \vee\ (\mathrm{o2f}_{\mathrm{null}}(\mathit{expr}_2)\ \wedge (\mathrm{o2f}_{\mathrm{true}}(\mathit{expr}_1) \vee \mathrm{o2f}_{\mathrm{null}}(\mathit{expr}_1) \vee \mathrm{o2f}_{\mathrm{false}}(\mathit{expr}_1)))$.

$\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_1 \text{ implies } \mathit{expr}_2) = (\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_1)\ \vee \mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_1))$.

isEmpty-expressions:

$\mathrm{o2f}_{\mathrm{true}}(\mathit{expr}\text{->isEmpty}()) =$
$\quad \forall(x)(\neg(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}), \mathrm{fVars}(\mathit{expr}), x)) \wedge \neg(\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr})))$.

$\mathrm{o2f}_{\mathrm{false}}(\mathit{expr}\text{->isEmpty}()) =$
$\quad \exists(x)(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}), \mathrm{fVars}(\mathit{expr}), x)) \wedge\ \neg(\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}))$.

$\mathrm{o2f}_{\mathrm{null}}(\mathit{expr}\text{->isEmpty}()) = \bot$.

$\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}\text{->isEmpty}()) = \mathrm{o2f}_{\mathrm{inval}}(\mathit{expr})$.

notEmpty-expressions:

$$\begin{aligned}&\mathrm{o2f}_{\mathrm{true}}(\mathit{expr}{-}{>}\mathsf{notEmpty}()) =\\&\quad\exists(x)(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}), \mathrm{fVars}(\mathit{expr}), x)) \wedge\ \neg(\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr})).\end{aligned}$$

$$\begin{aligned}&\mathrm{o2f}_{\mathrm{false}}(\mathit{expr}{-}{>}\mathsf{notEmpty}()) =\\&\quad\forall(x)(\neg(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}), \mathrm{fVars}(\mathit{expr}), x)) \wedge\ \neg(\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr})).\end{aligned}$$

$$\mathrm{o2f}_{\mathrm{null}}(\mathit{expr}{-}{>}\mathsf{notEmpty}()) = \bot.$$

$$\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}{-}{>}\mathsf{notEmpty}()) = \mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}).$$

forAll-expressions:

$$\begin{aligned}&\mathrm{o2f}_{\mathrm{true}}(\mathit{src}{-}{>}\mathsf{forall}(x \mid \mathit{body})) =\\&\quad\forall(x)(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}), \mathrm{fVars}(\mathit{src}), x) \Rightarrow \mathrm{o2f}_{\mathrm{true}}(\mathit{body})) \wedge \neg(\mathrm{o2f}_{\mathrm{inval}}(\mathit{src})).\end{aligned}$$

$$\begin{aligned}&\mathrm{o2f}_{\mathrm{false}}(\mathit{src}{-}{>}\mathsf{forall}(x \mid \mathit{body})) =\\&\quad\exists(x)(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}), \mathrm{fVars}(\mathit{src}), x) \wedge\ \mathrm{o2f}_{\mathrm{false}}(\mathit{body})) \wedge \neg(\mathrm{o2f}_{\mathrm{inval}}(\mathit{src})).\end{aligned}$$

$$\begin{aligned}&\mathrm{o2f}_{\mathrm{null}}(\mathit{src}{-}{>}\mathsf{forAll}(x \mid \mathit{body})) = \neg\ \mathrm{o2f}_{\mathrm{inval}}(\mathit{src})\\&\quad\wedge\ \exists(x)(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}), \mathrm{fVars}(\mathit{src}), x) \wedge\ \mathrm{o2f}_{\mathrm{null}}(\mathit{body}))\\&\quad\wedge\ \forall(x)(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}), \mathrm{fVars}(\mathit{src}), x) \Rightarrow (\mathrm{o2f}_{\mathrm{true}}(\mathit{body}) \vee\ \mathrm{o2f}_{\mathrm{null}}(\mathit{body}).\end{aligned}$$

$$\begin{aligned}&\mathrm{o2f}_{\mathrm{inval}}(\mathit{src}{-}{>}\mathsf{forAll}(x \mid \mathit{body}), \vec{v}) =\\&\quad\mathrm{o2f}_{\mathrm{inval}}(\mathit{src}, \vec{v}) \vee\ \exists(x)([\mathit{src}]^{\flat}(\vec{v}, x) \wedge \mathrm{o2f}_{\mathrm{inval}}(\mathit{body}[\mathsf{x} \mapsto x], \vec{v}))\\&\quad\wedge\ \forall(x)([\mathit{src}]^{\flat}(\vec{v}, x) \Rightarrow (\mathrm{o2f}_{\mathrm{true}}(\mathit{body}[\mathsf{x} \mapsto x], \vec{v})\vee\\&\qquad\quad\mathrm{o2f}_{\mathrm{null}}(\mathit{body}[\mathsf{x} \mapsto x], \vec{v}) \vee\ \mathrm{o2f}_{\mathrm{inval}}(\mathit{body}[\mathsf{x} \mapsto x], \vec{v}))).\end{aligned}$$

exists-expressions:

$$\begin{aligned}&\mathrm{o2f}_{\mathrm{true}}(\mathit{src}{-}{>}\mathsf{exists}(x \mid \mathit{body})) =\\&\quad\exists(x)(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}), \mathrm{fVars}(\mathit{src}), x) \wedge\ \mathrm{o2f}_{\mathrm{true}}(\mathit{body})) \wedge \neg(\mathrm{o2f}_{\mathrm{inval}}(\mathit{src})).\end{aligned}$$

$$\begin{aligned}&\mathrm{o2f}_{\mathrm{false}}(\mathit{src}{-}{>}\mathsf{exists}(x \mid \mathit{body})) =\\&\quad\forall(x)(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}), \mathrm{fVars}(\mathit{src}), x) \Rightarrow \mathrm{o2f}_{\mathrm{false}}(\mathit{body})) \wedge \neg(\mathrm{o2f}_{\mathrm{inval}}(\mathit{src})).\end{aligned}$$

$$\begin{aligned}&\mathrm{o2f}_{\mathrm{null}}(\mathit{src}{-}{>}\mathsf{exists}(x \mid \mathit{body})) = \neg(\mathrm{o2f}_{\mathrm{inval}}(\mathit{src}))\\&\quad\wedge\ \exists(x)(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}), \mathrm{fVars}(\mathit{src}), x) \wedge\ \mathrm{o2f}_{\mathrm{null}}(\mathit{body}))\\&\quad\wedge\ \forall(x)(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}), \mathrm{fVars}(\mathit{src}), x) \Rightarrow (\mathrm{o2f}_{\mathrm{false}}(\mathit{body}) \vee\ \mathrm{o2f}_{\mathrm{null}}(\mathit{body}).\end{aligned}$$

$$\begin{aligned}&\mathrm{o2f}_{\mathrm{inval}}(\mathit{src}{-}{>}\mathsf{exists}(x \mid \mathit{body})) = \mathrm{o2f}_{\mathrm{inval}}(\mathit{src})\\&\quad\vee\ \exists(x)(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}), \mathrm{fVars}(\mathit{src}), x) \wedge \mathrm{o2f}_{\mathrm{inval}}(\mathit{body}))\\&\quad\wedge\ \forall(x)(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}), \mathrm{fVars}(\mathit{src}), x)\\&\qquad\Rightarrow (\mathrm{o2f}_{\mathrm{false}}(\mathit{body}) \vee\ \mathrm{o2f}_{\mathrm{null}}(\mathit{body}) \vee\ \mathrm{o2f}_{\mathrm{inval}}(\mathit{body}))).\end{aligned}$$

excludes-expressions:

$$\begin{aligned}&\mathrm{o2f}_{\mathrm{true}}(\mathit{expr}_1\text{->}\mathsf{excludes}(\mathit{expr}_2)) =\\ &\quad \forall(x)(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_1), \mathrm{fVars}(\mathit{expr}_1), x) \Rightarrow x \neq \mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_2), \mathrm{fVars}(\mathit{expr}_2), []))\\ &\qquad \wedge \neg\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_1) \wedge \neg\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_2).\end{aligned}$$

$$\begin{aligned}&\mathrm{o2f}_{\mathrm{false}}(\mathit{expr}_1\text{->}\mathsf{excludes}(\mathit{expr}_2)) =\\ &\quad \exists(x)(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_1), \mathrm{fVars}(\mathit{expr}_1), x) \wedge\ x = \mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_2), \mathrm{fVars}(\mathit{expr}_2), []))\\ &\qquad \wedge \neg\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_1) \wedge \neg\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_2).\end{aligned}$$

$$\mathrm{o2f}_{\mathrm{null}}(\mathit{expr}_1\text{->}\mathsf{excludes}(\mathit{expr}_2)) = \bot.$$

$$\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_1\text{->}\mathsf{excludes}(\mathit{expr}_2))) = \mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_1) \vee \mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_2).$$

includes-expressions:

$$\begin{aligned}&\mathrm{o2f}_{\mathrm{true}}(\mathit{expr}_1\text{->}\mathsf{includes}(\mathit{expr}_2)) =\\ &\quad \exists(x)(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_1), \mathrm{fVars}(\mathit{expr}_1), x) \wedge\ x = \mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_2), \mathrm{fVars}(\mathit{expr}_2), []))\\ &\qquad \wedge \neg\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_1) \wedge \neg\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_2).\end{aligned}$$

$$\begin{aligned}&\mathrm{o2f}_{\mathrm{false}}(\mathit{expr}_1\text{->}\mathsf{includes}(\mathit{expr}_2)) =\\ &\quad \forall(x)(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_1), \mathrm{fVars}(\mathit{expr}_1), x) \Rightarrow x \neq \mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_2), \mathrm{fVars}(\mathit{expr}_2), []))\\ &\qquad \wedge \neg\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_1) \wedge \neg\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_2).\end{aligned}$$

$$\mathrm{o2f}_{\mathrm{null}}(\mathit{expr}_1\text{->}\mathsf{includes}(\mathit{expr}_2)) = \bot.$$

$$\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_1\text{->}\mathsf{includes}(\mathit{expr}_2)) = \mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_1) \vee \mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_2).$$

=-expressions (between sets):

$$\begin{aligned}&\mathrm{o2f}_{\mathrm{true}}((\mathit{expr}_1 = \mathit{expr}_2)) =\\ &\quad \forall(x)(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_1), \mathrm{fVars}(\mathit{expr}_1), x) \Leftrightarrow \mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_2), \mathrm{fVars}(\mathit{expr}_2), x))\\ &\qquad \wedge \neg\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_1) \wedge \neg\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_2).\end{aligned}$$

$$\begin{aligned}&\mathrm{o2f}_{\mathrm{false}}((\mathit{expr}_1 = \mathit{expr}_2)) =\\ &\quad \exists(x)((\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_1), \mathrm{fVars}(\mathit{expr}_1), x) \wedge \neg(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_2), \mathrm{fVars}(\mathit{expr}_2), x)))\\ &\qquad \vee\ (\neg(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_1), \mathrm{fVars}(\mathit{expr}_1), x) \wedge \mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_2), \mathrm{fVars}(\mathit{expr}_2), x))))\\ &\qquad \wedge \neg\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_1) \wedge \neg\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_2)).\end{aligned}$$

$$\mathrm{o2f}_{\mathrm{null}}((\mathit{expr}_1 = \mathit{expr}_2)) = \bot.$$

$$\mathrm{o2f}_{\mathrm{inval}}((\mathit{expr}_1 = \mathit{expr}_2)) = \mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_1) \vee \mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_2).$$

<>-expressions (between sets):

$$\begin{aligned}&\mathrm{o2f}_{\mathrm{true}}((\mathit{expr}_1 <> \mathit{expr}_2)) =\\ &\quad \exists(x)((\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_1), \mathrm{fVars}(\mathit{expr}_1), x) \wedge \neg(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_2), \mathrm{fVars}(\mathit{expr}_2), x)))\\ &\qquad \vee\ (\neg(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_1), \mathrm{fVars}(\mathit{expr}_1), x) \wedge \mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_2), \mathrm{fVars}(\mathit{expr}_2), x))))\\ &\qquad \wedge \neg\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_1) \wedge \neg\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_2)).\end{aligned}$$

$$\begin{aligned}&\mathrm{o2f}_{\mathrm{false}}((\mathit{expr}_1 <> \mathit{expr}_2)) =\\ &\quad \forall(x)(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_1), \mathrm{fVars}(\mathit{expr}_1), x) \Leftrightarrow\ \mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_2), \mathrm{fVars}(\mathit{expr}_2), x))\\ &\qquad \wedge \neg\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_1) \wedge \neg\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_2).\end{aligned}$$

$$\mathrm{o2f}_{\mathrm{null}}((\mathit{expr}_1 <> \mathit{expr}_2)) = \bot.$$

$$\mathrm{o2f}_{\mathrm{inval}}((\mathit{expr}_1 <> \mathit{expr}_2)) = \mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_1) \vee \mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_2).$$

includesAll-expressions:

$$\begin{aligned}&\mathrm{o2f}_{\mathrm{true}}(\mathit{expr}_1\text{->}\mathsf{includesAll}(\mathit{expr}_2)) =\\ &\quad \forall(x)(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_2), \mathrm{fVars}(\mathit{expr}_2), x) \Rightarrow \mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_1), \mathrm{fVars}(\mathit{expr}_1), x))\\ &\qquad \wedge \neg\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_1) \wedge \neg\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_2).\end{aligned}$$

$$\begin{aligned}&\mathrm{o2f}_{\mathrm{false}}(\mathit{expr}_1\text{->}\mathsf{includesAll}(\mathit{expr}_2)) =\\ &\quad \exists(x)(\mathrm{App}([\mathit{expr}_2]^{\flat}, \mathrm{fVars}(\mathit{expr}_2), x) \wedge \neg(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_1), \mathrm{fVars}(\mathit{expr}_1), x)))\\ &\qquad \wedge \neg\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_1) \wedge \neg\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_2).\end{aligned}$$

$$\mathrm{o2f}_{\mathrm{null}}(\mathit{expr}_1\text{->}\mathsf{includesAll}(\mathit{expr}_2)) = \bot.$$

$$\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_1\text{->}\mathsf{includesAll}(\mathit{expr}_2)) = \mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_1) \vee \mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_2).$$

excludesAll-expressions:

$$\begin{aligned}&\mathrm{o2f}_{\mathrm{true}}(\mathit{expr}_1\text{->}\mathsf{excludesAll}(\mathit{expr}_2)) =\\ &\quad \forall(x)(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_2), \mathrm{fVars}(\mathit{expr}_2), x) \Rightarrow \neg(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_1), \mathrm{fVars}(\mathit{expr}_1), x)))\\ &\qquad \wedge \neg\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_1) \wedge \neg\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_2).\end{aligned}$$

$$\begin{aligned}&\mathrm{o2f}_{\mathrm{false}}(\mathit{expr}_1\text{->}\mathsf{excludesAll}(\mathit{expr}_2)) =\\ &\quad \exists(x)(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_2), \mathrm{fVars}(\mathit{expr}_2), x) \wedge \mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_1), \mathrm{fVars}(\mathit{expr}_1), x))\\ &\qquad \wedge \neg\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_1) \wedge \neg\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_2).\end{aligned}$$

$$\mathrm{o2f}_{\mathrm{null}}(\mathit{expr}_1\text{->}\mathsf{excludesAll}(\mathit{expr}_2)) = \bot.$$

$$\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_1\text{->}\mathsf{excludesAll}(\mathit{expr}_2)) = \mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_1) \vee \mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_2).$$

oclIsUndefined-expressions (over sets):

$$\mathrm{o2f}_{\mathrm{true}}(\mathit{src}\text{->}\mathsf{oclIsUndefined()}) = \bot.$$

$$\mathrm{o2f}_{\mathrm{false}}(\mathit{src}\text{->}\mathsf{oclIsUndefined()}) = \top.$$

$$\mathrm{o2f}_{\mathrm{null}}(\mathit{expr}\text{->}\mathsf{oclIsUndefined()}) = \bot.$$

$$\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}\text{->}\mathsf{oclIsUndefined()}) = \bot.$$

oclIsInvalid-expressions (over sets):

$$\mathrm{o2f}_{\mathrm{true}}(\mathit{src}\text{->}\mathsf{oclIsInvalid()}) = \mathrm{o2f}_{\mathrm{inval}}(\mathit{src}).$$

$$\mathrm{o2f}_{\mathrm{false}}(\mathit{src}\text{->}\mathsf{oclIsInvalid()}) = \neg(\mathrm{o2f}_{\mathrm{inval}}(\mathit{src})).$$

$$\mathrm{o2f}_{\mathrm{null}}(\mathit{expr}\text{->}\mathsf{oclIsInvalid()}) = \bot.$$

$$\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}\text{->}\mathsf{oclIsInvalid()}) = \bot.$$

Now, consider expr, expr_1, expr_2 be non boolean expressions, and src be expression of the appropriate type. We define the mappings $\mathrm{o2f}_{\mathrm{true}}$*,* $\mathrm{o2f}_{\mathrm{false}}$ *to evaluate always to* $\top$ *(because these expressions are non-Boolean expressions), and* $\mathrm{o2f}_{\mathrm{null}}$*, and* $\mathrm{o2f}_{\mathrm{inval}}$ *as follow:*

integer-expressions (literals):

$\mathrm{o2f}_{\mathrm{null}}(i) = \bot.$

$\mathrm{o2f}_{\mathrm{inval}}(i) = \bot.$

variable-expressions:

$\mathrm{o2f}_{\mathrm{null}}(v_t) = (v_t = \mathrm{null}_t).$

$\mathrm{o2f}_{\mathrm{inval}}(v_t) = (v_t = \mathrm{inval}_t).$

--expressions (unary):

$\mathrm{o2f}_{\mathrm{null}}(\textsf{-}(\mathit{expr})) = \bot.$

$\mathrm{o2f}_{\mathrm{inval}}(\textsf{-}(\mathit{expr})) = \mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}) \vee \mathrm{o2f}_{\mathrm{null}}(\mathit{expr}).$

***op* ∈ {+, -, *, div, concat, indexOf, at}-expressions:**

$\mathrm{o2f}_{\mathrm{null}}(\mathit{expr}_1\ \mathit{op}\ \mathit{expr}_2, \vec{v}) = \bot.$

$\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}\ \mathit{op}\ \mathit{expr}', \vec{v}) =$
$\quad \mathrm{o2f}_{\mathrm{null}}(\mathit{expr}_1, \vec{v}) \vee \mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_1, \vec{v}) \vee \ \mathrm{o2f}_{\mathrm{null}}(\mathit{expr}_2, \vec{v}) \vee \mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_2, \vec{v}).$
for op $\in \{\textsf{+}, \textsf{-}, \textsf{*}, \textsf{div}\}$.

size-expressions:

$\mathrm{o2f}_{\mathrm{null}}(\mathit{expr}.\textsf{size()}, \vec{v}) = \bot.$

$\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}.\textsf{size()}, \vec{v}) = \mathrm{o2f}_{\mathrm{null}}(\mathit{expr}, \vec{v}) \vee \mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}).$

substring-expressions:

$\mathrm{o2f}_{\mathrm{null}}(\mathit{expr}_1.\textsf{substring(}\mathit{expr}_2, \mathit{expr}_3\textsf{)}), \vec{v}) = \bot.$

$\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}.\textsf{substring(}\mathit{expr}', \mathit{expr}''\textsf{)}, \vec{v}) = \mathrm{o2f}_{\mathrm{null}}(\mathit{expr}_1, \vec{v}) \vee \mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_1, \vec{v})$
$\quad \vee\ \mathrm{o2f}_{\mathrm{null}}(\mathit{expr}_2, \vec{v}) \vee \mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_2, \vec{v}) \vee \ \mathrm{o2f}_{\mathrm{null}}(\mathit{expr}_3, \vec{v}) \vee \mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_3, \vec{v}).$

allInstances-expressions:

$\mathrm{o2f}_{\mathrm{null}}(c.\textsf{allInstances()}) = \bot.$

$\mathrm{o2f}_{\mathrm{inval}}(c.\textsf{allInstances()}) = \bot.$

attribute-expressions:

$\mathrm{o2f}_{\mathrm{null}}(\mathit{expr}.\mathit{at}) = (\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}.\mathit{at}) = \mathit{nullt}).$
where $t \in \mathit{Integer}, \mathit{String}$ *or* t *is a class type.*

$\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}.\mathit{at}) = \mathrm{o2f}_{\mathrm{null}}(\mathit{expr}) \vee \mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}).$

association-end-expressions (arity 0..1):

$\mathrm{o2f}_{\mathrm{null}}(\mathit{expr}.\mathit{as}) = (\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}.\mathit{as}) = \mathit{nullt})$.

$\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}.\mathit{as}) = \mathrm{o2f}_{\mathrm{null}}(\mathit{expr}) \vee \mathrm{o2f}_{\mathrm{inval}}(\mathit{expr})$.
where $t \in \mathit{Integer}, \mathit{String}$ *or* t *is a class type.*

$\mathrm{o2f}_{\mathrm{null}}(\mathit{expr}.\mathit{as}()) = \bot$.

$\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}.\mathit{as}()) = \mathrm{o2f}_{\mathrm{inval}}(\mathit{exp}) \vee \mathrm{o2f}_{\mathrm{null}}(\mathit{exp})$.

max-expressions:

$\mathrm{o2f}_{\mathrm{null}}(\mathit{src}\text{->}\mathsf{max}()) = (\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}\text{->}\mathsf{max}()), \mathrm{fVars}(\mathit{src}), []) = \mathit{nullInt})$.

$\mathrm{o2f}_{\mathrm{inval}}(\mathit{src}\text{->}\mathsf{max}()) = (\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}\text{->}\mathsf{max}()), \mathrm{fVars}(\mathit{src}), []) = \mathrm{invalInt}))$.

min-expressions:

$\mathrm{o2f}_{\mathrm{null}}(\mathit{src}\text{->}\mathsf{min}()) = (\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}\text{->}\mathsf{min}()), \mathrm{fVars}(\mathit{src}), []) = \mathit{nullInt})$.

$\mathrm{o2f}_{\mathrm{inval}}(\mathit{src}\text{->}\mathsf{min}()) = (\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}\text{->}\mathsf{min}()), \mathrm{fVars}(\mathit{src}), []) = \mathit{invalInt})$.

any-expressions:

$\mathrm{o2f}_{\mathrm{null}}(\mathit{src}\text{->}\mathsf{any}(x_t|\mathit{body})) =$
$(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}\text{->}\mathsf{any}(x_t|\mathit{body})), \mathrm{fVars}(\mathit{src}), x_t) = \mathrm{null}(t))$.

$\mathrm{o2f}_{\mathrm{inval}}(\mathit{src}\text{->}\mathsf{any}(x_t|\mathit{body})) =$
$(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}\text{->}\mathsf{any}(x_t|\mathit{body})), \mathrm{fVars}(\mathit{src}), x_t) = \mathrm{inval}(t))$.

$\mathit{op} \in$ **{select, reject}-expressions:**

$\mathrm{o2f}_{\mathrm{null}}(\mathit{src}\text{->}\mathit{op}(p \mid \mathit{body})) = \bot$.

$\mathrm{o2f}_{\mathrm{inval}}(\mathit{src}\text{->}\mathit{op}(p \mid \mathit{body})) = \mathrm{o2f}_{\mathrm{inval}}(\mathit{src})$.

$\mathit{op} \in$ **{including, excluding, union}-expressions:**

$\mathrm{o2f}_{\mathrm{null}}(\mathit{expr}_1)\text{->}\mathit{op}(\mathit{expr}_2) = \mathrm{o2f}_{\mathrm{null}}(\mathit{expr}_1) \vee \mathrm{o2f}_{\mathrm{null}}(\mathit{expr}_2)$.

$\mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_1)\text{->}\mathit{op}(\mathit{expr}_2) = \mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_1) \vee \mathrm{o2f}_{\mathrm{inval}}(\mathit{expr}_2)$.

collect-expressions:

$\mathrm{o2f}_{\mathrm{null}}(\mathit{src}\text{->}\mathsf{collect(x|}\mathit{body}\mathsf{)}) = \bot.$

$\mathrm{o2f}_{\mathrm{inval}}(\mathit{src}\text{->}\mathsf{collect(x|}\mathit{body}\mathsf{)}) =$
$\quad \mathrm{o2f}_{\mathrm{inval}}(\mathit{src}) \vee \exists(x)(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}), \mathrm{fVars}(\mathit{src}), x) \wedge \mathrm{o2f}_{\mathrm{inval}}(\mathit{body})).$

Finally, we illustrate the recursive definitions of the mappings $\mathrm{o2f}_{\mathrm{true}}$, $\mathrm{o2f}_{\mathrm{false}}$, $\mathrm{o2f}_{\mathrm{null}}$, and $\mathrm{o2f}_{\mathrm{inval}}$ with some examples:

Example 13 *Consider the Boolean expression:*

Profile.allInstances()->notEmpty().

Then,

$$\begin{aligned}
&\mathrm{o2f}_{\mathrm{true}}(\mathsf{Profile.allInstances()\text{->}notEmpty()})\\
&\quad = \exists(x{:}\mathit{Classifier})(\mathrm{o2f}_{\mathrm{eval}}(\mathsf{Profile.allInstances()})(x)) \wedge \neg(\mathrm{o2f}_{\mathrm{inval}}(\mathsf{Profile.allInstances()}))\\
&\quad = \exists(x{:}\mathit{Classifier})(\mathrm{o2f}_{\mathrm{eval}}(\mathsf{Profile.allInstances()})(x)) \wedge \neg(\bot).
\end{aligned}$$

Example 14 *Consider the Boolean expression:*

Profile.allInstances()->forAll(p|not(p.age.oclIsUndefined())).

Then,

$$\begin{aligned}
&\mathrm{o2f}_{\mathrm{false}}(\mathsf{Profile.allInstances()\text{->}forAll(p|not(p.age.oclIsUndefined()))})\\
&\quad = \exists(p{:}\mathit{Classifier})(\mathrm{o2f}_{\mathrm{eval}}(\mathsf{Profile.allInstances()})(p)\\
&\qquad\qquad \wedge\ \mathrm{o2f}_{\mathrm{false}}(\mathsf{not(p.age.oclIsUndefined())}))\\
&\qquad \wedge\ \neg(\mathrm{o2f}_{\mathrm{inval}}(\mathsf{Profile.allInstances()}))),
\end{aligned}$$

where

$$\begin{aligned}
&\mathrm{o2f}_{\mathrm{false}}(\mathsf{not(p.age.oclIsUndefined())})\\
&\quad = \mathrm{o2f}_{\mathrm{true}}(\mathsf{p.age.oclIsUndefined()})\\
&\quad = \mathrm{o2f}_{\mathrm{null}}(\mathsf{p.age}) \vee \mathrm{o2f}_{\mathrm{inval}}(\mathsf{p.age})\\
&\quad = \mathrm{o2f}_{\mathrm{eval}}(\mathsf{p.age}) = \mathit{nullClassifier} \vee (\mathrm{o2f}_{\mathrm{null}}(\mathsf{p}) \vee \mathrm{o2f}_{\mathrm{inval}}(\mathsf{p}))\\
&\quad = \mathrm{o2f}_{\mathrm{eval}}(\mathsf{p.age}) = \mathit{nullClassifier} \vee (p = \mathit{nullClassifier} \vee p = \mathit{invalClassifier}).
\end{aligned}$$

Example 15 *Consider the Boolean expression:*

Profile.allInstances()->select(p|p.age.oclIsUndefined())->notEmpty().

Then,

$$\begin{aligned}
&\mathrm{o2f}_{\mathrm{true}}(\mathsf{Profile.allInstances()\text{->}select(p|p.age.oclIsUndefined())\text{->}notEmpty()})\\
&\quad = \exists(x{:}\mathit{Classifier})(\mathrm{o2f}_{\mathrm{eval}}(\mathsf{Profile.allInstances()\text{->}select(p|p.age.oclIsUndefined())})(x))\\
&\qquad \wedge\ \neg(\mathrm{o2f}_{\mathrm{inval}}(\mathsf{Profile.allInstances()\text{->}select(p|p.age.oclIsUndefined())}))\\
&\quad = \exists(x{:}\mathit{Classifier})(\mathrm{o2f}_{\mathrm{eval}}(\mathsf{Profile.allInstances()\text{->}select(p|p.age.oclIsUndefined())})(x))\\
&\qquad \wedge\ \neg(\mathrm{o2f}_{\mathrm{inval}}(\mathsf{Profile.allInstances()}))\\
&\quad = \exists(x{:}\mathit{Classifier})(\mathrm{o2f}_{\mathrm{eval}}(\mathsf{Profile.allInstances()\text{->}select(p|p.age.oclIsUndefined())})(x))\\
&\qquad \wedge\ \neg(\bot).
\end{aligned}$$

The mapping $\mathrm{o2f}_{\mathrm{eval}}$, $\mathrm{o2f}_{\mathrm{def\_c}}$, and $\mathrm{o2f}_{\mathrm{def\_o}}$

In the definition of the mapping $\mathrm{o2f}_{\mathrm{eval}}$ we distinguish three classes of non-Boolean expressions. In a nutshell, the differences between these classes are the followings:

- The first class is formed by variables and by expressions that *denote primitive values and objects*. Expressions denoting primitive values and objects are basically the literals (integers or strings), the arithmetic expressions, the expressions denoting operations on strings, and the dot-expressions for attributes or association-ends with multiplicity 0..1. Variables are mapped to MSFOL variables of the appropriate sort. Expressions denoting primitive values and objects are mapped by $\mathrm{o2f}_{\mathrm{eval}}$ following the definition of the mapping $\mathrm{o2f}_{\mathrm{data}}$. The output of the mapping $\mathrm{o2f}_{\mathrm{eval}}$ for this first class of non-Boolean expressions is always an MSFOL term.

- The second class of non-Boolean expressions is formed by the expressions that *define sets*. These expressions are basically the allInstances-expressions, the select and reject-expressions, the including and excluding-expressions, the intersection and union-expressions, and the collect-expressions. Each expression *expr* in this class is mapped by $\mathrm{o2f}_{\mathrm{eval}}$ to a *new* predicate, denoted as [*expr*]. This predicate formalizes the set defined by the expression *expr* and its definition is generated by calling another mapping, $\mathrm{o2f}_{\mathrm{def\_c}}$, over the expression *expr*.

- Finally, the third class of non-Boolean expressions is formed by the expressions that *distinguish an element from a set*. These expressions are, basically, the any, max, and min-expressions. Each expression *expr* in this class is mapped by $\mathrm{o2f}_{\mathrm{eval}}$ to a *new* function, denoted as [*expr*], which represents the element referred to by *expr*. To generate the axioms defining [*expr*], we call another mapping, $\mathrm{o2f}_{\mathrm{def\_o}}$, over *expr*.

Definition 5 *Let expr, $expr_1$, $expr_2$, and src be expression of the appropriate type, we define the mappings* $\mathrm{o2f}_{\mathrm{eval}}$*, as follow:*

integer-expressions (literals):

$\mathrm{o2f}_{\mathrm{eval}}(i) = i.$

variable-expressions:

$\mathrm{o2f}_{\mathrm{eval}}(v_t) = v_t.$

allInstances-expressions:

$\mathrm{o2f}_{\mathrm{eval}}(c.\mathsf{allInstances}()) = [c].$

association-end-expressions (multiplicity 0..1 or 1):

$\mathrm{o2f}_{\mathrm{eval}}(expr.as) = as(\mathrm{o2f}_{\mathrm{eval}}(expr), as).$

attribute-expressions:

$\mathrm{o2f}_{\mathrm{eval}}(expr.at) = at(\mathrm{o2f}_{\mathrm{eval}}(expr), at).$

$it \in$ {select, reject, collect}-expressions:

$\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}\text{->}\mathit{it}(x|\mathit{body})) = [\mathit{src}\text{->}\mathit{it}(x|\mathit{body})]$.

$op \in$ {including, excluding, union, intersection, *set-difference*, symmetricDifference}-expressions:

$\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_1\text{->}\mathit{op}(\mathit{expr}_2)) = [\mathit{src}\text{->}\mathit{op}()]$.

$op \in$ {max, min}-expressions:

$\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}\text{->}\mathit{op}()) = [\mathit{src}\text{->}\mathit{op}()]$.

any-expressions:

$\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}\text{->}\mathsf{any}(x|\mathit{body})) = [\mathit{src}\text{->}\mathsf{any}(x|\mathit{body})]$.

--expressions (unary):

$\mathrm{o2f}_{\mathrm{eval}}(\text{-}(\mathit{expr})) = -(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}))$.

$op \in$ {+, -, *, div}-expressions:

$\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_1\ \mathit{op}\ \mathit{expr}_2) = \mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_1)\ \mathit{op}\ \mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_2)$.

$op \in$ {+, concat}-expressions:

$\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_1\ \mathit{op}\ \mathit{expr}_2) = \mathrm{str.concat}\ \mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_1)\ \mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_2)$.

$op \in$ {size}-expressions:

$\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}\text{->}\mathsf{size}()) = \mathrm{str.len}\ \mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_1)\ \mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_2)$.

$op \in$ {at}-expressions:

$\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_1.\mathsf{at}(\mathit{expr}_2)) = \mathrm{str.at}\ \mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_1)\ \mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_2)$.

$op \in$ {indexOf}-expressions:

$\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_1\ \mathit{op}\ \mathit{expr}_2) = \mathrm{str.indexOf}\ \mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_1)\ \mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_2)\ 0$.

substring-expressions:

$$\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_1.\mathsf{substring}(\mathit{expr}_2, \mathit{expr}_3)) = \\ \quad \mathrm{str.substr}\ \mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_1)\ \mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_2)(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_2) + \mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_3)).$$

Example 16 *Consider the non-Boolean expression:* p.age*, where* p *is a variable of type* Profile*. Then,*

$\mathrm{o2f}_{\mathrm{eval}}(\mathsf{p.age}) = (\mathit{age}(\mathrm{o2f}_{\mathrm{eval}}(\mathsf{p})) = \mathit{age}(p))$,

where p is a variable of sort Classifier. □

Definition 6 *Let expr, expr*$_1$*, expr*$_2$*, and src be expression of the appropriate type, we define the mappings* $\mathrm{o2f}_{\mathrm{def\_c}}$*, as follow:*

allInstances-expressions:

$\mathrm{o2f}_{\mathrm{def\_c}}(c.\mathsf{allInstances()}) = \{\forall(x)(\mathrm{App}([c], \emptyset, x) \Leftrightarrow (\bigvee_{s \preceq c}(s(x))))\}.$

association-end-expressions (multiplicity *):

$\mathrm{o2f}_{\mathrm{dfn\_c}}(\mathit{expr}.\mathit{as}()) = \{\forall(Y)\forall(x)(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}.\mathit{as}()), Y, x) \Leftrightarrow \ \mathit{as}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}), x))\}.$

where $Y = \mathrm{fVars}(\mathit{expr})$ *and* $x \notin Y$.

select-expressions:

$$\begin{aligned}&\mathrm{o2f}_{\mathrm{dfn\_c}}(\mathit{src}\text{->}\mathsf{select}(x \mid \mathit{body})) = \\ &\quad \{\forall(Y)\forall(x)(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}\text{->}\mathsf{select}(x \mid \mathit{body})), Y, x) \\ &\qquad \Leftrightarrow (\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}), \mathrm{fVars}(\mathit{src}), x) \wedge \mathrm{o2f}_{\mathrm{true}}(\mathit{body})))\}.\end{aligned}$$

where $Y = \mathrm{fVars}(\mathit{src}\text{->}\mathsf{select}(x \mid \mathit{body}))$.

reject-expressions:

$$\begin{aligned}&\mathrm{o2f}_{\mathrm{dfn\_c}}(\mathit{src}\text{->}\mathsf{reject}(x \mid \mathit{body})) = \\ &\quad \{\forall(Y)\forall(x)(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}\text{->}\mathsf{reject}(x \mid \mathit{body})), Y, x) \\ &\qquad \Leftrightarrow (\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}), \mathrm{fVars}(\mathit{src}), x) \wedge \mathrm{o2f}_{\mathrm{false}}(\mathit{body})))\}.\end{aligned}$$

where $Y = \mathrm{fVars}(\mathit{src}\text{->}\mathsf{reject}(x \mid \mathit{body}))$.

including-expressions:

$$\begin{aligned}&\mathrm{o2f}_{\mathrm{dfn\_c}}(\mathit{src}\text{->}\mathsf{including}(\mathit{expr})) = \\ &\quad \{\forall(Y)\forall(x)(\mathrm{App}([\mathit{src}\text{->}\mathsf{including}(\mathit{expr})], Y, x) \\ &\qquad \Leftrightarrow (\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}), \mathrm{fVars}(\mathit{src}), x) \vee \mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}) = x))\}.\end{aligned}$$

where $Y = \mathrm{fVars}(\mathit{src}\text{->}\mathsf{including}(\mathit{expr}))$.

excluding-expressions:

$$\begin{aligned}&\mathrm{o2f}_{\mathrm{dfn\_c}}(\mathit{src}\text{->}\mathsf{excluding}(\mathit{expr})) = \\ &\quad \{\forall(Y)\forall(x)(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}\text{->}\mathsf{excluding}(\mathit{expr})), Y, x) \\ &\qquad \Leftrightarrow (\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}), \mathrm{fVars}(\mathit{src}), x) \wedge \mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}) \neq x))\}.\end{aligned}$$

where $Y = \mathrm{fVars}(\mathit{src}\text{->}\mathsf{excluding}(\mathit{expr}))$.

union-expressions:

$$\begin{aligned}&\mathrm{o2f}_{\mathrm{dfn\_c}}(\mathit{expr}_1\text{->}\mathsf{union}(\mathit{expr}_2)) = \\ &\quad \{\forall(Y)\forall(x)(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_1\text{->}\mathsf{union}(\mathit{expr}_2)), Y, x) \\ &\qquad \Leftrightarrow (\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_1), \mathrm{fVars}(\mathit{expr}_1), x) \vee \ \mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_2), \mathrm{fVars}(\mathit{expr}_2), x)))\}.\end{aligned}$$

where $Y = \mathrm{fVars}(\mathit{expr}_1\text{->}\mathsf{union}(\mathit{expr}_2))$.

intersection-expressions:

$$\begin{aligned}&\mathrm{o2f}_{\mathrm{dfn\_c}}(\mathit{expr}_1\text{->}\mathsf{intersection}(\mathit{expr}_2)) = \\ &\quad \{\forall(Y)\forall(x)(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_1\text{->}\mathsf{intersection}(\mathit{expr}_2)), Y, x) \\ &\qquad \Leftrightarrow (\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_1), \mathrm{fVars}(\mathit{expr}_1), x) \wedge \ \mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_2), \mathrm{fVars}(\mathit{expr}_2), x)))\}.\end{aligned}$$

where $Y = \mathrm{fVars}(\mathit{expr}_1\text{->}\mathsf{intersection}(\mathit{expr}_2))$.

set-difference-expressions:

$$\begin{aligned}&\mathrm{o2f}_{\mathrm{dfn\_c}}(\mathit{expr}_1\text{->-}(\mathit{expr}_2)) =\\ &\quad\{\forall(Y)\forall(x)(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_1\text{->-}(\mathit{expr}_2)), Y, x)\\ &\qquad\Leftrightarrow (\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_1), \mathrm{fVars}(\mathit{expr}_1), x) \wedge\ \neg(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_2), \mathrm{fVars}(\mathit{expr}_2), x))))\}.\end{aligned}$$

where $Y = \mathrm{fVars}(\mathit{expr}_1\text{->-}(\mathit{expr}_2))$.

symmetricDifference-expressions:

$$\begin{aligned}&\mathrm{o2f}_{\mathrm{dfn\_c}}(\mathit{expr}_1\text{->}\mathsf{symmetricDifference}(\mathit{expr}_2)) =\\ &\quad\{\forall(Y)\forall(x)(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_1\text{->}\mathsf{symmetricDifference}(\mathit{expr}_2)), Y, x)\\ &\qquad\Leftrightarrow (\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_1), \mathrm{fVars}(\mathit{expr}_1), x) \wedge\ \neg(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_2), \mathrm{fVars}(\mathit{expr}_2), x))\\ &\qquad\quad\vee\ \mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_2), \mathrm{fVars}(\mathit{expr}_2), x) \wedge \neg(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{expr}_1), \mathrm{fVars}(\mathit{expr}_1), x))))\}.\end{aligned}$$

where $Y = \mathrm{fVars}(\mathit{expr}_1\text{->}\mathsf{symmetricDifference}(\mathit{expr}_2))$.

collect-expressions (with body of type set):

$$\begin{aligned}&\mathrm{o2f}_{\mathrm{dfn\_c}}(\mathit{src}\text{->}\mathsf{collect}(x|\mathit{body})\text{->}\mathsf{asSet}(), \vec{v}) =\\ &\quad\{\forall(Y)\forall(x)(\mathrm{App}([\mathit{src}\text{->}\mathsf{collect}(x|\mathit{body})\text{->}\mathsf{asSet}()], Y, x)\\ &\qquad\Leftrightarrow \exists(z)(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}), \mathrm{fVars}(\mathit{src}), z) \wedge \mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{body}[\mathsf{x} \mapsto z]), \mathrm{fVars}(\mathit{body}), x)))\}.\end{aligned}$$

where $Y = \mathrm{fVars}(\mathit{src}\text{->}\mathsf{collect}(x|\mathit{body})\text{->}\mathsf{asSet}())$ *and* $z \notin Y$.

collect-expressions (with body of class or primitive type):

$$\begin{aligned}&\mathrm{o2f}_{\mathrm{dfn\_c}}(\mathit{src}\text{->}\mathsf{collect}(x|\mathit{body})\text{->}\mathsf{asSet}()) =\\ &\quad\{\forall(Y)\forall(x)(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}\text{->}\mathsf{collect}(x|\mathit{body})\text{->}\mathsf{asSet}()), Y, x)\\ &\qquad\Leftrightarrow \exists(z)(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}), \mathrm{fVars}(\mathit{src}), z) \wedge \mathrm{o2f}_{\mathrm{eval}}(\mathit{body}[\mathsf{x} \mapsto z]) = x))\}.\end{aligned}$$

where $Y = \mathrm{fVars}(\mathit{src}\text{->}\mathsf{collect}(x|\mathit{body})\text{->}\mathsf{asSet}())$ *and* $z \notin Y$.

Example 17 *Consider the non-Boolean expression:*

Post.allInstances().

Then, $\mathrm{o2f}_{\mathrm{eval}}(\mathsf{Post.allInstances()}) = [\mathsf{Post.allInstances()}]$*, where the new predicate* $[\mathsf{Post.allInstances()}]$ *is defined by* $\mathrm{o2f}_{\mathrm{def\_c}}$ *as follows:*

$\forall(x{:}\mathit{Classifier})([\mathsf{Post.allInstances()}] \Leftrightarrow (\mathit{Post}(x) \vee \mathit{Photo}(x) \vee \mathit{Status}(x)))$.

□

Example 18 *Consider the non-Boolean expression:*

Profile.allInstances()->select(p|p.age.oclIsUndefined).

Then,

$$\begin{aligned}&\mathrm{o2f}_{\mathrm{eval}}(\mathsf{Profile.allInstances()\text{->}select(p|p.age.oclIsUndefined())}) =\\ &\quad[\mathsf{Profile.allInstances()\text{->}select(p|p.age.oclIsUndefined())}],\end{aligned}$$

where the new predicate

[Profile.allInstances()->select(p|p.age.oclIsUndefined())]

is defined by $\mathrm{o2f}_{\mathrm{def\_c}}$ *as follows:*

$$\begin{aligned}
&\forall(p{:}\,\mathit{Classifier})([\textsf{Profile.allInstances()->select(p|p.age.oclIsUndefined())}](p)\\
&\quad\Leftrightarrow (\mathrm{o2f}_{\mathrm{eval}}(\textsf{Profile.allInstances()})(p) \wedge\ \mathrm{o2f}_{\mathrm{true}}(\textsf{p.age.oclIsUndefined()})))\\
&= \forall(p{:}\,\mathit{Classifier})([\textsf{Profile.allInstances()->select(p|p.age.oclIsUndefined())}](p)\\
&\qquad\Leftrightarrow [\textsf{Profile.allInstances()}](p) \wedge (\mathrm{o2f}_{\mathrm{eval}}(\textsf{p.age}) = \mathit{nullClassifier}\\
&\qquad\quad \vee(p = \mathit{nullClassifier} \vee p = \mathit{invalClassifier}))\\
&= \forall(p{:}\,\mathit{Classifier})([\textsf{Profile.allInstances()->select(p|p.age.oclIsUndefined())}](p)\\
&\qquad\Leftrightarrow [\textsf{Profile.allInstances()}](p) \wedge (\mathit{age}(p) = \mathit{nullClassifier}\\
&\qquad\quad \vee(p = \mathit{nullClassifier} \vee p = \mathit{invalClassifier})).
\end{aligned}$$

and where the new predicate $[\textsf{Profile.allInstances()}]$ *is defined by* $\mathrm{o2f}_{\mathrm{def\_c}}$ *as follows:*

$$\forall(x{:}\,\mathit{Classifier})([\textsf{Profile.allInstances()}] \Leftrightarrow \mathit{Profile}(x)).$$

□

Definition 7 *Let expr,* expr_1*,* expr_2*, and src be expression of the appropriate type, we define the mappings* $\mathrm{o2f}_{\mathrm{def\_o}}$*, as follow:*

max-expressions:

$$\begin{aligned}
&\mathrm{o2f}_{\mathrm{dfn\_o}}(\mathit{src}\textsf{->max()}) =\\
&\quad\{\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}\textsf{->max()}), \mathrm{fVars}(\mathit{src})) = \mathit{invalInt}) \Leftrightarrow \mathrm{o2f}_{\mathrm{inval}}(\mathit{src}),\\
&\quad\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}\textsf{->max()}), \mathrm{fVars}(\mathit{src})) = \mathit{nullInt}\\
&\qquad\Leftrightarrow\ \ (\neg(\mathrm{o2f}_{\mathrm{inval}}(\mathit{src})) \wedge \forall(x)(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}), \mathrm{fVars}(\mathit{src}), x) \Rightarrow x = \mathit{nullInt})),\\
&\quad((\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}\textsf{->max()}), \mathrm{fVars}(\mathit{src})) \neq \mathit{nullInt}\\
&\qquad\quad \wedge\ \mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}\textsf{->max()}), \mathrm{fVars}(\mathit{src})) \neq \mathit{invalInt}))\\
&\qquad\Leftrightarrow\ \ (\neg(\mathrm{o2f}_{\mathrm{inval}}(\mathit{src}))\\
&\qquad\qquad \wedge\ \mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}), \mathrm{fVars}(\mathit{src}), \mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}\textsf{->max()}), \mathrm{fVars}(\mathit{src})))\\
&\qquad\qquad \wedge\ \forall(y)(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}), \mathrm{fVars}(\mathit{src}), y)\\
&\qquad\qquad\qquad \Rightarrow \mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}\textsf{->max()}), \mathrm{fVars}(\mathit{src})) \geq y)))\}.
\end{aligned}$$

min-expressions:

$$\begin{aligned}
&\mathrm{o2f}_{\mathrm{dfn\_o}}(\mathit{src}\textsf{->min()}) =\\
&\quad\{\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}\textsf{->min()}), \mathrm{fVars}(\mathit{src})) = \mathit{invalInt}) \Leftrightarrow \mathrm{o2f}_{\mathrm{inval}}(\mathit{src}),\\
&\quad\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}\textsf{->min()}), \mathrm{fVars}(\mathit{src})) = \mathit{nullInt}\\
&\qquad\Leftrightarrow\ \ (\neg(\mathrm{o2f}_{\mathrm{inval}}(\mathit{src}))\\
&\qquad\qquad \wedge\ \forall(x)(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}), \mathrm{fVars}(\mathit{src}), x) \Rightarrow x = \mathit{nullInt})),\\
&\quad((\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}\textsf{->min()}), \mathrm{fVars}(\mathit{src})) \neq \mathit{nullInt}\\
&\qquad\quad \wedge\ \mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}\textsf{->min()}), \mathrm{fVars}(\mathit{src})) \neq \mathit{invalInt}))\\
&\qquad\Leftrightarrow\ \ (\neg(\mathrm{o2f}_{\mathrm{inval}}(\mathit{src}))\\
&\qquad\qquad \wedge\ \mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}), \mathrm{fVars}(\mathit{src}), \mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}\textsf{->min()}), \mathrm{fVars}(\mathit{src})))\\
&\qquad\qquad \wedge\ \forall(y)(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}), \mathrm{fVars}(\mathit{src}), y)\\
&\qquad\qquad\qquad \Rightarrow \mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}\textsf{->min()}), \mathrm{fVars}(\mathit{src})) \leq y)))\}.
\end{aligned}$$

any-expressions:

$$\begin{aligned}
&\mathrm{o2f}_{\mathrm{dfn\_o}}(\mathit{src}\textsf{->any}(x_t|\mathit{body})) =\\
&\quad\{\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}\textsf{->any}(x|\mathit{body})), Y) = \mathit{invalt} \Leftrightarrow \mathrm{o2f}_{\mathrm{inval}}(\mathit{src}),\\
&\quad\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}\textsf{->any}(x|\mathit{body})), \mathrm{fVars}(\mathit{src})) = \mathrm{nullOf}(t) \Leftrightarrow\ \ (\neg(\mathrm{o2f}_{\mathrm{inval}}(\mathit{src}))\\
&\qquad \wedge\ \forall(x)(\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}), \mathrm{fVars}(\mathit{src}), x) \Rightarrow \neg(\mathrm{o2f}_{\mathrm{true}}(\mathit{body})))),
\end{aligned}$$

$$\begin{aligned}&((\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}\text{->}\mathsf{any}(x|\mathit{body})),Y) \neq \mathit{nullt}\\ &\quad\wedge\ \mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}\text{->}\mathsf{any}(x|\mathit{body})),Y) \neq \mathit{invalt})\\ &\quad\Leftrightarrow\ (\neg(\mathrm{o2f}_{\mathrm{inval}}(\mathit{src}))\\ &\qquad\wedge\ \mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}),\mathrm{fVars}(\mathit{src}),\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}\text{->}\mathsf{any}(x|\mathit{body})),Y))\\ &\qquad\wedge\ \mathrm{o2f}_{\mathrm{true}}(\mathit{body}[x\mapsto \mathsf{x}]\mathrm{App}(\mathrm{o2f}_{\mathrm{eval}}(\mathit{src}\text{->}\mathsf{any}(x|\mathit{body})),Y)))))\}.\end{aligned}$$

where $Y = \mathrm{fVars}(\mathit{src}\text{->}\mathsf{any}(x|\mathit{body}))$*, and invalt is the invalid value for the type t.*

In what follows we denote by $\mathrm{o2f}_{\mathrm{def}}(\mathit{expr})$ the set of axioms that result from applying $\mathrm{o2f}_{\mathrm{def\_c}}$ and $\mathrm{o2f}_{\mathrm{def\_o}}$ to the corresponding non-Boolean subexpression in *expr*. Notice that, in particular, for each literal integer i and literal string st in *expr*, $\mathrm{o2f}_{\mathrm{def\_o}}$ generates the following axioms:

$\mathrm{o2f}_{\mathrm{dfn\_o}}(i) = \{\neg(i = \mathit{nullInt}) \wedge \neg(i = \mathit{invalInt})\}$.
$\mathrm{o2f}_{\mathrm{dfn\_o}}(st) = \{\neg(i = \mathit{nullString}) \wedge \neg(i = \mathit{invalString})\}$

3.1.3 Checking satisfiability

The mapping from OCL into MSFOL generates as output a class of satisfiability problems that can be efficiently handled by SMT solvers with finite model finding capabilities.

The following remark formalizes the main property of our mapping from OCL to many-sorted first-order logic.

Remark 1 *Let $\mathcal{D}$ be a data model, let and $\mathcal{I}$ be a set of $\mathcal{D}$-constraints, and let expr be a Boolean OCL expression. Then, expr evaluates to* `true` *in every valid instance of $\mathcal{D}$ if and only if*

$$\begin{aligned}\mathrm{o2f}_{\mathrm{data}}(\mathcal{D}) \cup \left(\bigcup_{\mathit{inv}\in\mathcal{I}} \mathrm{o2f}_{\mathrm{def}}(\mathit{inv})\right) \cup \left(\bigcup_{\mathit{inv}\in\mathcal{I}} \{\mathrm{o2f}_{\mathrm{true}}(\mathit{inv})\}\right)\\ \cup\ \mathrm{o2f}_{\mathrm{def}}(\mathit{expr}) \cup \{\mathrm{o2f}_{\mathrm{false}}(\mathit{expr})\}.\end{aligned}$$

is unsatisfiable.

3.1.4 The OCL2MSFOL tool

OCL2MSFOL [70] is a Java Web Application that implements the mapping. More specifically, it takes as input a data model $\mathcal{D}$, a set of $\mathcal{D}$-constraints $\mathcal{I}$. Also, the user can introduce a $\mathcal{D}$-expression *expr* and request to OCL2MSFOL to map *expr* to MSFOL using either $\mathrm{o2f}_{\mathrm{true}}$, $\mathrm{o2f}_{\mathrm{false}}$, $\mathrm{o2f}_{\mathrm{null}}$, or $\mathrm{o2f}_{\mathrm{inval}}$. The result will be a file containing the MSFOL theory, described before.

Figure 3.2 shows the main components of the tool architecture. These are:

- DM validator: This component checks whether the input data model fulfills the restrictions about well-formedness that we explain in Section 2.2.1, so as to serve as a valid context for OCL queries.

- OCL validator: This component parses each OCL query of input in the context of the data model. Only if a constraint parses correctly (and our mapping covers it), it is used as input to produce code.

- SMT engine selector: This component receives as input the vendor identifier so as the code generated is syntactically adapted to the selected SMT solver.

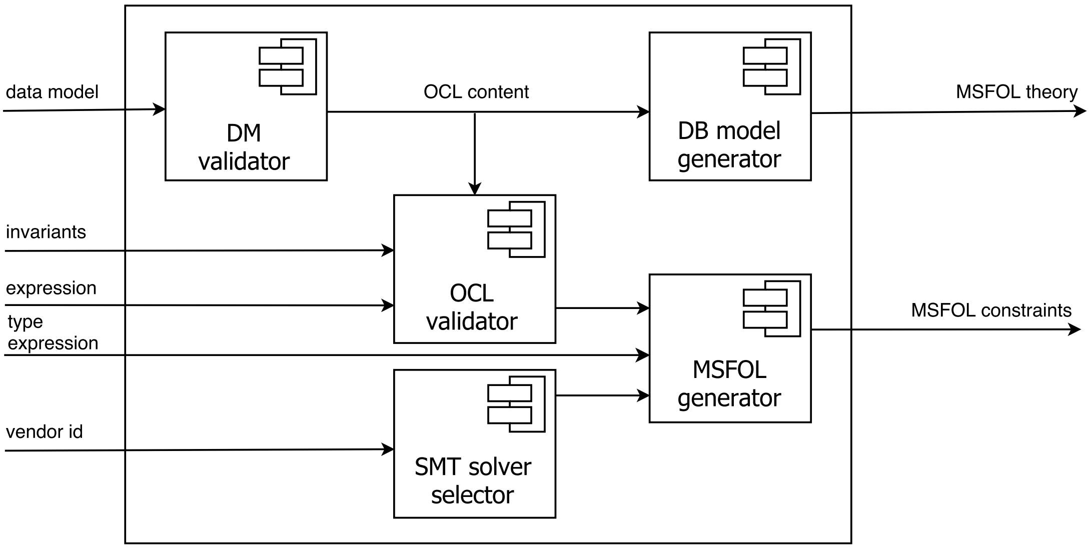


Figure 3.2: OCL2MSFOL tool component architecture

- MSFOL model generator: This component generates the *engine-specific* statements to create the MSFOL theory.
- MSFOL generator: This component generates the *SMT solver-specific* statements to create the MSFOL contraints (hypothesis and assertion) corresponding to the input OCL.

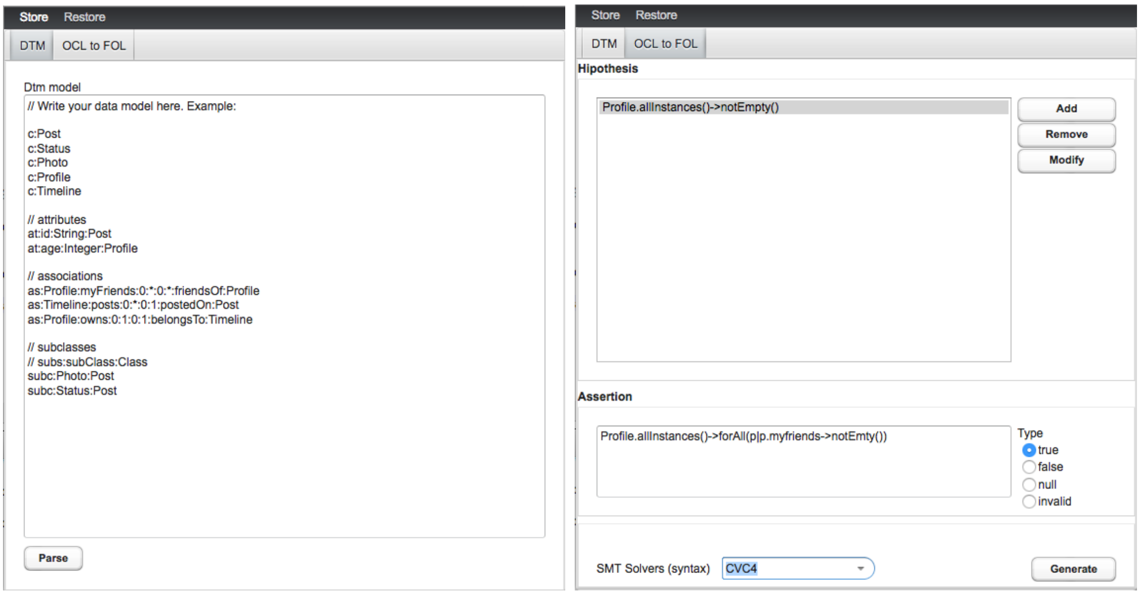
The typical use case for OCLMSFOL is as follows. Suppose a data-model $\mathcal{D}$, with invariants $\mathcal{I}$, and a Boolean $\mathcal{D}$-expression *expr*. Then, to check whether there exists a valid instance of $\mathcal{D}$ in which *expr* evaluates to true, we do the following: i) we input in OCL2MSFOL the data model $\mathcal{D}$, the set of invariants $\mathcal{I}$, and the expression *expr*; ii) we select the option true; and iii) we input the file generated by OCL2MSFOL into our SMT solver of choice.

Figure 3.3 shows two screen-shots of the tool interface. If the SMT solver returns sat, then we know that such an instance of $\mathcal{D}$ exists; if the SMT solver returns unsat then we know that no such an instance of $\mathcal{D}$ exists; and, finally, if the SMT solver returns unknown, then we know that it remains unknown whether such an instance of $\mathcal{D}$ exists. The process is entirely similar if we want to know whether there exists a valid instance of $\mathcal{D}$ in which an expression *expr* evaluates to false, null, or inval; the only difference is that, instead of true, we will select, respectively, false, null, or inval.

We introduce below a benchmark for checking the satisfiability of OCL constraints, and report on our results. All checks are ran on a laptop computer, with an Intel Core i7 processor running at 3.1GHz with 8Gb of RAM. As back-end theorem-provers, we use Z3 [30] (version 4.4.1), and CVC4 [10] (version 1.5-prerelease). In the case of Z3, we use its default setting, but in the case of CVC4, we use two different settings, namely, with and without the option finite-model-find. In what follows, we refer to the latter as CVC4 Finite Model (or CVC4fm, for short).

The data model for our benchmark is the basic social network model shown in Figure 3.1. The Boolean expressions that we consider in our benchmark are the following:

1. Profile.allInstances()->forAll(p|p.age>18)



Dtm model

```
// Write your data model here. Example:

c:Post
c:Status
c:Photo
c:Profile
c:Timeline

// attributes
at:id:String:Post
at:age:Integer:Profile

// associations
as:Profile:myFriends:0:*:0:*:friendsOf:Profile
as:Timeline:posts:0:*:0:1:postedOn:Post
as:Profile:owns:0:1:0:1:belongsTo:Timeline

// subclasses
// subs:subClass:Class
subc:Photo:Post
subc:Status:Post
```

Hipothesis

```
Profile.allInstances()->notEmpty()
```

Assertion

```
Profile.allInstances()->forAll(p|p.myfriends->notEmty())
```

Type
● true
○ false
○ null
○ invalid

SMT Solvers (syntax): CVC4

Figure 3.3: OCL2MSFOL tool: screenshots

2. Profile.allInstances()–>exists(p|p.age<=18)
3. Profile.allInstances()–>exists(p|p.age.oclIsUndefined())
4. Profile.allInstances()–>exists(p|p.oclIsUndefined())
5. Profile.allInstances()–>forAll(p|p.oclIsUndefined())
6. Profile.allInstances()–>notEmpty()
7. Profile.allInstances()–>collect(p|p.age)–>asSet()–>exists(a|a.oclIsUndefined())
8. Profile.allInstances()–>any(p|p.age>16).oclIsUndefined()
9. Profile.allInstances()–>any(p|p.age>16).age.oclIsInvalid()
10. not(Profile.allInstances()–>any(p|p.age<16).age.oclIsInvalid())
11. Status.allInstances()–>notEmpty()
12. Post.allInstances()–>forAll(p|Photo.allInstances()–>exists(q|p.id=q.id))
13. Post.allInstances()–>forAll(p|not(p.id.oclIsUndefined()))
14. Status.allInstances()–>notEmpty()
15. Status.allInstances()–>isEmpty()
16. Photo.allInstances()–>notEmpty()
17. Photo.allInstances()–>isEmpty()
18. Post.allInstances()–>notEmpty()
19. Post.allInstances()–>isEmpty()
20. Post.allInstances()–>forAll(p|Photo.allInstances()–>exists(q|p.id=q.id))
21. Photo.allInstances()–>forAll(p|p.oclIsKindOf(Post))
22. Photo.allInstances()–>forAll(p|p.oclIsKindOf(Timeline))
23. Post.allInstances()–>forAll(p|p.oclIsTypeOf(Timeline))
24. Post.allInstances()–>forAll(p|not(p.oclIsTypeOf(Post)))
25. 2.oclIsUndefined()
26. Post.allInstances()–>forAll(p| Post.allInstances()–>forAll(q|p<>q implies p.id<>q.id)
27. Profile.allInstances()–>forAll(p|p.myFriends–>notEmpty())

In Table 3.1 we show the result of checking, using OCL2MSFOL, the satisfiability of different subsets of our benchmark's Boolean expressions.

		CVC4	Z3	CVC4fm
{1,2}	unsat	161	24	48
{1,3}	unsat	173	13	22
{2,3}	sat	—	16	25
{4}	unsat	138	15	27
{5}	sat	—	17	22
{5,6}	unsat	172	13	30
{1,7}	unsat	237	14	30
{1,8}	sat	—	18	25
{1,6,8}	unsat	198	16	26
{1,9}	sat	—	18	25
{1,6,9}	unsat	200	19	29
{1,10}	unsat	203	18	30
{12}	sat	—	169	27
{11,12,13}	sat	—	24	174

(a) Undefinedness-related (times in ms)

		CVC4	Z3	CVC4fm
{14,20}	sat	—	105	28
{16,20}	sat	—	466	32
{17,20}	sat	—	14	22
{14,17,20}	unsat	239	13	26
{16,19}	unsat	168	16	28
{21}	sat	—	17	27
{22}	sat	—	199	24
{16,22}	unsat	149	18	25
{16,23}	unsat	148	16	26
{15,17,18,24}	unsat	250	15	35
{25}	unsat	63	58	24
{11,12,13,18}	sat	—	—	27
{6,27}	sat	—	—	26
{11,12,13,18,26}	unsat	352	13	25

(b) Generalization-related (times in ms)

Table 3.1: Checking satisfiability of OCL constraints.

We have grouped all our checks in two tables: (3.1a) contains the checks related to undefinedness, while (3.1b) contains the checks related to UML generalization. In both cases, the first column indicates the set of Boolean expressions to be checked for satisfiability; the second column indicates the expected result, according to our understanding of the semantics of OCL; and the third, fourth, and fifth column indicate, respectively, the time (in milliseconds) taken by CVC4, Z3, and CVC4 Finite Model to return the expected result, or '—', in the case they return unknown. Notice that CVC4 Finite Model is able to return the expected result in all cases, while Z3 and CVC4 return *unknown* in some

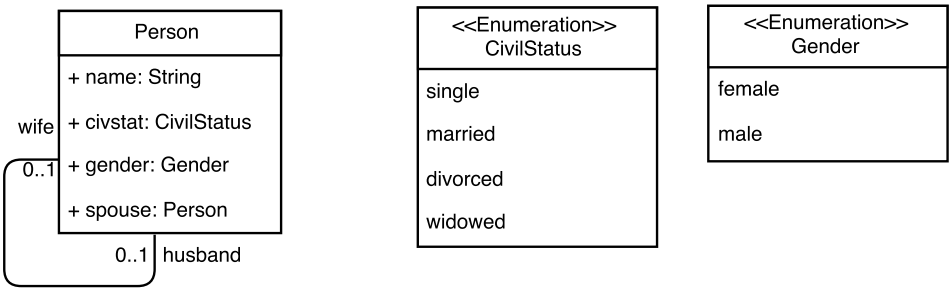


Figure 3.4: CivilStatus: A civil status model

cases.

3.2 Benchmark

With the intention of helping developers to choose the UML/OCL tool more appropriate for their projects, [38] has proposed a benchmark for assessing validation and verification techniques on UML/OCL models. It includes four models each posing different computational challenges. We use this benchmark to assess OCL2MSFOL and to compare it with other tools for verifying UML/OCL models.

3.2.1 Description

The benchmark proposed in [38] includes four UML/OCL models, namely, CivilStatus, WritesReviews, DisjointSubclasses, and ObjectsAsIntegers, together with a set of questions for each of these models. It is sufficient for our purpose to consider only the first three models: CivilStatus, WritesReviews, and DisjointSubclasses.[5]

CivilStatus

Figure 3.4 shows the first UML class diagram considered in the benchmark. Basically, it models that a person has a name, a gender (either female or male), a civil status (either single, married, divorced, or widowed), and possibly a spouse, and that a person has a husband or a wife or none. The following OCL invariants further constrain this model.[6]

- **attributesDefined:** *A person has a defined name, civil status and gender.*

```
Person.allInstances()–>forAll(p|not(p.name.oclIsUndefined()
  and not(p.civStat.oclIsUndefined())
  and not(p.gender.oclIsUndefined())).
```

- **nameIsUnique:** *A person has a unique name.*

```
Person.allInstances()–>forAll(p1|Person.allInstances()
  –>forAll(p2|p1 <> p2 implies p1.name <> p2.name)).
```

[5] The fourth model, ObjectAsIntegers, is definitely more “artificial”; furthermore, it requires inductive reasoning, which is out of the scope of both our analysis tool and the tools we are comparing to in this benchmarking exercise.

[6] The benchmark includes an additional constraint about the format of a person’s name, which for our present purpose we omit here since it plays no significant role in answering the questions later posed about the model.

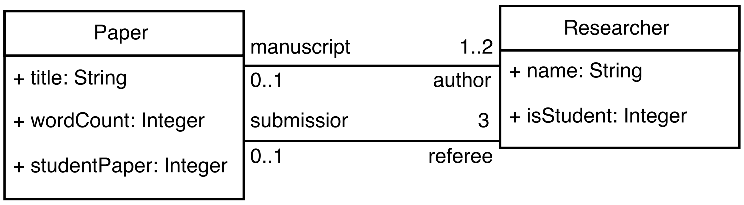


Figure 3.5: WritesReviews: A writes reviews model

- **femaleHasNoWife:** *A female person does not possess a wife.*

  Person.allInstances()–>forAll(p|p.gender = Gender::female
  implies p.wife.oclIsUndefined()).

- **maleHasNoHusband:** *A male person does not possess a husband.*

  Person.allInstances()–>forAll(p|p.gender = Gender::male
  implies p.husband.oclIsUndefined()).

- **hasSpouse_EQ_civstatMarried:** *A person has a spouse, if and only if his/her civil status is married.*

  Person.allInstances()–>forAll(p|
  (not(p.spouse.oclIsUndefined()) implies p.civStat = CivilStatus::married)
  and (p.civStat = CivilStatus::married implies not(p.spouse.oclIsUndefined()))).

In the benchmark the following questions are posed about this model:

1. **Consisten tInvariants:** Is the model *consistent*? That is, is there at least one instance of the model satisfying all the stated invariants?
2. **Independence:** Are all the invariants *independent*? Or, on the contrary, is there at least one invariant which is a consequence of the conditions imposed by the model and the other invariants?
3. **Consequences:** Is the model bigamy-free? Or, on the contrary, is it possible for a person to have both a wife and a husband?

WritesReviews

Figure 3.5 shows the second UML class diagram considered in the benchmark. Basically, it models a simple conference review system. There are papers and researchers. A paper has a title and a number of words, and can be a studentPaper. A researcher has a name and can be a student. A researcher can be assigned at most one submission to review it and can submit at most one manuscript. A paper can have one or two authors and must be assigned exactly three referees. The following invariants further constrain this model:

- **oneManuscript:** *A researcher must submit one manuscript.*

  Researcher.allInstances()–>forAll(r| not(r.manuscript.oclIsUndefined())).

- **oneSubmission:** *A research must be assigned one submission.*

  ```
  Researcher.allInstances()->forAll(r|not(r.submission.oclIsUndefined())).
  ```

- **noSelfReviews:** *A paper cannot be refereed by one of its authors.*

  ```
  Researcher.allInstances()->forAll(r| not(r.submission.oclIsUndefined())
    r.submission.author->forAll(a|a<>r))).
  ```

- **paperLength:** *A paper must have at most 10000 words.*

  ```
  Paper.allInstances()->forAll(p|p.wordCount < 10000).
  ```

- **authorsOfStudentPaper:** *One of the authors of a student paper must be a student.*

  ```
  Paper.allInstances()->forAll(p|(p.studentPaper = 1)
    p.author->exists(x|x.isStudent = 1))).
  Paper.allInstances()->forAll(p| (p.author->exists(x|x.isStudent = 1))
    p.studentPaper = 1)).
  ```

- **noStudentReviewers:** *Students are not allowed to review any paper.*

  ```
  Paper.allInstances()->forAll(p|p.referee->forAll(r|r.isStudent <> 1)).
  ```

- **limitsOnStudentPapers:** *There must be at least one student paper.*[7]

  ```
  Paper.allInstances()->exists(p|p.studentPaper = 1).
  ```

In the benchmark the following questions are posed about this model:

1. **InstantiateNonemptyClass:** Can the model be instantiated with non-empty populations for all classes? That is, is there at least one instance of this model with at least one paper and one researcher?

2. **InstantiateNonemptyAssoc:** Can the model be instantiated with non-empty populations for all classes and all associations? That is, is there at least one instance of this model with at least one paper, one researcher, one instance of the manuscript-author association, and one instance of the submission-referee association?

3. **InstantiateInvariantIgnore:** Can the model be instantiated if the invariants **oneManuscript** and **oneSubmission** are ignored?

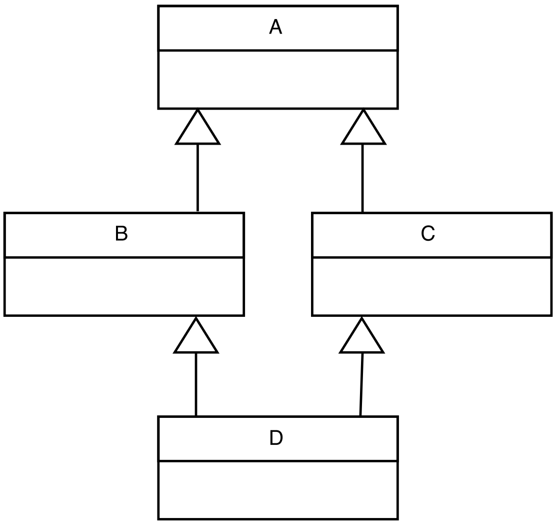


Figure 3.6: DisjointSubclasses: A disjoint subclasses datamodel

DisjointSubclasses

Figure 3.6 shows the third UML class diagram considered in the benchmark. There are four classes: A, B, C, and D. Class B and C inherit from class A, while class D inherits from both class B and class C. The following invariant further constrains this model:

- **disjointBC:** *Class B and class C are disjoint.*

  C.allInstances()–>forAll(x|B.allInstances()–>forAll(y|x<>y))

In the benchmark the following questions are posed about this model:

1. **InstantiateDisjointInheritance:** Can all classes be populated? That is, is there at least one instance of this model with at least one element of each class? In particular, is there at least one instance of this model with at least one element of class D?

2. **InstantiateMultipleInheritance:** Can class D be populated if the constraint **disjointBC** is ignored?

3.2.2 Results

Comparison with USE and EMFtoCSP

USE [39] and EMFtoCSP [41] are two different tools for automatically verifying and validating UML/OCL models. While USE checks UML/OCL consistency using enumeration and SAT-based techniques, EMFtoCSP turns a UML/OCL consistency problem into a constraint satisfaction problem (CSP) and uses constraint solvers to solve it. In both cases, the user is required to specify *ranges* for the class and association extents and for the attribute domains. The fact that both USE and OCL2MSFOL operate on *bounded* search state spaces implies, on the one hand, that, when there is a valid instance of a

[7] The benchmark requires also that the number of student papers should be less than 5. For the sake of simplicity, we only consider here the first part of the constraint since the second one plays no significant role in answering the questions later posed about the model.

Question	*Answer*	*Time (in secs)*	*Remarks*
ConsistentInvariants	sat	0.08	
Independence	sat	0.29	For invariant 1
		0.40	For invariant 2
		0.32	For invariant 3
		0.34	For invariant 4
		0.16	For invariant 5
Consequences	unsat	0.24	

Table 3.2: Analyzing CivilStatus with OCL2MSFOL

given UML/OCL model *within* the selected range, both USE and EMFtoCSP will find it —assuming that the selected range is sufficiently small, of course. But, on the other hand, it also means that, when either USE or EMFtoCSP communicates to the user that no valid instance of a model has been found, this answer is *inconclusive*, since a valid instance may still exist outside of the the selected range.

In what follows we use the benchmark to assess OCL2MSFOL. The results of USE and EMFtoCSP on this same benchmark were reported in [38]. For the sake of comparison with OCL2MSFOL, the results obtained by USE and EMFtoCSP are entirely analogous, and we will draw explicit comparisons only with the former.

All OCL2MSFOL checks were run on a laptop computer, with an Intel Core i7 processor running at 1.8GHz with 4Gb of RAM. As back-end SMT solver, we use CVC4 (version 1.5-prerelease) with the option `finite-model-find`. We denote this configuration as CVC4$^{\mathrm{fmf}}$.

CivilStatus

In Table 3.2 we show the results of analysing, using OCL2MSFOL, the questions posed in the benchmark about CivilStatus. In particular,

- **ConsistentInvariants:**. CVC4$^{\mathrm{fmf}}$ finds a valid instance of CivilStatus and returns **sat**. Thus, we can conclude that the model is consistent.

- **Independence:** For each of the five invariants, CVC4$^{\mathrm{fmf}}$ finds a valid instance of a modified version of CivilStatus, where the given invariant is negated while the other are still affirmed, returning **sat** in each case. Thus, we can conclude that the invariants are independent.

- **Consequences:** CVC4$^{\mathrm{fmf}}$ returns **unsat** when the following invariant is added to CivilStatus:

  ```
  Person.allInstances()->exists(p|
    not(p.husband.oclIsUndefined()) and not(p.wife.oclIsUndefined())).
  ```

  Thus, we can conclude that the model is bigamy-free.

We can now compare these results with the ones obtained by analyzing CivilStatus using USE, as reported in [38]. In particular,

- **ConsistentInvariants:** Selecting as search state space the instances of CivilStatus with exactly one male person and one female person, USE finds a valid instance of CivilStatus. Thus, as with OCL2MSFOL, we can conclude that the model is consistent.

- **Independence:** Selecting as search state space the instances of CivilStatus with exactly one male person and one female person, for each of the five invariants, USE finds an instance of CivilStatus such that this invariant is not satisfied while the others are satisfied. Thus, as with OCL2MSFOL, we can conclude that the invariants are independent.

- **Consequences:** Selecting as search state space the instances of CivilStatus with at most three persons, USE is not able to find an instance of CivilStatus which is bigamy-free. Notice that this answer is *inconclusive*, since a bigamy instance of CivilStatus may still exist outside of the selected range. On the contrary, the answer provided by OCL2MSFOL guarantees that CivilStatus is bigamy-free.

WritesReviews

In Table 3.3 we show the results of analysing, using OCL2MSFOL, the questions posed in the benchmark about WritesReviews. In particular,

- **InstantiateNonemptyClass:** CVC4$^{\text{fmf}}$ returns **unsat** when the following invariants are added to WritesReviews:

  ```
  Paper.allInstances()–>notEmpty().
  Researcher.allInstances()–>notEmpty().
  ```

  Thus, we can conclude that there is no valid instance of WritesReviews with at least one paper and one researcher.

- **InstantiateNonemptyAssoc:** As expected, CVC4$^{\text{fmf}}$ also returns **unsat** when the following invariants are added to WritesReviews:

  ```
  Paper.allInstances()–>notEmpty().
  Researcher.allInstances()–>notEmpty().
  Paper.allInstances()–>exists(p|p.author–>notEmpty()).
  Paper.allInstances()–>exists(p|p.referee–>notEmpty()).
  ```

  Thus, we can conclude that there is no valid instance of WritesReviews with at least one paper, one researcher, one instance of the manuscript-author association, and one instance of the submission-referee association.

- **InstantiateInvariantIgnore:** CVC4$^{\text{fmf}}$ returns **sat** when the invariants **oneManuscript** and **oneSubmission** are removed from WritesReviews. Thus, we can conclude that, if the invariants **oneManuscript** and **oneSubmission** are ignored, there exists at least one valid instance of WritesReviews.

We can now compare these results with the ones obtained by analyzing WritesReviews using USE, as reported in [38]. In particular,

- **InstantiateNonemptyClass:** Selecting as search state space the instances of WritesReviews with at most four researchers and four papers, USE is not able to find a valid instance of WritesReviews with at least one researcher and one paper. Again,

Question	*Answer*	*Time (in secs)*	*Remarks*
InstantiateNonemptyClass	unsat	0.66	
InstantiateNonemptyAssoc	unsat	1.70	
InstantiateInvariantIgnore	sat	0.22	

Table 3.3: Analyzing WritesReviews with OCL2MSFOL

Question	*Answer*	*Time (in secs)*	*Remarks*
InstantiateDisjointInheritance	unsat	0.08	
InstantiateMultipleInheritance	sat	0.06	

Table 3.4: Analyzing DisjointSubclasses with OCL2MSFOL

notice that this answer is *inconclusive*, since a valid instance of WritesReviews with at least one researcher and one paper may still exist outside the selected range. On the contrary, the answer provided by OCL2MSFOL guarantees that no valid instance of WritesReviews exists with at least one researcher and one paper.

- **InstantiateNonemptyAssoc:** The result is exactly as in the case of **InstantiateNonemptyClass**.

- **InstantiateInvariantIgnore:** Having removed from WritesReviews the constraints **oneManuscript** and **oneSubmission**, and selecting as search state space the instances of WritesReviews with exactly one paper and at most four researchers, USE finds a valid instance of WritesReviews. Thus, as with OCL2MSFOL, we can conclude that, if the constraints **oneManuscript** and **oneSubmission** are ignored, there is at least one valid instance of WritesReviews.

DisjointSubclasses

In Table 3.4 we show the results of analysing, using OCL2MSFOL, the questions posed in the benchmark about DisjointSubclasses. In particular,

- **InstantiateDisjointInheritance:** CVC4$^{\text{fmf}}$ returns **unsat** when the following invariant is added to DisjointSubclasses:

  ```
  D.allInstances()->notEmpty()
  ```

  Thus, we can conclude that there is no valid instance of DisjointSubclasses with at least one element of class D.

- **InstantiateMultipleInheritance:** CVC4$^{\text{fmf}}$ returns **sat** when the following invariant is added to DisjointSubclasses

  ```
  D.allInstances()->notEmpty()
  ```

  and at the same time the invariant **disjointBC** is removed from DisjointSubclasses. Thus, we can conclude that, if the constraint **disjointBC** is ignored, there is at least one instance of DisjointSubclasses with at least one element of class D.

We can now compare these results with the ones obtained by analyzing DisjointSubclasses using USE, as reported in [38]. In particular,

- **InstantiateDisjointInheritance:** Selecting as search state space the instances of DisjointSubclasses with exactly one element of class A, one of class B, one of class C, and one of class D, USE is not able to find a valid instance of DisjointSubclasses. Notice that this answer is again *inconclusive*, since a valid instance of DisjointSubclasses may still exist outside of the selected range. On the contrary, the answer provided by OCL2MSFOL guarantees that no valid instance of DisjointSubclasses exists at all with at least one element of class D.

- **InstantiateMultipleInheritance:** Having eliminated from DisjointSubclasses the constraint **disjointBC**, and selecting as search state space the instances of DisjointSubclasses with exactly one element of class A, one of class B, one of class C, and one of class D, and removing from DisjointSubclasses the constraint **DisjointBC**, USE finds an instance of DisjointSubclasses. Therefore, as with OCL2MSFOL, we can conclude that, if the constraint **DisjointBC** is ignored, there is at least one instance of DisjointSubclasses with at least one element of class D.

Chapter 4

Application domains

© Joaquín S. Lavado, QUINO. Toda Mafalda, Penguin Random House, España

In this chapter we report on a number of non-trivial case studies that we carried out, at different stages in our work, in order to validate the applicability (efficiency, usability) of our mapping(s) from OCL to (many-sorted) first-order logic. Following the evolution of our research, and, in particular, of our growing understanding of the different available SMT solvers and of their heuristics, each case study used a different version of our mapping and (in some cases also) a different SMT solver. In this regard, the actual figures reported here regarding execution times are not as relevant as the fact that the case studies cover a wide range of different application domains of practical interest, and that they were successfully completed even using (in some cases) rather preliminary versions of our mapping and SMT solvers.

4.1 Checking model satisfiability

4.1.1 The eHealth record management system

This case study was proposed within NESSoS, the European Network of Excellence on Engineering Secure Future Internet Software Services and Systems [62]. It consists of a electronic health record management (EHRM). Electronic health records (EHR) record information created by, or on behalf of, a health professional in the context of the care of a patient. Electronic health records are highly sensitive. Here we check satisfiability over data model and a set of OCL invariants, using OCL2MSFOL.

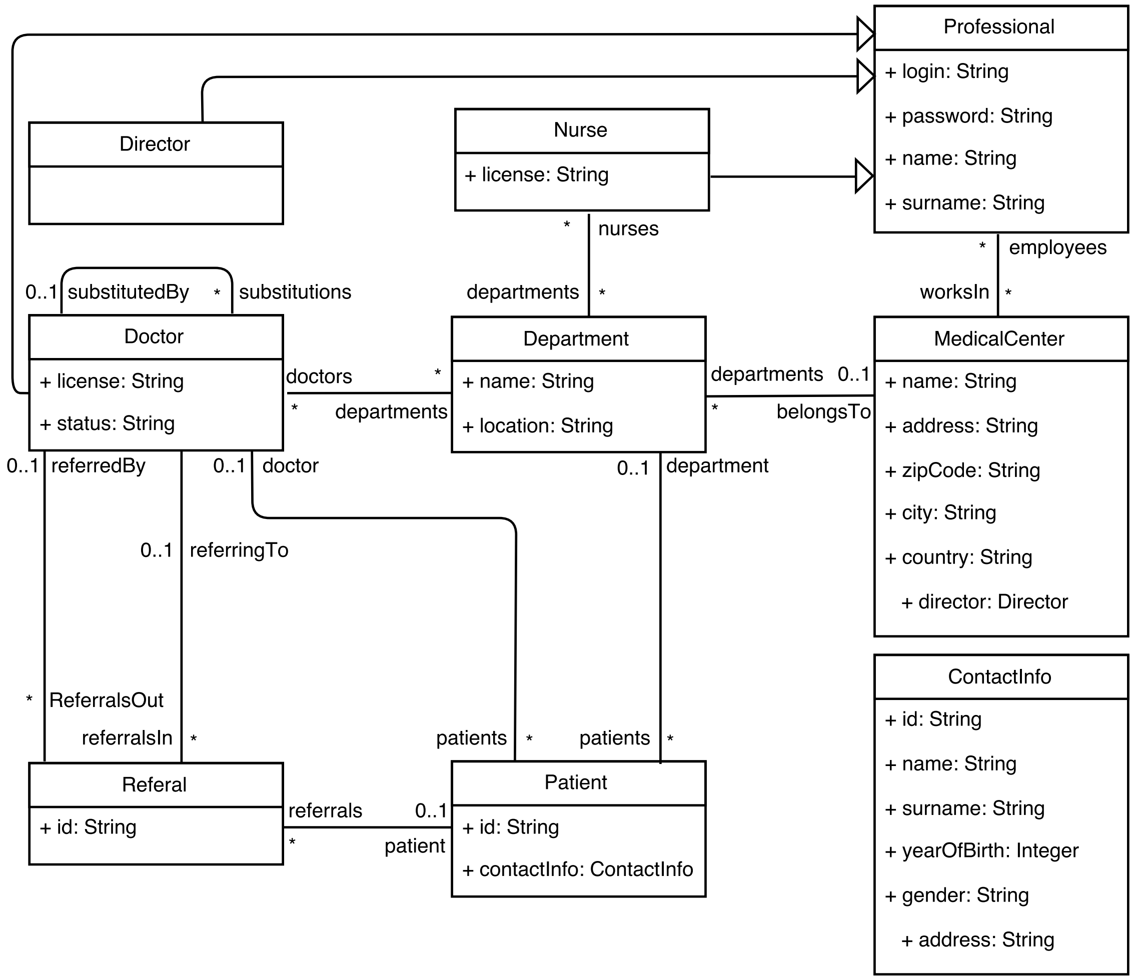


Figure 4.1: EHR: a data model for a basic eHealth Record Management System.

Data model

The data model for our case study, shown in Figure 4.1, models a basic eHealth management system. It contains 9 classes, 3 generalizations, 24 attributes, and 10 associations.

We discuss below just the entities, attributes, and association-ends that are required for our case study.

Professional. This class represents the EHRM's users. The medical centers where a user works are linked to the user through the association-end worksIn.

MedicalCenter. This class represents medical centers. The departments belonging to a medical center are linked to the medical center through the association-end departments. The professionals working for a medical center are linked to the medical center through the association-end employees.

Doctor. This class represents doctor information. Doctor is a subclass of Professional. The departments where a doctor works are linked to the doctor's information through the association-end departments. The patients treated by a doctor are linked to the doctor's information through the association-end patients. Doctors can be substituted by another doctor through the association-end substitutedby.

Director. This class represents director information. Director is a subclass of Professional.

Nurse. This class represents nurse information. Nurse is a subclass of Professional. The departments where a nurse works are linked to the nurse's information through the association-end departments.

Department. This class represents departments. The medical center to which a department belongs is linked to the department through the association-end belongsTo. The doctors working in a department are linked to the department through the association-end doctors. The patients treated in a department are linked to the department through the association-end patients.

Patient. This class represents patients. The doctor treating a patient is linked to the patient through the association-end doctor. The department where a patient is treated is linked to the patient through the association-end department.

Referral. This class represents referrals. The doctor can by referred by a referral through the association-end referredBy. And a patient can be also referred by a referral through the association-end patient.

ContactInfo. This class contain the contact information.

OCL invariants

The data model for our case study also contains 38 invariants, which can be grouped in the 5 categories:[1]

G1. Invariants stating the non-emptiness of certain classes. For example, *There must be at least one medical center.*

MedicalCenter.allInstances()–>notEmpty().

There are 9 invariants in this category, one for each class in the data model: namely, MedicalCenter, Department, Professional, Director, Doctor, Nurse, Referral, Patient, and ContactInfo.

G2. Invariants stating the definedness of certain attributes. For example, *The name of a professional cannot be undefined.*

Professional.allInstances()–>forAll(p|not(p.name.oclIsUndefined())).

There are 11 invariants in this category, stating the definedness of the attributes name (MedicalCenter), city (MedicalCenter), country (MedicalCenter), director (MedicalCenter), name (Department), name (Professional), surname (Professional), login (Professional), password (Professional), contactInfo (Patient), and license (Nurse),

[1] Notice that the given set of invariants is not intended to be complete.

G3. Invariants stating the uniqueness of certain data with respect to certain attributes. For example, *There cannot be two different doctors with the same medical license number.*

```
Doctor.allInstances()->forAll(d1|Doctor.allInstances()->forAll(d2|
    not(d1=d2) implies not(d1.license=d2.license)).
```

There are 5 invariants in this category, stating the uniqueness of certain data with respect to different attributes. In particular, data of the class MedicalCenter, when considering together address, zipCode, city, and country; data of the class Professional, with respect to login; data of the class Doctor, with respect to license; data of the class Nurse, with respect to license; and data of the class Referral, when considering together patient, referringTo, and referredBy.

G4. Invariants stating the non-emptiness of certain association-ends. For example, *Every medical center should have at least one employee.*

```
MedicalCenter.allInstances()->forAll(m|m.employees->notEmpty()).
```

There are in total 6 invariants in this category, stating the non-emptiness of the association-ends employees (MedicalCenter), belongsTo (Department), doctors (Department), nurses (Department), patient (Referral), referredBy (Referral), doctor (Patient), and department (Patient).

G5. Other invariants: namely,

- *A patient should be treated in a department where his/her doctor works.*

  ```
  Patient.allInstances()->forAll(p|
    p.doctor.departments->exists(d|d=p.department))
  ```

- *A professional cannot have an empty string as password.*

  ```
  Professional.allInstances()->forAll(p|not(p.password = ''))
  ```

- *A professional cannot have an empty string as login.*

  ```
  Professional.allInstances()->forAll(p|not(p.login = ''))
  ```

- *If a doctor's status is 'unavailable', then he/she should have a substitute different from him/herself.*

  ```
  Doctor.allInstances()->forAll(d|d.status='unavailable' implies
      (not(d.substitutedBy.oclIsUndefined() or d.substitutedBy = d)))
  ```

- *If doctor's status is 'available', then he/she should not have any substitute.*

  ```
  Doctor.allInstances()->forAll(d|
    d.status='available' implies d.substitutedBy.oclIsUndefined())
  ```

- *If a doctor is a substitute of other doctors, his/her status should be 'available'.*

  ```
  Doctor.allInstances()->forAll(d|
    d.substitutions->notEmpty() implies d.status="available")
  ```

- *If a referral indicates both the patient and the doctor whom the patient is referred to, then the doctor who is referring the patient cannot be the same than the doctor whom the patient is referred to.*

```
Referral.allInstances()->forAll(r|
  not(r.patient.oclIsUndefined() and r.referringTo.oclIsUndefined())
    implies not(r.referringTo = r.referredBy))
```

4.1.2 Checking data model satisfiability

In what follows we report on the results obtained in our case study. We use our mapping OCL2MSFOL, and we check the generated theories using the SMT solver CVC4$^{\mathrm{fmf}}$. Finally, we ran the code generated on a machine with an Intel Core2 processor running at 2.83 GHz with 8GB of RAM. The code generated by the mapping is available in [70].

The first experiment we carried out was to check whether there exists an instance of the case study's data model satisfying all the given invariants. Using CVC4$^{\mathrm{fmf}}$ we obtained the answer in 6 seconds. In particular, we show in Figure 4.2 the valid instance of the case study's data-model automatically generated by CVC4$^{\mathrm{fmf}}$.

The second experiment consisted in checking whether there exists a valid instance of the case study's data model for which the following also holds:

- *There exists only one doctor and his/her status is "unavailable".*

```
Doctor.allInstances()->exists(d1|
  d1.status='unavailable' and Doctor.allInstances()->forAll(d1|d1=d2)).
```

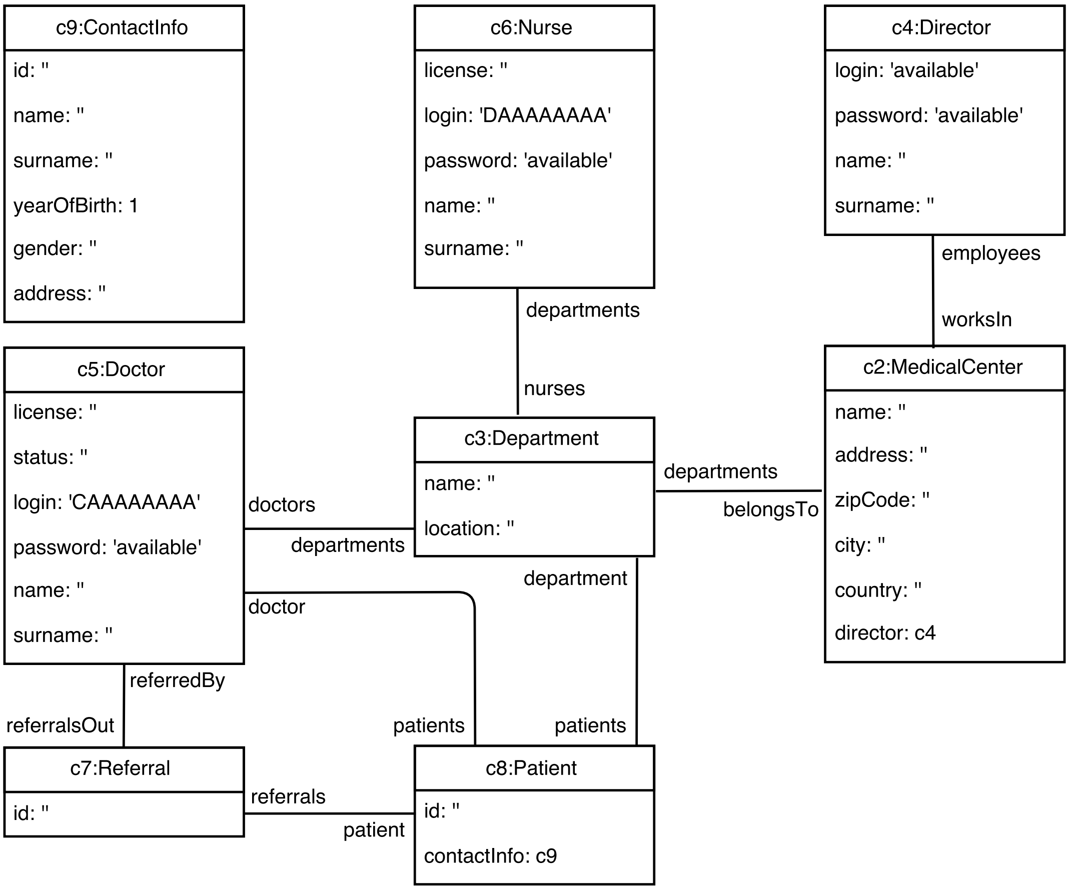


Figure 4.2: Automatically generated instance of the case study's data model satisfying all the invariants.

Using CVC4$^{\mathrm{fmf}}$ we obtained unsat in about 4 seconds. This is the expected result, since there is an invariant stating that "if a doctor's status is 'unavailable', then he/she should have a substitute different from him/herself."

Finally, the third experiment consisted in checking whether there exists a valid instance of the case study's data model for which the following holds:

- *There exists only one doctor*

  Doctor.allInstances->exists(d1|Doctor.allInstances->forAll(d2|d1=d2)).

In this experiment, we obtained sat in 6 seconds. This may be an unexpected result, since there is an invariant stating that "that there must be at least one referral" and there is another invariant stating that "if a referral has defined both the patient and the doctor to whom the patient is referred to, then the doctor referring the patient cannot be the same than the doctor the patient is referred to". However, the result returned by CVC4$^{\mathrm{fmf}}$ is correct, since there is no invariant stating that every referral has defined the doctor to whom the patient is referred to. Therefore, it is certainly possibly a valid instance of the case study's data model basically containing only one doctor and only one referral in which the doctor to whom the patient is referred to is not defined. This is in fact the instance automatically generated by CVC4$^{\mathrm{fmf}}$ when returning sat in this experiment. Of course, if we want to "correct" this, we can simply add to the case study's data model the following invariant:

Referral.allInstances()->forAll(r|not(r.referringTo.oclIsUndefined()))

Then, if we run again the experiment, the answer returned by CVC4$^{\mathrm{fmf}}$, in about 4 seconds, is unsat.

4.1.3 Concluding remarks

We have reported here on a non-trivial case study, which shows that our mapping can be used for efficiently checking the satisfiability of non-trivial UML/OCL models using SMT solvers with finite model finding capabilities.

However, the reader should not forget that our results ultimately depend on the (hard-won) positive logical interaction between (i) our formalization of UML/OCL in MSFOL, and (ii) the heuristics implemented in the SMT solver.

This means, in particular, that changes in the SMT solver's heuristics may have consequences (hopefully positive) in the applicability of OCL2MSFOL. It also means that a deeper understanding from our part of the SMT solver's heuristics may lead us to redefine OCL2MSFOL.

4.2 Validating and instantiating metamodels

4.2.1 The Core Security Metamodel (CSM)

Development of secure applications is a challenging task due to the evolvable risks that threaten any system under design. Nowadays, the exposure of systems to cloud environments claims for a stronger development approach able to support a large number of complex security requirements and interplay in the creation of cloud applications. Most of the proposed approaches agree in the necessity to sit a solid and affordable engineering process that can prevent, from design time, non-secure states due to wrong security

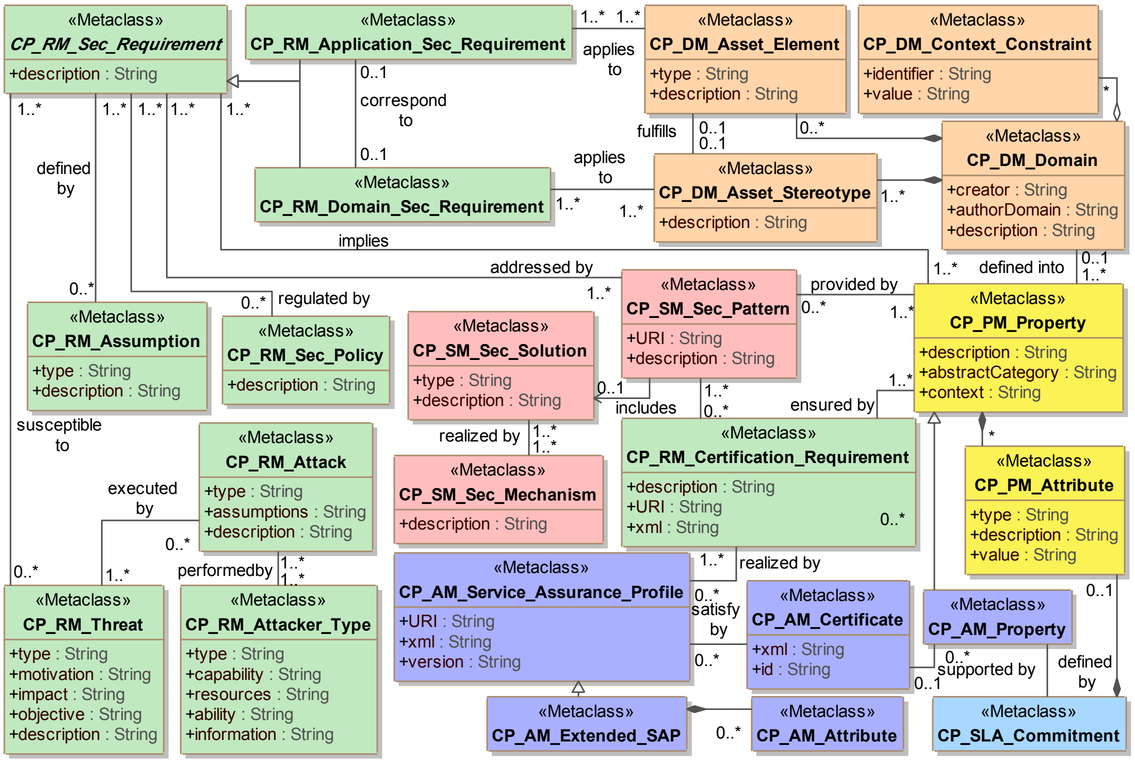


Figure 4.3: Core Security Metamodel

mechanisms used as a late solution [43]. In line with this approach, the CluUMULUS [23] and the PARIS [74] EU projects have proposed a security engineering process to develop secure applications. It includes a complete Model Based System Engineering (MBSE) methodology to address the different stages involved in the development of secure and privacy preserving applications. We focus here on the first stage of the corresponding work flow: the Core Security metamodel (CSM), designed to gather and represent the security knowledge. The CSM and the OCL validation rules imposed on it establish a language that supports, validates and drives instance creation and subsequent steps of the engineering process.

Data model

Reflecting the complexity of the security field, the CSM is a composition of 6 sub-models that address different security expertise sub-areas. Thus, CSM instantiation is facilitated by these groups of related elements which are displayed in the metamodel with different colors, as shown in Figure 4.3.[2] Next we describe these sub-models, also to understand how they fit together.

- *Requirement sub-model (green):* it is used to qualify security and certification requirements by means of security valuators, mechanisms and certified services.
- *Property sub-model (yellow):* it is used to describe abstract security properties involved in a security requirement, specifying its attributes and values.

[2] CSM has been already proved its use for real applications to integrate security mechanisms in high risk environments [83, 84], but using a different security engineering process in the context of the SecFutur Project [88].

- *Domain sub-model (brown):* it is used to describe the domain or context of the CSM instance, identifying the assets to be protected.
- *Solution sub-model (pink):* it is used to show how the security requirements will be achieved by means of solutions and security mechanisms.
- *Assurance sub-model (blue):* it is used to specify the assurance profile and the certification-related elements that would fulfill the certification requirements.
- *Service Level Agreement sub-model (light blue):* it is used to specify SLA agreements that may affect the security properties.

The CUMULUS engineering process aims not only at supporting experts to express their expertise into a model, but also to orchestrate an automated sorting and processing of that information to make it accessible and useful for non security experts. The effectiveness of this approach heavily relies on the OCL validation system which supports three goals in the CSM instantiation activity:

1. *Perform an active validation of the modeling process.* This validation raises a warning if the instance does not conform to the metamodel. It also highlights the pieces of information that are missing or wrong. This validation helps experts to avoid wrong specifications that would impact the run time of the system.
2. *Check that required information is present.* It validates whether a valid CSM instance lacks information that is needed by the engineering activities. E.g., transitive association between specific components, empty attributes, etc..
3. *Guide experts during the creation of the CSM instance.* They are guided towards the next piece of information that is needed and its goal in the engineering process.

Therefore the list of OCL constraints is expected to be consistent and reactive enough to support constant interaction with it. Our rules drive an incremental validation system that is gradually triggered within the MagicDraw modelling framework [55].

OCL Invariants.

The OCL validation package is composed of 33 invariants. Out of these, 27 are structural constraints restraining metamodel associations. Next, we introduce those OCL invariants that do not deal directly with multiplicities.

1. A domain instance must exist and be unique.

   ```
   CP_DM_Domain.allInstances()->size() = 1
   ```

2. A certification requirement needs to be associated with a service assurance profile.

   ```
   CP_RM_Certification_Requirement.allInstances()->forAll(c|
     (not c.URI.oclIsUndefined()) implies c.service_assurance_profile->notEmpty())
   ```

3. A certification requirement must be linked directly and through a security pattern to a security requirement and a property

```
CP_PM_Property.allInstances()->forAll(c|
  c.certification_requirement->notEmpty() implies
      c.certification_requirement.sec_pattern.sec_requirement
        ->intersection(c.sec_requirement)->notEmpty())
```

4. A certification requirement should be directly linked to a property and a security pattern for that property

```
CP_RM_Certification_Requirement->forAll(c|
  c.property->intersection(c.sec_pattern.property)->notEmpty())
```

5. An asset stereotype is set up over an asset element that must be considered by an application security requirement of that asset stereotype domain

```
CP_DM_Asset_Stereotype.allInstances()->forAll(c|
  (not c.asset_element.oclIsUndefined()) implies
    c.domain_sec_requirement.application_sec_requirement.
       asset_element->includes(c.asset_element))
```

6. A security pattern must display a security solution

```
CP_SM_Sec_Pattern.allInstances()->forAll(c|
  (not c.URI.oclIsUndefined()) implies (not c.sec_solution.oclIsUndefined()))
```

4.2.2 Validating the Core Security metamodel

We explain now the analysis that we perform on the OCL constrained CSM metamodel once it is translated to FOL. We use the mapping OCL2FOL^{+} and we check the generated theories using the SMT solver CVC4$^{\text{fmf}}$. Finally, we ran the code generated on a machine with an Intel Core2 processor running at 2.83 GHz with 8GB of RAM. The code generated by the mapping is available in [69]. We first tried to check whether the OCL constraints imposed on the CSM were or not unsatisfiable (and generate an example in the latter case) by feeding them to the SMT solvers Z3 [30] and CVC4 [10]. However, after more than 3 hours running, they did not return any result, and we decided to stop them. We know that this lack of result from Z3 and CVC4 is due to the fact that current techniques for dealing with quantified formulas in SMT are generally incomplete. In particular, they usually have problems to prove the unsatisfiability of a formula with universal quantifiers. Then, we decided to employ CVC4 as a finite model finder on our specification to check its satisfiability because the input required by it is the same input for the SMT solvers. CVC4 performed a bounded checking and succeeded by returning *sat* and automatically producing finite instances that conform to the OCL constrained CSM. Let us note that to work with the finite model finder CVC4, since the output of our tool [70] is SMT-LIB, we only needed to change in our mapping the sorts Int by a finite sort U. CVC4 run less than 30 seconds to answer `SAT` and return a simple CSM instance.

Then, we included additional OCL constraints to require a defined URI for all instances of CP_SM_Sec_Pattern, to contain a minimum of two CP_RM_Attack instances, and at least one instance of each of the next classes: CP_RM_Attack_Type, CP_RM_Certification_Requirement, CP_SM_Sec_Solution and CP_SM_Sec_Mechanism.
They ensure that generated instances contain at least a minimum amount of information

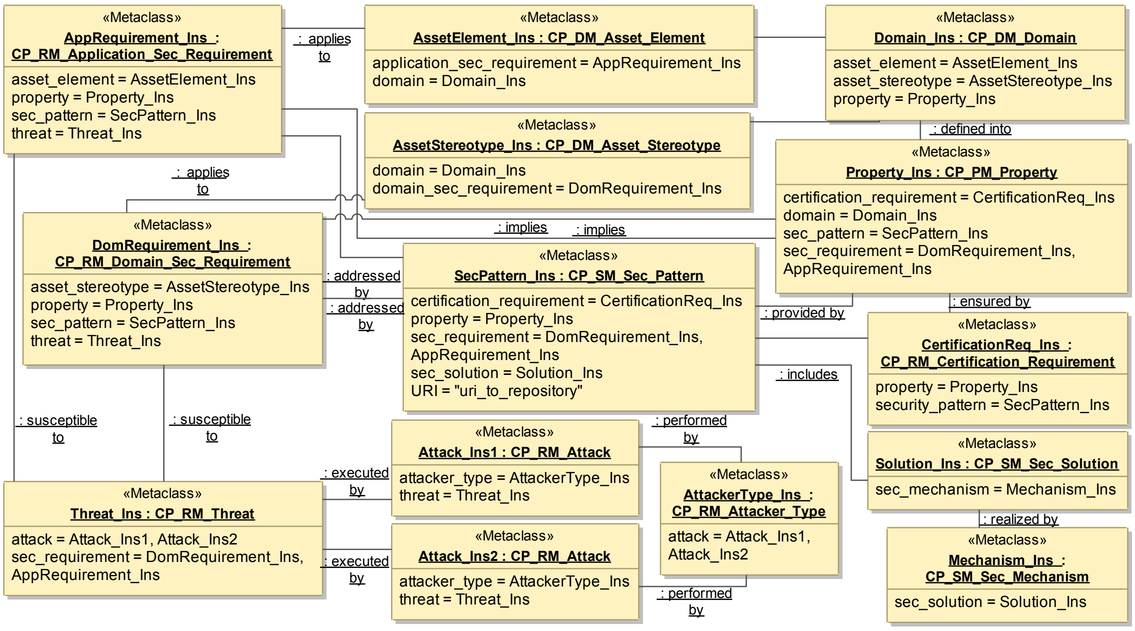

«Metaclass»
AppRequirement_Ins : CP_RM_Application_Sec_Requirement
asset_element = AssetElement_Ins
property = Property_Ins
sec_pattern = SecPattern_Ins
threat = Threat_Ins
: applies to
«Metaclass»
AssetElement_Ins : CP_DM_Asset_Element
application_sec_requirement = AppRequirement_Ins
domain = Domain_Ins
«Metaclass»
Domain_Ins : CP_DM_Domain
asset_element = AssetElement_Ins
asset_stereotype = AssetStereotype_Ins
property = Property_Ins
«Metaclass»
AssetStereotype_Ins : CP_DM_Asset_Stereotype
domain = Domain_Ins
domain_sec_requirement = DomRequirement_Ins
: defined into
«Metaclass»
Property_Ins : CP_PM_Property
certification_requirement = CertificationReq_Ins
domain = Domain_Ins
sec_pattern = SecPattern_Ins
sec_requirement = DomRequirement_Ins, AppRequirement_Ins
: applies to
: implies
: implies
«Metaclass»
DomRequirement_Ins : CP_RM_Domain_Sec_Requirement
asset_stereotype = AssetStereotype_Ins
property = Property_Ins
sec_pattern = SecPattern_Ins
threat = Threat_Ins
: addressed by
: addressed by
«Metaclass»
SecPattern_Ins : CP_SM_Sec_Pattern
certification_requirement = CertificationReq_Ins
property = Property_Ins
sec_requirement = DomRequirement_Ins, AppRequirement_Ins
sec_solution = Solution_Ins
URI = "uri_to_repository"
: provided by
: ensured by
«Metaclass»
CertificationReq_Ins : CP_RM_Certification_Requirement
property = Property_Ins
security_pattern = SecPattern_Ins
: includes
: susceptible to
: susceptible to
«Metaclass»
Attack_Ins1 : CP_RM_Attack
attacker_type = AttackerType_Ins
threat = Threat_Ins
: performed by
«Metaclass»
Solution_Ins : CP_SM_Sec_Solution
sec_mechanism = Mechanism_Ins
: executed by
«Metaclass»
Threat_Ins : CP_RM_Threat
attack = Attack_Ins1, Attack_Ins2
sec_requirement = DomRequirement_Ins, AppRequirement_Ins
«Metaclass»
AttackerType_Ins : CP_RM_Attacker_Type
attack = Attack_Ins1, Attack_Ins2
: realized by
«Metaclass»
Attack_Ins2 : CP_RM_Attack
attacker_type = AttackerType_Ins
threat = Threat_Ins
: executed by
: performed by
«Metaclass»
Mechanism_Ins : CP_SM_Sec_Mechanism
sec_solution = Solution_Ins

Figure 4.4: Automatically generated instance of the security metamodel presented in the Figure 4.3.

that makes them meaningful for a security expert. Then, we run CVC4 again with these additional constraints, and after less than 1 minute, the instance that we depict in Figure 4.4 was returned. The instances so obtained with CVC4 match structurally those obtained following the security engineering process and would allow to skip some of its steps (provided that we could automatically tailor the instances obtained by CVC4 to serve as inputs for the modeling framework). As we show next, these instances can be enhanced with knowledge (semantics) from the security domain so as they can serve as input for subsequent steps of the security engineering process.

4.2.3 Security enhanced CSM instances

As we already mentioned, the CSM is part of an assisted methodology, supported by the CUMULUS modelling tool [24], that has been initially conceived to take advantage of the multiple capabilities provided by the MagicDraw framework [55], particularly of its OCL validation engine. This methodology aims at supporting security experts to specify and communicate to system engineers how to solve security issues for cloud applications. When security experts design their models, i.e., CSM instances, the CUMULUS framework guides the construction of these instances (Domain Security Metamodels-DSMs) with the OCL rules that are continuously validated over them, raising warnings that claim for mandatory elements that are not yet present or errors. This process establishes a common format for the knowledge modeled, ensuring its applicability later on. The resulting instance (i.e., a DSM) is a validated artifact ready to transform security requirements into certification requirements and links to the solutions and mechanisms able to assure local system architectures and their interaction with cloud platforms [6].

For example, the rule 1 in Section 4.2.1 requires a unique domain instance. Experts dealing with security knowledge in the *EHealth* domain in cloud environments may describe a model for non security experts so as to improve a health care process (we follow

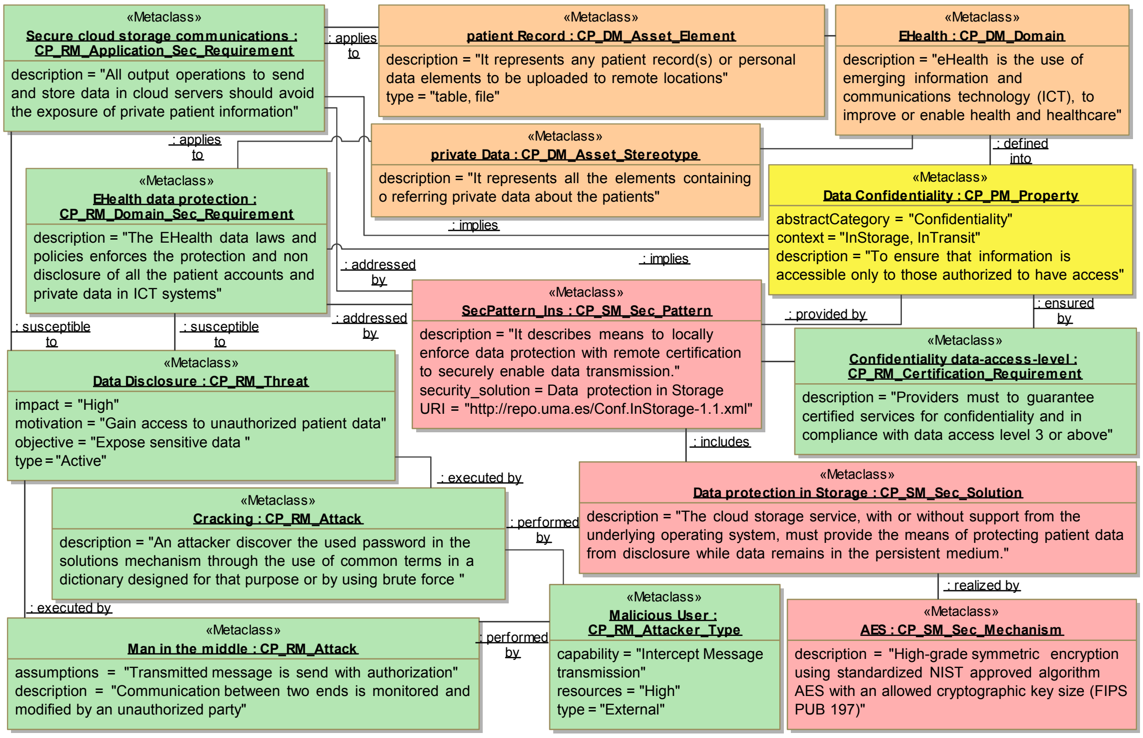

«Metaclass»
Secure cloud storage communications : CP_RM_Application_Sec_Requirement
description = "All output operations to send and store data in cloud servers should avoid the exposure of private patient information"
: applies to
«Metaclass»
patient Record : CP_DM_Asset_Element
description = "It represents any patient record(s) or personal data elements to be uploaded to remote locations"
type = "table, file"
«Metaclass»
EHealth : CP_DM_Domain
description = "eHealth is the use of emerging information and communications technology (ICT), to improve or enable health and healthcare"
: applies to
«Metaclass»
private Data : CP_DM_Asset_Stereotype
description = "It represents all the elements containing o referring private data about the patients"
: defined into
«Metaclass»
EHealth data protection : CP_RM_Domain_Sec_Requirement
description = "The EHealth data laws and policies enforces the protection and non disclosure of the patient accounts and private data in ICT systems"
«Metaclass»
Data Confidentiality : CP_PM_Property
abstractCategory = "Confidentiality"
context = "InStorage, InTransit"
description = "To ensure that information is accessible only to those authorized to have access"
: implies
: implies
: addressed by
«Metaclass»
SecPattern_Ins : CP_SM_Sec_Pattern
description = "It describes means to locally enforce data protection with remote certification to securely enable data transmission."
security_solution = Data protection in Storage
URI = "http://repo.uma.es/Conf.InStorage-1.1.xml"
: provided by
: ensured by
: addressed by
: susceptible to
: susceptible to
«Metaclass»
Confidentiality data-access-level : CP_RM_Certification_Requirement
description = "Providers must to guarantee certified services for confidentiality and in compliance with data access level 3 or above"
«Metaclass»
Data Disclosure : CP_RM_Threat
impact = "High"
motivation = "Gain access to unauthorized patient data"
objective = "Expose sensitive data "
type = "Active"
: includes
: executed by
«Metaclass»
Data protection in Storage : CP_SM_Sec_Solution
description = "The cloud storage service, with or without support from the underlying operating system, must provide the means of protecting patient data from disclosure while data remains in the persistent medium."
«Metaclass»
Cracking : CP_RM_Attack
description = "An attacker discover the used password in the solutions mechanism through the use of common terms in a dictionary designed for that purpose or by using brute force "
: performed by
: realized by
: executed by
«Metaclass»
Malicious User : CP_RM_Attacker_Type
capability = "Intercept Message transmission"
resources = "High"
type = "External"
«Metaclass»
AES : CP_SM_Sec_Mechanism
description = "High-grade symmetric encryption using standardized NIST approved algorithm AES with an allowed cryptographic key size (FIPS PUB 197)"
«Metaclass»
Man in the middle : CP_RM_Attack
assumptions = "Transmitted message is send with authorization"
description = "Communication between two ends is monitored and modified by an unauthorized party"
: performed by

Figure 4.5: Domain Security Metamodel

Figure 4.5). Domain specification is critical to upload DSMs into the appropriate repository, to classify the DSM content adequately. Once a valid domain instance has been created, the validation system triggers those rules that are not yet satisfied so as the model has to be extended to fulfill them.

Here we do not describe in full the DSM creation process. But we further describe the DSM instance in Figure 4.5. It contains as security requirements *EHealth data protection* and *Secure cloud storage communications*, both associated to the threat *Data Disclosure*. In addition, we have created an additional asset *patient record*, potential attacks as *Cracking* or *Man in the middle* and, finally, a common attacker type *Malicious User*. Probably, the most important part of a DSM is the selection of security patterns and certification requirements. The issue to be solved is described in the pattern, in our example, *means to locally enforce data protection with remote certification to securely enable data transmission*. How it should be guaranteed is specified by the certification requirement, in our example, *the usage of certified services for confidentiality and in compliance with data access level 3 or above*. Both plain descriptions have consequences in the security engineering process because they limit the solutions to be deployed for cloud applications. Recalling subsection 4.2.1, the last constraint requires that for a security pattern and a solution to be linked, the URI attribute of the pattern must be defined. This constraint demands intervention of the security expert since they search and select from existing repositories, through an API provided by the framework, a suitable pattern that also links a target solution, e.g., *Data protection in Storage* and a security mechanism, e.g., *AES*. As a result of the modeling process, security experts provide a complete artifact ready to fulfill security requirements addressing both the local mechanisms and the remote certification requirements.

Finally, we remark that both instances shown in Figures 4.4 and 4.5 resp., are structurally identical. Thus, the engineering process receive a shortcut from the use of automatic finite model finders that ease the path and reduce the time required to build instances since they can automatically generate them. Then, instances can be enhanced with security domain specific knowledge and trigger subsequent engineering activities.

4.2.4 Concluding remarks

We have introduced here a security metamodel constrained by 33 OCL rules. We have formally analyzed this metamodel, which is both complex and large, and its constraints, using our mapping to automatically map the metamodel and its constraints to first order logic. Then, we have successfully employed a finite model finder, CVC4, for checking the satisfiability of the resulting specification. Moreover, we have illustrated how the instances automatically generated by CVC4 conform to the aforementioned security metamodel and its constraints.

4.3 Analyzing security models

Nowadays, most of the main database management systems offer, in one way or another, the possibility of protecting data using fine-grained access control policies, i.e., policies that depend on dynamic properties of the system state. Reasoning about these policies typically amounts to answering questions about whether a security-related property holds in a (possibly infinite) set of system states.

In this section we discuss how to carry out formal reasoning about fine-grained access control policies using our mapping OCL2MSFOL.

4.3.1 SecureUML

SecureUML [11] is a modeling language for specifying fine-grained access control policies (FGAC) for actions on protected resources.

Using SecureUML, one can then model access control decisions that depend on two kinds of information:

1. *static information*, namely the assignments of users and permissions to roles, and the role hierarchy, and

2. *dynamic information*, namely the satisfaction of authorization constraints in the given system state.

Resources and Actions

In SecureUML the protected resources are the entities (classes), along with their attributes and association-ends (but not the associations as such), and the actions that they offer to the actors are those shown in the following table:

Resource	**Actions**
Entity	create, delete
Attribute	read, update
Association-end	read, create, delete

Authorization Constraints

In SecureUML, authorization constraints specify the conditions that need to be satisfied for a permission being granted to an actor (user) who requests it to perform an action. They are formalized using OCL, but they can also contain the following keywords:

- self: it refers to the root resource upon which the action will be performed, if the permission is granted. The root resource of an attribute or an association-end is the entity to which it belongs.
- caller: it refers to the actor that will perform the action, if the permission is granted.
- value: it refers to the value that will be used to update an attribute, if the permission is granted.
- target: it refers to the object that will be linked at the end of an association, if the permission is granted.

Permissions

SecureUML provides various syntactic sugar constructs for expressing FGAC policies in a more compact way. Basically, in the 'sweeter' presentation of a model, some roles may not have *explicitly* assigned any permission for some actions, while the following always holds in the de-sugared presentation of the model: every role has assigned exactly one permission for every action, and this permission has assigned exactly one authorization constraint. The rules for de-sugaring a SecureUML model are the following:

- *Role hierarchies.* Let *act* be an action and let r and r' be two roles. Suppose that r is a subrole of r' in $\mathcal{S}$, and that there is a permission in $\mathcal{S}$ for r' to execute *act* under the constraint *auth*. Then, when de-sugaring $\mathcal{S}$, we add a new permission to $\mathcal{S}$ for the role r to execute *act* under the same constraint *auth*.
- *Delete actions.* Let *entity* be an entity. Let *act* be the action delete(*entity*). Suppose that there is a permission in $\mathcal{S}$ for a role r to execute *act* under the constraint *auth*. Then, when de-sugaring $\mathcal{S}$, for every association-end *assoc* owned by *entity*, we add to $\mathcal{S}$ a new permission for r to execute delete(*assoc*) under the same constraint *auth*.
- *Opposite association-ends.* Let *assoc* and *assoc*$'$ be two opposite association-ends. Let *act* be the action create(*assoc*). Suppose that there is a permission in $\mathcal{S}$ for a role r to execute *act* under the constraint *auth*. Then, when de-sugaring $\mathcal{S}$, we add to $\mathcal{S}$ a new permission for the role r to execute create(*assoc*$'$) under the constraint that results from replacing in *auth* the variable *self* by *target* and the variable *target* by *self*. De-sugaring is done similarly when *act* is the action delete(*assoc*).
- *Denying by default.* Let r be a role and let *act* be an action. Suppose that there is no permission in $\mathcal{S}$ for the role r to execute *act*. When de-sugaring $\mathcal{S}$, we add to $\mathcal{S}$ a new permission for the role r to execute *act* under the constraint `false`.
- *Disjunction of authorization constraints.* Let r be a role and let *act* be an action. Suppose that there are n permissions in $\mathcal{S}$ for the role r to execute *act*. When de-sugaring $\mathcal{S}$, we replace these n permissions by a new permission and assign to it the authorization constraint that results from disjoining together all the authorization constraints of the original n individual permissions.

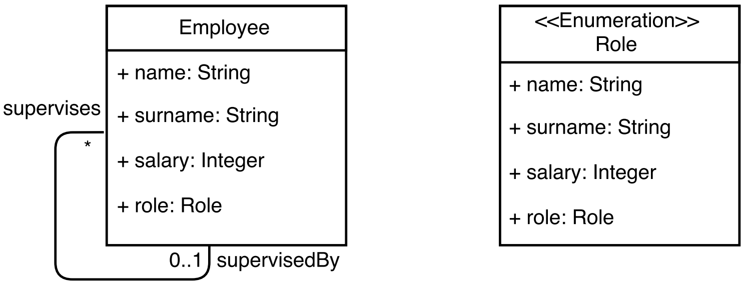


Figure 4.6: EmplBasic: a data model for employees' information.

In what follows, we will denote by Auth($\mathcal{S}$, r, act) the authorization constraint assigned, in the de-sugared presentation of the SecureUML model $\mathcal{S}$, to the role r's permission for performing the action act.

4.3.2 A running example

In Figure 4.6 we show a data model, named EmplBasic. This model specifies that every employee may have a name, a surname, and a salary; that every employee may have a supervisor and may in turn supervise other employees; and that every employee may take one of two roles: Worker or Supervisor. Notice that the association-end supervises has multiplicity 0..*, meaning that an employee may supervise zero or more employees, while the association-end `supervisedBy` has multiplicity 0..1 meaning that an employee may have at most one supervisor.

In Figure 4.7 we show two different instances of EmplBasic. In Instance 4.7a there are three employees, e_1, e_2 and e_3, and e_1 is supervised by e_2, e_2 is supervised by e_3, and e_3 has no supervisor at all. Moreover, e_1 has role Worker and both e_2 and e_3 have role Supervisor. Instance 4.7b has also three employees, e_1, e_2 and e_3, but this time e_1 is supervised by e_2, e_2 is supervised by itself, and e_3 has no supervisor at all. As before, e_1 has role Worker and both e_2 and e_3 have role Supervisor.

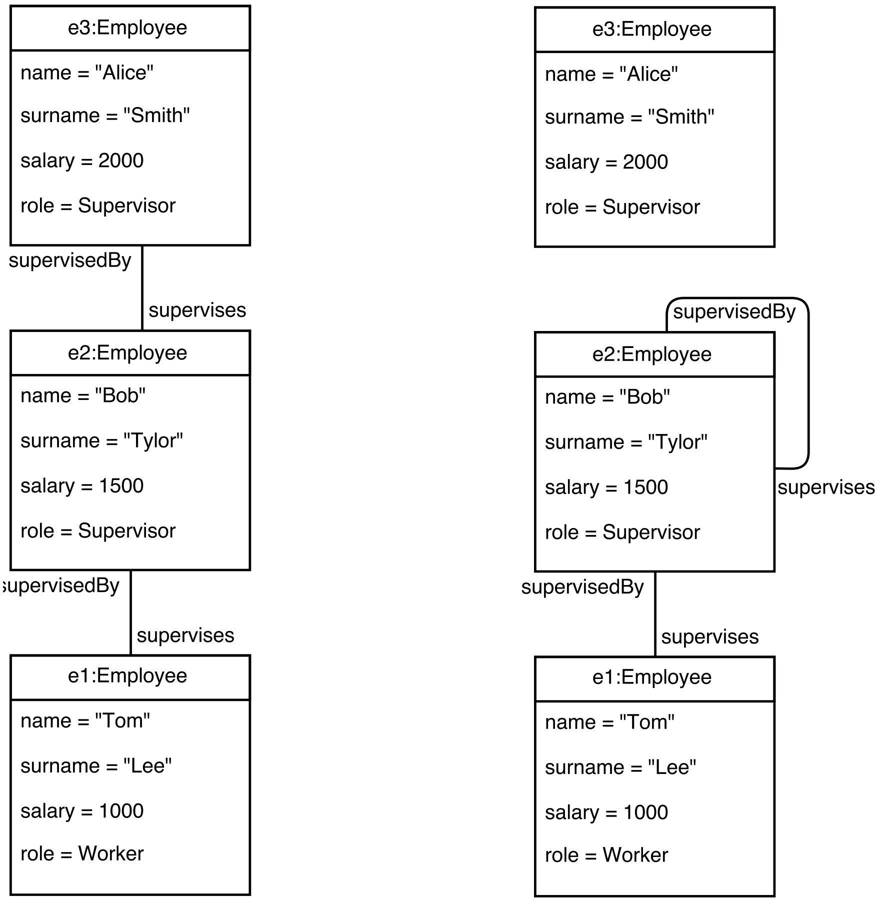
We can refine the model EmplBasic (Figure 4.6) by adding invariants to this model. In particular, consider the following constraints:

1. There is exactly one employee who has no supervisor.
2. Nobody is its own supervisor.
3. An employee has role Supervisor if and only if it has at least one supervisee.
4. Every employee has one role.

These constraints can be formalized in OCL as follows:

```
(1) Employee.allInstances()–>one(e|e.supervisedBy.oclIsUndefined())
(2) Employee.allInstances()–>forAll(e|not(e.supervisedBy = e))
(3) Employee.allInstances()–>forAll(e|
      (e.role = Supervisor implies e.supervises–>notEmpty())
      and (e.supervises–>notEmpty() implies e.role = Supervisor))
(4) Employee.allInstances()–>forAll(e|not(e.role.oclIsUndefined())
```

In what follows, we will refer to the constraint (1) as oneBoss, (2) as noSelf- Super, (3) as roleSuper, and (4) as allRole. Also, we will denote by Empl1 the refined version of



(a) Instance 1

(b) Instance 2

Figure 4.7: Two instances of EmplBasic

EmplBasic that includes as invariants the constraints oneBoss, noSelfSuper, roleSuper, and allRole. Notice that these four constraints evaluate to `true` in Instance 4.7a of EmplBasic (Figure 4.7), and therefore we say that this instance is a *valid* instance of Empl1. On the other hand, since noSelfSuper and roleSuper evaluate to false in Instance 4.7b of EmplBasic (Figure 4.7), we say that this other instance is a not a valid instance of Empl1.

In Figure 4.8 we show a SecureUML model, named Empl. This model specifies a basic FGAC policy for accessing the employees' information modeled in Empl1. Permissions are assigned to users depending on their *roles*, which can be Worker or Supervisor. Also, users with role Supervisor *inherit* all the permissions granted to users with role Worker, since Supervisor is a *subrole* of Worker. Finally, permissions are constrained by *authorization constraints*: namely,

1. A worker is granted permission to read an employee's salary, provided that it is its own salary, as specified by the authorization constraint caller = self.

2. A supervisor is granted unrestricted permission to read an employee's salary, as specified by the authorization constraint true.

3. A supervisor is granted permission to update an employee's salary, provided that it supervises this employee, as specified by the authorization constraint self.supervisedBy

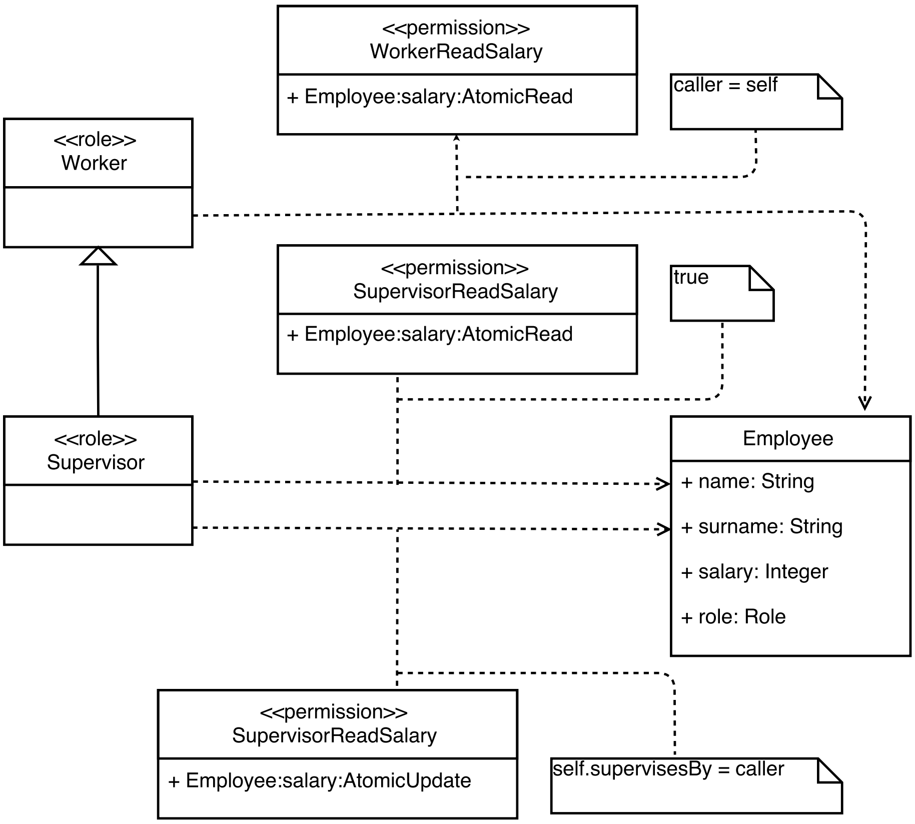


Figure 4.8: Empl: a SecureUML model for accessing employees' information.

= caller.

Example 19 *Consider the value of* $\mathrm{Auth}(\mathcal{S}, r, act)$ *in the following cases:*

$\mathrm{Auth}(\mathsf{Empl}, \mathsf{Worker}, \mathrm{update}(\mathsf{salary})) = \mathsf{false}$,

by the rule "denying by default".

$\mathrm{Auth}(\mathsf{Empl}, \mathsf{Supervisor}, \mathrm{update}(\mathsf{salary})) = (\mathsf{self.supervisedBy = caller\ or\ false})$,

by the combination of the rules "denying by default", "role hierarchies", and "disjunction of authorization constraints".

$\mathrm{Auth}(\mathsf{Empl}, \mathsf{Worker}, \mathrm{read}(\mathsf{salary})) = (\mathsf{caller = self})$.

$\mathrm{Auth}(\mathsf{Empl}, \mathsf{Supervisor}, \mathrm{read}(\mathsf{salary})) = (\mathsf{caller = self\ or\ true})$,

by the combination of the rules "denying by default", "role hierarchies", and "disjunction of authorization constraints". □

In what follows, when a data model $\mathcal{D}$ contains invariants $expr_1, \ldots, expr_n$, we will consider that $\mathrm{o2f}_{\mathrm{data}}(\mathcal{D})$ includes also the formulas $\bigcup_{i=1}^{n} \mathrm{o2f}_{\mathrm{true}}(expr_i)$.

Example 20 *Consider the following question about the model* Empl1*:* Is there a valid instance in which someone is supervised by one of its own supervisees? *Let us formalize the property that no employee is supervised by their own supervisees as follows:*

Employee.allInstances()−>forAll(e|e.supervises−>`excludes(e.supervisedBy))`.

We will refer to this expression as `noMixSuper`*. Then, according to Remark 1, the answer to our question is 'Yes' since*

$$\mathrm{o2f}_{\mathrm{data}}(\mathsf{Empl1}) \cup \{\neg(\mathrm{o2f}_{\mathrm{true}}(\mathsf{noMixSuper}))\}.$$

is satisfiable. Indeed, consider, for example, an instance of Empl1 *with just four employees,* e_1*,* e_2*,* e_3*, and* e_4*, such that* e_1 *is linked through the association-end* supervisedBy *with* e_4*, and similarly* e_3 *with* e_2*, and* e_2 *with* e_3*. Suppose also that* e_1 *is of role* Worker*, and* e_2*,* e_3*, and* e_4 *are of role* Supervisor*. This instance is certainly a valid one, since all the invariants evaluate to* true*. However, the expression* noMixSuper *evaluates to* false *because* e_2 *is linked through* supervisedBy *with* e_3*, but at the same time* e_2 *is also linked through the association-end* supervises *with* e_3 *(since* e_3 *is linked through* supervisedBy *with* e_2*).* □

4.3.3 Analyzing fine-grained access control policies

As discussed by [13], SecureUML models have a rigorous semantics. In particular, let $\mathcal{S}$ be a SecureUML model and let $\mathcal{I}$ be an instance of its underlying data model. Also, let u be a user, with role r, and let *act* be an action, with arguments *args*. Then, according to the semantics of SecureUML, $\mathcal{S}$ authorizes u to execute *act* in $\mathcal{I}$ if and only if $[\mathrm{Auth}(\mathcal{S}, r, act)]^{(u,args)}$ evaluates to `true` in $\mathcal{I}$, where $[\mathrm{Auth}(\mathcal{S}, r, act)]^{(u,args)}$ is the expression that results from replacing in $\mathrm{Auth}(\mathcal{S}, r, act)$ the keyword `caller` by u, and the keywords `self`, `value`, and `target` by the corresponding values in *args*.

In what follows, given a SecureUML model $\mathcal{S}$, we use the term *scenario* to refer to any valid instance of $\mathcal{S}$'s underlying data model in which a user requests permission to execute an action. For the sake of simplicity, we will assume that neither the user requesting permission nor the resource upon which the action will be executed can be *undefined.*

Next, we will explain, and illustrate with examples, how one can use our mapping from OCL to MSFOL to reason about SecureUML models. Unless stated otherwise, all our examples refer to the SecureUML model Empl Recall that this model's underlying data model is the model Empl1, which includes the invariants `oneBoss`, `noSelfSuper`, `roleSuper`, and `allRole`.

We organize our examples in blocks or categories. In the first block, we are interested in knowing if there is any scenario in which someone with role r will be allowed to execute an action *act*. Notice that, by Remark 1, the answer will be 'No' if and only if the following set of formulas is unsatisfiable:

$$\mathrm{o2f}_{\mathrm{data}}(\mathcal{D}) \cup \ \{\exists(caller)\exists(self)\exists(target)\exists(value)$$
$$(\mathrm{o2f}_{\mathrm{true}}(caller.\mathtt{role} = r) \wedge \mathrm{o2f}_{\mathrm{true}}(\mathrm{Auth}(\mathcal{S}, r, act)))\}.$$

Example 21 *Consider the following question:* Is there any scenario in which someone with role `Worker` is allowed to change the salary of someone else (including itself)? *Recall that*

Auth(Empl*,* Worker*,* update(salary))= `false`.

According to Remark 1, the answer to this question is 'No', since the following set of formulas is clearly unsatisfiable:

$$\mathrm{o2f}_{\mathrm{data}}(\mathsf{Empl1}) \cup \{\exists(caller)\exists(self)$$
$$(\mathrm{o2f}_{\mathrm{true}}(caller.\mathsf{role} = \mathsf{Worker}) \wedge \mathrm{o2f}_{\mathrm{true}}(\mathsf{false}))\},$$

(Note that $\mathrm{o2f}_{\mathrm{true}}(\mathsf{false})$ *returns* $\perp$*.) Indeed, there is no scenario in which the expression* `false` *can evaluate to* `true`*.* □

Example 22 *Consider the following question:* Is there any scenario in which someone with role Supervisor is allowed to change the salary of someone else (including itself)? *Recall that*

$$\mathrm{Auth}(\mathsf{Empl}, \mathsf{Supervisor}, \mathrm{update}(\mathsf{salary})) = (\mathsf{self.supervisedBy = caller\ or\ false}).$$

According to Remark 1, the answer to this question is 'Yes', since the following set of formulas is satisfiable:

$$\mathrm{o2f}_{\mathrm{data}}(\mathtt{Empl1}) \cup \{\exists(caller)\exists(self)(\mathrm{o2f}_{\mathrm{true}}(caller.\mathsf{role} = \mathsf{Supervisor}) \\ \wedge\ \mathrm{o2f}_{\mathrm{true}}(self.\mathsf{supervisedBy} = caller\ \mathsf{or\ false}))\}.$$

(Note: $\mathrm{o2f}_{\mathrm{true}}(self.\mathsf{supervisedBy} = caller)$ *returns supervisedBy(self) = caller). Consider, for example, a scenario with just two employees,* e_1 *and* e_2*, such that* e_1 *is linked with* e_2 *through the association-end* supervisedBy*. Suppose also that* e_1 *has role* Worker *and* e_2 *has role* Supervisor*. Clearly, for caller =* e_2 *and self =* e_1*, the expression* $self.\mathsf{supervisedBy} = caller$ *evaluates to* true *in this scenario.* □

Example 23 *Consider the following question:* Is there any scenario in which someone with role Supervisor is allowed to change its own salary? *Notice that in any scenario in which someone is requesting to change its own salary, the values of self (i.e., the employee whose salary is to be updated) and caller (i.e., the employee who is updating this salary) are the same. According to Remark 1, the answer to this question is 'No', since the following set of formulas is unsatisfiable:*

$$\mathrm{o2f}_{\mathrm{data}}(\mathsf{Empl1}) \cup \{\exists(caller)\exists(self)(\mathrm{o2f}_{\mathrm{true}}(caller.\mathsf{role} = \mathsf{Supervisor}) \\ \wedge\ \mathrm{o2f}_{\mathrm{true}}(self = caller\ \mathsf{and}\ (self.\mathsf{supervisedBy} = caller\ \mathsf{or\ false})))\}.$$

Indeed, notice that, in every valid scenario the invariant noSelfSuper *evaluates to* true*, which implies that there are no values for caller and self such that the expressions* $self = caller$ *and* $self.\mathsf{supervisedBy} = caller$ *both evaluate to* true*.* □

Example 24 *Consider the following question:* Is there any scenario in which someone with role Supervisor is allowed to change the salary of someone who has no supervisor at all? *Notice that in any scenario in which someone (caller) is requesting to change the salary of someone (self) who has no supervisor at all, the value of self.*supervisedBy *must be* null*. According to Remark 1, the answer to this question is 'No', since the following set of formulas is unsatisfiable:*

$$\mathrm{o2f}_{\mathrm{data}}(\mathsf{Empl1}) \cup \{\exists(caller)\exists(self)(\mathrm{o2f}_{\mathrm{true}}(caller.\mathsf{role} = \mathsf{Supervisor}) \\ \wedge\ \mathrm{o2f}_{\mathrm{true}}(self.\mathsf{supervisedBy} = \mathsf{null\ and}\ (self.\mathsf{supervisedBy} = caller\ \mathsf{or\ false})))\}.$$

□

Indeed, notice that, by assumption, *caller* is always a defined object, i.e., it can not be null, and therefore, if the expression $self.\mathsf{supervisedBy} = \mathsf{null}$ evaluates to `true`, then the expression $self.\mathsf{supervisedBy} = \mathsf{caller}$ evaluates to false.

In our second block of examples, we are interested in knowing if there is any scenario in which someone with role r will not be allowed to execute an action *act*. Notice that, by Remark 1, the answer will be 'No' if and only if the following set of formulas is unsatisfiable:

$$\begin{gathered}\mathrm{o2f}_{\mathrm{data}}(\mathcal{D}) \cup \{\exists(\mathit{caller})\exists(\mathit{self})\exists(\mathit{target})\exists(\mathit{value})\\ (\mathrm{o2f}_{\mathrm{true}}(\mathit{caller}.\mathsf{role} = r) \wedge \mathrm{o2f}_{\mathrm{true}}(\neg(\mathrm{Auth}(\mathcal{S}, r, \mathit{act}))))\}.\end{gathered}$$

Example 25 *Consider the following question:* Is there any scenario in which someone with role Supervisor is not allowed to change the salary of someone else (including itself)? *According to Remark 1, the answer to this question is 'Yes', since the following set of formulas is satisfiable:*

$$\begin{aligned}\mathrm{o2f}_{\mathrm{data}}(\texttt{Empl1}) \cup \{\exists(\mathit{caller})\exists(\mathit{self})(&\mathrm{o2f}_{\mathrm{true}}(\mathit{caller}.\mathsf{role} = \mathsf{Supervisor}) \wedge\\ &\neg(\mathrm{o2f}_{\mathrm{true}}(\mathit{self}.\mathsf{supervisedBy} = \mathit{caller}\ \mathsf{or}\ \mathsf{false})))\}.\end{aligned}$$

Consider, for example, a scenario with just three employees, e_1*,* e_2*, and* e_3 *such that* e_1 *is linked with* e_2 *through the association-end* supervisedBy*, and similarly* e_2 *with* e_3*; but* e_1 *is not linked with* e_3 *through the association-end* supervisedBy*. Suppose that* e_2 *and* e_3 *have role* Supervisor *and* e_1 *has role* Worker*. Clearly, for* $\mathit{caller} = e_3$ *and* $\mathit{self} = e_1$*, the expression* $\mathit{self}.\mathsf{supervisedBy} = \mathit{caller}$ *evaluates to* false *in this scenario.* □

In our third block of examples, we are interested in knowing if there is any scenario in which nobody with role r will be allowed to execute an action *act*. Notice that, by Remark 1, the answer will be 'No' if and only if the following set of formulas is unsatisfiable:

$$\begin{gathered}\mathrm{o2f}_{\mathrm{data}}(\mathcal{D}) \cup \{\exists(\mathit{self})\exists(\mathit{target})\exists(\mathit{value})\forall(\mathit{caller})\\ (\mathrm{o2f}_{\mathrm{true}}(\mathit{caller}.\mathsf{role} = r) \Rightarrow \neg(\mathrm{o2f}_{\mathrm{true}}(\mathrm{Auth}(\mathcal{S}, r, \mathit{act}))))\}.\end{gathered}$$

Example 26 *Consider the following question:* Is there any scenario in which nobody with role Supervisor is allowed to change the salary of someone else (including itself)*? According to Remark 1, the answer to this question is 'Yes', since the following set of formulas, is satisfiable:*

$$\begin{aligned}\mathrm{o2f}_{\mathrm{data}}(\texttt{Empl1}) \cup \{\exists(\mathit{self})\forall(\mathit{caller})(&\mathrm{o2f}_{\mathrm{true}}(\mathit{caller}.\mathsf{role} = \mathsf{Supervisor}) \Rightarrow\\ &\neg(\mathrm{o2f}_{\mathrm{true}}(\mathit{self}.\mathsf{supervisedBy} = \mathit{caller}\ \mathsf{or}\ \mathsf{false})))\}.\end{aligned}$$

Indeed, consider, for example, a scenario with just two employees, e_1 *and* e_2*, such that* e_1 *is linked with* e_2 *through the association-end* supervisedBy*. Suppose that* e_1 *has role* Worker *and* e_2 *has role* Supervisor*. Clearly, for* $\mathit{self} = e_2$*, for every value for caller, the expression* $\mathit{self}.\mathsf{supervisedBy} = \mathit{caller}$ *evaluates to* false*.* □

In our fourth block of examples, we are interested in knowing if, in every scenario, there is at least one object upon which nobody with role r will be allowed to execute an action *act*. Notice that, by Remark 1, the answer will be 'Yes' if and only if the following set of formulas is unsatisfiable:

$$\begin{gathered}\mathrm{o2f}_{\mathrm{data}}(\mathcal{D}) \cup \{\forall(\mathit{self})\exists(\mathit{target})\exists(\mathit{value})\exists(\mathit{caller})\\ (\mathrm{o2f}_{\mathrm{true}}(\mathit{caller}.\mathsf{role} = r) \wedge \mathrm{o2f}_{\mathrm{true}}(\mathrm{Auth}(\mathcal{S}, r, \mathit{act})))\}.\end{gathered}$$

Example 27 *Consider the following question:* In every scenario, is there at least one employee whose salary can not be changed by anybody with role Supervisor? *According to Remark 1, the answer to this question is 'Yes', since the following set of formulas is unsatisfiable:*

$$\mathrm{o2f}_{\mathrm{data}}(\texttt{Empl1}) \cup \{\forall(\mathit{self})\exists(\mathit{caller})(\mathrm{o2f}_{\mathrm{true}}(\mathit{caller}.\mathsf{role} = \mathsf{Supervisor}) \wedge \mathrm{o2f}_{\mathrm{true}}(\mathit{self}.\mathsf{supervisedBy} = \mathit{caller}\ \mathsf{or\ false}))\}.$$

Indeed, notice that in every valid scenario the invariant `oneBoss` *evaluates to* true*, which means that there is one employee in the scenario who has no supervisor. In other words, for every valid scenario, we can find a value for self such that no value for caller can be found such that the expression self*.supervisedBy = *caller evaluates to* true*.* □

Finally, we want to illustrate the importance of taking into account the invariants of the underlying data model when reasoning about FGAC policies. Let Empl2 be the data model that results from adding to the model EmplBasic the invariants noSelfSuper, roleSuper, allRole, plus the following invariant:

5. Everybody has one supervisor.

This invariant, which we will refer to as allSuper, can be formalized in OCL as follows:

Employee.allInstances()–>forAll(e|not(e.supervisedBy.oclIsUndefined())).

Example 28 *Consider the security model* Empl*, but this time with* Empl2 *as its underlying data model. Consider again the question that we asked ourselves in Example 26: namely,* is there any scenario in which nobody with role Supervisor is allowed to change the salary of someone else (including itself)*? According to Remark 1, the answer to this question is different from Example 26, namely, 'No', since the set of formulas is now unsatisfiable. Indeed, notice that in every valid scenario the invariants* allSuper *and* roleSuper *both evaluate to* true*, which means that, for each value for self, we can find a value for caller such that the expressions self*.supervisedBy = *caller and caller*.role = Supervisor *aboth evaluate to* true*.* □

Finally, let Empl3 be the data model that results from removing from Empl2, the invariant roleSuper.

Example 29 *Consider the security model* Empl*, but this time with* Empl3 *as its underlying data model. Consider, once again, the question that we asked ourselves in Example 26: namely,* is there any scenario in which nobody with role Supervisor is allowed to change the salary of someone else (including itself)*? According to Remark 1, the answer to this question is now different from Example 28, namely, 'Yes', since the set of formulas is now satisfiable. Indeed, consider a scenario with three employees e_1, e_2, and e_3, such that e_1 is linked with e_2 through the association-end* supervisedBy*, and similarly e_2 with e_3 and e_3 with e_1. Suppose also that e_2 and e_3 have role* Supervisor*, but e_1 has role* Worker*. (Notice that, since* roleSuper *is not included in* Empl3*, nothing prevents e_1 from not having the role* Supervisor*, despite the fact that it is linked with e_3 through the association-end* supervises*.) Clearly, for $self = e_3$, for every caller of role* Supervisor*, namely, e_2 and e_3, the expression self*.supervisedBy = *caller evaluates to* false*.* □

We briefly report here on our experience using the Z3 SMT solver [30] to automatically obtain the answers to the questions posed in the examples in Section 4.3.3. Table 4.1 below summarizes the results of our experiments. For each example, we show the time it takes Z3 to return an answer (in all cases, less than 1 second); the answer that it returns (in all cases, the expected one); and the first-order model that it generates when the answer is `sat`, i.e., when it finds that the input set of formulas is satisfiable. Each model represents a scenario (not necessarily the one discussed in Section 4.3.3 for the corresponding example), and here we simply indicate the number of employees that it contains, which employees are linked through the association-end supervisedBy, which employees have the role Worker, which employees have the role Supervisor, which employee is the one requesting permission to change the salary (*caller*), and which employee is the one whose salary will be changed (*self*) if permission is granted. We ran our experiments on a laptop computer, with a 2.66GHz Intel Core 2 Duo processor and 4GB 1067 MHz memory, using Z3 version 4.3.2 9d221c037a95-x64-osx-10.9.2. Finally, the input for Z3 has been generated using our tool SecProver [89]

4.3.4 Concluding remarks

We have presented a novel, tool-supported methodology for reasoning about fine-grained access control policies (FGAC). We have also reported on our experience using the Z3 SMT solver for automatically proving non-trivial properties about FGAC policies. The key component of our methodology is our mapping from OCL to first-order logic, which allows one to transform questions about FGAC policies into satisfiability problems in first-order logic. Although this mapping does not cover the complete OCL language, our experience shows that the kind of OCL expressions typically used for specifying invariants and authorization constraints are covered by our mapping. More intriguing is, however, the issue about the effectiveness of SMT solvers for automatically reasoning about FGAC policies. Ultimately, we know that there is a trade-off when using SMT solvers. On the one hand, they are necessarily incomplete and their results depend on heuristics, which may change. In fact, we have experienced (more than once) that two different versions of Z3 may return ‘sat’ and ‘unknown’ for the very same problem. This is not surprising (since two versions of the same SMT solver may implement two different heuristics) but it is certainly disconcerting. On the other hand, SMT solvers are capable of checking, in a fully automatic and very efficient way, the satisfiability of large sets of complex formulas. In fact, we have examples, involving more than a hundred non-trivial OCL expressions, which are checked by Z3 in just a few seconds.

4.4 Analyzing privacy models

4.4.1 Facebook: posting and tagging

Nowdays many people consider themselves as “Internet natives” (and many others are happy “Internet immigrants”): when they need information, they naturally open a browser and search for it; when they want to share information, they naturally post it on a social network. At the same time, privacy-related issues are a growing concern among users of social networking sites and among their regulators.

On December 21, 2011, the Office of the Irish Data Protection Commissioner (DPC) announced the results of its “thorough and detailed audit of Facebook’s practices and policies” [101], which includes, among many others, the following recommendations and findings [45] (see [46] for a DPC’s follow-up review):

Example	Time	Answer	Interpretation
21	0.078s	`unsat`	—
22	0.107s	`sat`	#employees = 3 supervisedBy $= \{(\mathrm{e}_3, \mathrm{e}_2), (\mathrm{e}_1, \mathrm{e}_2)\}$ Worker $= \{\mathrm{e}_1, \mathrm{e}_3\}$ Supervisor $= \{\mathrm{e}_2\}$ $\mathit{caller} = e_2, \mathit{self} = e_1$
23	0.041s	`unsat`	—
24	0.042s	`unsat`	—
25	0.306s	`sat`	#employees = 6 supervisedBy $= \{(\mathrm{e}_1, \mathrm{e}_2), (\mathrm{e}_2, \mathrm{e}_3), (\mathrm{e}_4, \mathrm{e}_2), (\mathrm{e}_5, \mathrm{e}_3), (\mathrm{e}_6, \mathrm{e}_3)\}$ Worker $= \{e_1, e_4, e_5, e_6\}$ Supervisor $= \{e_2, e_3\}$ $\mathit{caller} = e_3, \mathit{self} = e_1$
26	0.078s	`sat`	#employees = 1 supervisedBy $= \emptyset$ Worker $= \{e_1\}$ Supervisor $= \emptyset$ $\mathit{self} = e_1$
27	0.485s	`unsat`	—
28	0.060s	`unsat`	—
29	0.506s	`sat`	#employees = 15 supervisedBy $= \{(e_1, e_2), (e_2, e_4), (e_3, e_4), (e_4, e_6), (e_5, e_4),$ $(e_6, e_{12}), (e_7, e_4), (e_8, e_{14}), (e_9, e_4), (e_{10}, e_4), (e_{11}, e_{15}),$ $(e_{12}, e_{13}), (e_{13}, e_4), (e_{14}, e_4), (e_{15}, e_4)\}$ Worker = all Supervisor $= \emptyset$ $\mathit{self} = e_2$

Table 4.1: Automatic reasoning over the examples 21-29 introduced in Section 4.3.3.

Facebook must work towards:(i) simpler explanations of its privacy policies; (ii) easier accessibility and prominence of these policies during registration and subsequently; (iii) an enhanced ability for users to make their own informed choices based on the available information.

Many policies and procedures that are in operation are not formally documented. This should be remedy.

We recommend that Facebook introduce increased functionality to allow a poster to be informed prior to posting how broad an audience will be to view their post and that they be notified should the setting on that profile be subsequently change to make a post that was initially restricted available to a broader audience.

To Facebook's credit, over the past years, users have been equipped with new tools and resources which are designed to give them more control over their so-called Facebook experience, including: an easier way to select your audience when making a new post; inline privacy control on all your existing posts; the ability to review tags made by others before they appear on your profile; a tool to view your profile as someone else would see it; and more privacy education resources. Despite all these efforts, many users are still concerned about how to maintain their privacy or —in Mark Zuckerberg's own words— "rightfully questions how their information was protected" [102]. In our opinion, there are at least three reasons for this:

- Facebook's privacy policy is hardly trivial to understand. For example, when default policies and privacy settings for posting and tagging conflict to each other (which happens very often) the solution will depend (sometimes in a convoluted way) on the existing relationships among all the users involved: the owner of the timeline, the creator of the post, the creators of the tags, and the reader of the post.
- Facebook's privacy policy has been in a constant state of flux over the past few years [72], and it is prompted to change again in the future.
- Facebook's privacy policy is only informally and partially described in a collection of privacy education resources and blogs, which cannot provide a coherent and complete account of this policy.

As a consequence, even advanced Facebook users may find difficult to understand the actual visibility of a post.

To illustrate our point, we recall first the answers given in Facebook's 2013 Frequently Asked Questions (FAQ) [35] regarding the policy for posting and tagging:

- *If I post something on my friend's timeline, who gets to see it?*
  When you post something on a friend's timeline, who else gets to see it will depend on the privacy settings your friend has selected. If you want to write something to your friend privately, don't post it.
- *What does the 'Only Me' privacy setting mean?*
  Sometimes you might want certain posts visible only to you. Post with the 'Only Me' audience will appear on your timeline and in your news feed but won't be visible to anyone else. If you tag someone in an 'Only Me' post, they will be able to see the post.
- *When I share something, how do I choose can see it?*
  Before you post, look at the audience selector. Use the dropdown menu to choose who want to share a post with.
  - Public
  - Friends (+ friends of anyone tagged)
  - Only Me
  - Custom (Includes specific groups, friends lists or people you've specified to include or exclude)

Remember: anyone you tag in a post, along their friends may see the post. (...)
Note: When you post to another person's timeline, that person controls what audience can view your post.

- *What is tagging and how does it work?*
  A tag is a special kind of link. When you tag someone, you create a link to their timeline. The post you tag the person in is also added to the person's timeline. For example, you can tag a photo to show who's in the photo or post status update and say who you're with. (..)

  When you tag someone, they'll be notified. Also, if you or a friend tags someone in your post and it's set to 'Friends' or more, the post could be visible to the audience you selected plus friends of the tagged person.
  When someone adds a tag of you to a post, your friends may be able to see this. The tagged post also goes on your timeline.

Now, suppose that Bob, Alice, Ted, and Peter have Facebook profiles: Bob is a friend of Alice and Ted; Ted is a friend of Peter; Ted is not a friend of Alice; Peter is not a friend of Alice or Bob; and none of them has blocked to another.

To appreciate the challenge of understanding the actual visibility of a post, consider the scenarios S1–S4 below and try to justify (based on the previously recalled Facebook's policy) our answers to the given questions.[3]

S1 Alice posts a photo of herself, Bob and Ted in her timeline, and sets its audience to 'Friends'. Then, Alice tags Bob in this photo. *Question: Can Bob see the photo in Alice's timeline?* The answer is Yes.

S2 Alice has set to 'Only Me' the default audience for posts made by her friends in her timeline. Bob posts a photo in Alice's timeline. *Question: Can Bob see this photo in Alice's timeline?* The answer is Yes.

S3 Alice posts a photo of herself, Bob and Ted in her timeline, and set its audience to 'Friends'. Then, Bob tags Ted in this photo. *Question: Can Peter see this photo in Alice's timeline?* The answer is Yes.

S4 Bob posts a photo of himself, Ted and Alice in Alice's timeline. Alice has setting by default 'Only Me'. Then, Bob tags Ted in this photo. *Question: Can Peter see this photo in Alice's timeline?* The answer is No.

Clearly, as was explicitly requested in the DPC audit, Facebook should provide simpler explanations of its privacy policies. Even better, it should formally document these policies.

4.4.2 Modeling Facebook privacy policy

Data model

Facebook is a social network that "helps you connect and share with the people in your life." Each user has a *profile* that, basically, contains his/her personal information (name, surname, email, birthday, gender, and relationship status) and preferences (about music, television, movies, and games). Moreover, each user's *timeline* can displays posts

[3] These answers were obtained in 2013 on real Facebook scenarios.

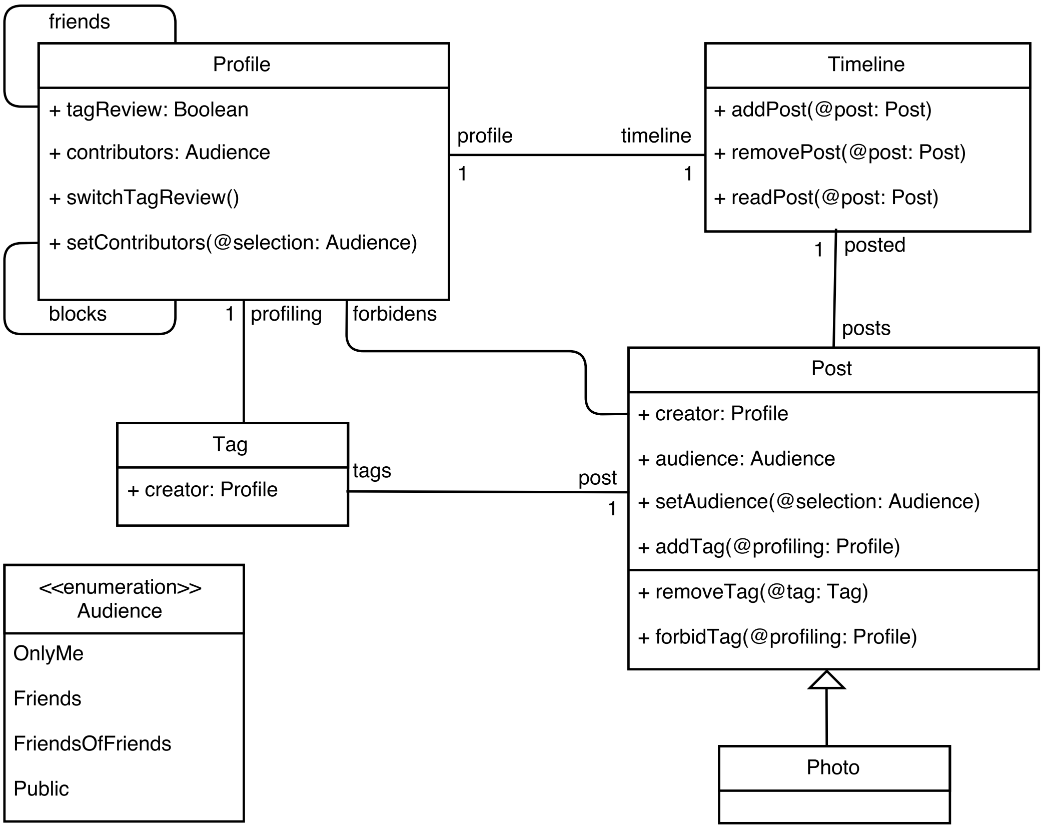


Figure 4.9: Modeling Facebook's data model (partial).

or stories, status updates, tags on status updates, comments to posts or stories, photos, comments to photos, and tags on photos.

We now introduce our data model for Facebook's profiles, timelines, posts, and tags. We do not intend to model these features in full, but rather those aspects that will play a role when modeling Facebook's policy for posting and tagging.

In Figure 4.9 we show how we can model, using a UML class diagram, profiles, timelines, posts, and tags. The following explanations highlight our main modeling decisions.

- Each profile, timeline, post, and tag, is modeled, respectively, as an instance of the classes Profile, Timeline, Post, and Tag.
- The method addPost(@post) adds the post @post to the given timeline.
- The method removePost(@post) removes the post @post from the given timeline.
- The method readPost(@post) read the post @post in the given timeline.
- Each photo is a type of post.
- Each profile is linked to exactly one timeline via timeline. This is the profile's timeline.
- Each profile is linked to those who are friends of him or her via friends.
- Each profile is linked to those that he or she has blocked via blocks.

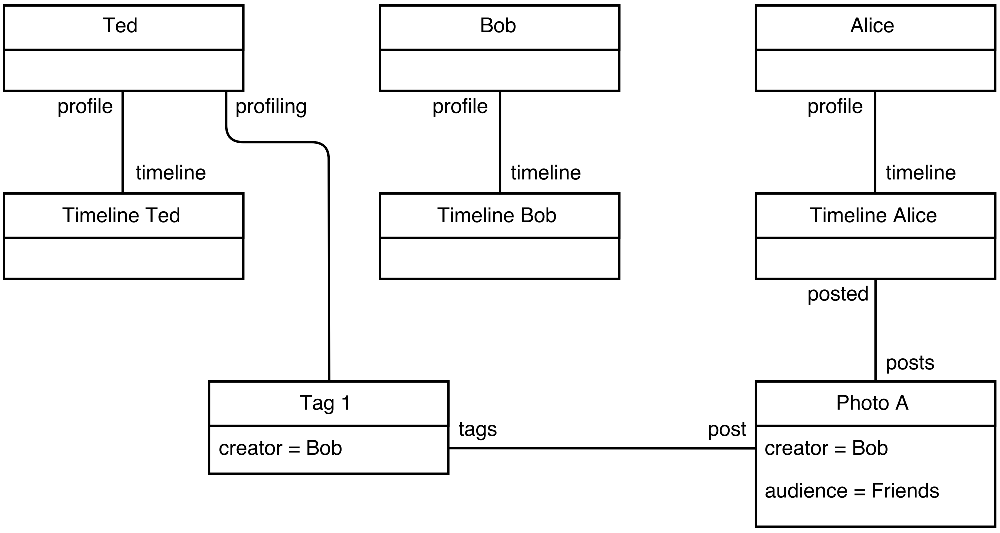


Figure 4.10: Modeling a Facebook scenario.

- Each profile has two attributes, namely, tagReview and contributors. The attribute tagReview holds the setting chosen by the profile's owner for Tag Review. The attribute contributors holds the setting chosen by the profile's owner for posting on its timeline.

- The method switchTagReview() switches on/off the Tag Review on the given profile.

- The method setContributors(@selection) set to @selection the intended post contributors to a given profile's timeline.

- Each post is linked to exactly one timeline via posted. This is the timeline on which the post is posted.

- Each post has two attributes, namely, creator and audience. These attributes hold, respectively, the post's creator and the post's selected audience.

- The method setAudience(@selection) set to @selection the intended audience for a given post.

- The method addTag(@profiling) adds a tag of a profile @profiling to a given post.

- The method removeTag(@tag) removes a tag @tag from a given post.

- The method forbidTag(@profiling) adds a profile @profiling to the list of profiles that can not be tagged on a given post.

- Each tag is linked to exactly one post via post. This is the post on which the tag appears.

- Each tag is linked to exactly one profile via profiling. This is the tag's target.

- Each tag has one attribute, namely, creator, that holds the tag's creator.

- Each post is linked to those profiles that can not be tagged in the post via forbidens.

We show in Figure 4.10 an instance of our data model for Facebook. It represents the following Facebook scenario: Bob, Alice and Ted have Facebook profiles. Bob is a friend of Alice, and Ted is a friend of Bob but not a friend of Alice. Alice's timeline has a photo that was posted by Bob in her timeline. Bob has tagged Ted in this photo.

Next we show how we can formalize *queries* about the Facebook scenario represented in Figure 4.10 using OCL. In particular,

- To query about Bob's friends, we can use the OCL expression:

  ```
  Bob.friends.
  ```

  This expression evaluates to Set{Alice, Ted} in our sample scenario.

- To query about friends of Alice's friends, we can use the OCL expression:

  ```
  Alice.friends.friends->asSet().
  ```

  This expression evaluates to Set{Ted, Alice} in our sample scenario (Alice is certainly a friend of any of her friends).

- To query about friends of Alice and their friends, but not including Alice herself, we can use the OCL expression:

  ```
  Alice.friends->union(Alice.friends.friends)->excluding(Alice).
  ```

  This expression evaluates to Set{Bob, Ted} in our sample scenario.

- To query about whether Ted is tagged in any of the posts appearing on Alice's timeline, we can use the OCL expression:

  ```
  Alice.timeline.posts.tags.profiling->includes(Ted).
  ```

  This expression evaluates to true in our sample scenario.

Privacy Policy

We are now ready to model, using SecureUML, the Facebook's 2013 policy for posting and tagging.[4] A word of caution: given the lack of a formal documentation, our understanding of this policy is based not only on the official information available at [35] (which is not always complete or coherent) but also on our own experiments using Facebook on "precooked" scenarios.

In what follows, for each method in our data model for Facebook, after describing the policy for executing this method, we will formally specified, using OCL, the corresponding authorization constraint.

[4] For the sake of simplicity, we have omitted some features, including: who is notified (and how) when a tag is added to a post; how (and by whom) a post's audience can be customized; how (and by whom) a post on which someone is tagged can be reviewed before it appears on his or her profile; how (and by whom) the maximum audience for posts appearing in someone's profile because he or she is tagged on them can be selected by default; how (and by whom) a tag can be added to a post different from a photo; how (and by whom) something different from a user can be tagged.

Method: switchTagReview() The next clause describes the policy for executing the method switchTagReview:

- anybody can turn on/off the option of reviewing any tag that anybody else wants to add to any post published in his or her timeline before they are actually published.

More formally, the permission to execute the method switchTagReview() has the following authorization constraint: caller = self.

Method: setContributors(@audience) The following clause describes the policy for executing the method setContributors:

- anybody can choose between not allowing anybody (except him or herself) to post on his or her timeline or allowing also their friends to post on his or her timeline.

More formally, the permission to execute the method setContributors(@audience) has the following authorization constraint: caller = self.

Method: setAudience(@audience) The following clause describes the policy for executing the method setAudience:

- anybody can select the audience for any post that is posted on his or her timeline.

More formally, the permission to execute the method setAudience(@audience) has the following authorization constraint: caller = self.posted.profile.

Method: addPost(@post) The following clauses describe the policy for executing the method addPost:

- anybody can add a post on his or her timeline.
- anybody can add a post on any of his or her friends' timelines, if the owner of this timeline has its preferences for posting set to 'Friends'.

More formally, the permission to execute the method addPost(@post) has the following authorization constraint:

```
caller = self.profile or (self.profile.contributors = 'Friends'
  and self.profile.friends->includes(caller)).
```

Method: removePost(@post) The following clause describes the policy for executing the method removePost:

- anybody can remove a post that he or she has posted on a timeline.

More formally, the permission to execute the method removePost(@post) has the following authorization constraint: caller = @post.creator.

Method: addTag(@profiling) The following clauses describe the policy for executing the method addTag:

- anybody can add a tag of him or herself, or of any of his or her friends, on a post that is posted on his or her timeline, unless this friend has previously untagged him or herself from this post.
- anybody can add a tag of him or herself, or of any of his or her friends, on a post that is posted on a timeline, unless the owner of the timeline has switched 'On' the tag review preferences and he or she is not the owner of the timeline, or unless this friend has previously untagged him or herself from this post.

More formally, the permission to execute the method addTag(@profiling) has the following authorization constraint:

```
((caller=@profiling or caller.friends–>includes(@profiling))
 and caller=self.posted.profile and self.forbidens–>excludes(@profiling))
or ((caller=@profiling or caller.friends–>includes(@profiling))
 and self.posted.profile.tagReview=false and self.forbidens–>excludes(@profiling)).
```

Method: removeTag(@tag) The following clauses describe the policy for executing the method removeTag:

- anybody can remove any tag of him or her on a post.
- anybody can remove any tag from a post that he or she has posted on a timeline.
- anybody can remove any tag that he or she has added to a post.

More formally, the permission to execute the method removeTag(@tag) has the following authorization constraint:

```
caller = @tag.profiling or caller = @tag.post.creator or caller = @tag.creator.
```

Method: forbidTag(@profiling) The following clause describes the policy for executing the method forbidTag:

- anybody can forbid anybody else to tag him or her again on a post.

More formally, the permission to execute the method forbidTag(@profiling) has the following authorization constraint: caller = @profiling.

Method: readPost(@post) The following clauses describe the policy for executing the method readPost:

- anybody can read any post that is posted on his or her timeline.
- anybody can read any post that was posted by him or her on a timeline, unless he or she is blocked by the owner of the timeline.
- anybody can read any post that has its audience selected to 'Friends', if he or she is a friend of the owner of the timeline.

- anybody can read any post that has its audience selected to ‘FriendsOfFriends’, if he or she is a friend of the owner of the timeline, or a friend of a friend of the owner of the timeline, unless he or she is blocked by the owner of the timeline.

- anybody can read any post that has its audience selected to ‘Public’, unless he or she is blocked by the owner of the timeline.

- anybody can read any post, if he or she is tagged on this post, unless he or she is blocked by the owner of the timeline.

- anybody can read any post that has its audience selected to ‘Friends’ and was created by the owner of the timeline, if he or she is a friend of somebody tagged on the post, unless he or she is blocked by the owner of the timeline.

More formally, the permission to execute the method readPost(@post) has the following authorization constraint:

```
caller = self.profile
or (caller= @post.creator and self.profile.blocks->excludes(caller))
or (@post.audience = 'Friends' and self.profile.friends->includes(caller))
or (@post.audience = 'FriendsOfFriends'
 and (self.profile.friends->includes(caller)
    or self.profile.friends.friends->includes(caller))
 and self.profile.blocks->excludes(caller))
or (@post.audience = 'Public' and self.profile.blocks->excludes(caller))
or (@post.tags.profiling->includes(caller) and self.profile.blocks->excludes(caller))
or (@post.audience = 'Friends' and @post.creator = self.profile
 and @post.tags.profiling.friends->includes(caller)
 and self.profile.blocks->excludes(caller)).
```

Next we validate our modeling of the policy for executing the method readPost, using the scenarios S1–S4 that we introduced in Section 4.4.1. Clearly, if our model is correct, the answers obtained in our real experiments about the visibility of the posts in these scenarios should correspond to the results of evaluating the method readPost’s authorization constraint on the corresponding instances of our data model for Facebook.

Recall that Bob is a friend of Alice and Ted; Ted is a friend of Peter; Ted is not a friend of Alice; Peter is not a friend of Alice or Bob; and none of them has blocked to another.

S1 Alice posts a photo of herself, Bob and Ted in her timeline, and sets its audience to ‘Friends’. Then, Alice tags Bob in this photo. *Question: Can Bob see the photo in Alice’s timeline?* The answer is Yes, because Alice has set her default audience to ‘Friends’ and Bob is a friend of Alice. Indeed, the readPost’s authorization constraint evaluates to true in this scenario, since

```
(@post.audience = 'Friends' and self.profile.friends->includes(caller))
```

evaluates to true when replacing @post by Alice’s photo, caller by Bob, and self by Alice’s timeline.

S2 Alice has set to ‘Only Me’ the default audience for posts made by her friends in her timeline. Bob posts a photo in Alice’s timeline. *Question: Can Bob see this photo in Alice’s timeline?* The answer is Yes, because Bob is the person who posted the photo and Bob is not blocked by Alice. Indeed, the readPost’s authorization constraint evaluates to true in this scenario, since

```
(caller = @post.creator and self.profile.blocks–>excludes(caller))
```

evaluates to true when replacing @post by Alice’s photo, caller by Bob, and self by Alice’s timeline.

S3 Alice posts a photo of herself, Bob and Ted in her timeline, and set its audience to ‘Friends’. Then, Bob tags Ted in this photo. *Question: Can Peter see this photo in Alice’s timeline?* The answer is Yes. because the audience selected by Alice is ‘Friends’ and, therefore, after Bob tags Ted the audience is extended to Ted and his friends. Indeed, the readPost’s authorization constraint evaluates to true in this scenario, since

```
(@post.audience = ’Friends’ and @post.creator = self.profile
  and @post.tags.profiling.friends–>includes(caller)
  and self.profile.blocks–>excludes(caller)).
```

evaluates to true when replacing @post by Alice’s photo, *caller* by Peter, and self by Alice’s timeline.

S4 Bob posts a photo of himself, Ted and Alice in Alice’s timeline. Alice has setting by default ‘Only Me’. Then, Bob tags Ted in this photo *Question: Can Peter see this photo in Alice’s timeline?* The answer is No, because the audience selected by Alice by default is ‘Only Me’, and Peter is neither the person who posted the photo, nor the person who is tagged in the photo. Indeed, the readPost’s authorization constraint evaluates to false in this scenario, when replacing @post by Alice’s photo, *caller* by Peter, and self by Alice’s timeline.

Next, we show how we can adjust our model to changes. Facebook’s privacy policy has been in a constant state of flux over the past years [72]. This is certainly the case for Facebook’s policy for tagging and posting, which is now explained in its 2014 FAQ [36] as follows:

- *When someone adds a tag to a photo or post I shared, who can see it?*
  When someone adds a tag to something you shared, it’s visible to: 1. The audience you chose for the post or photo. 2. The person tagged in the post, and their friends. If you’d like, you can adjust this visibility. You can select Custom, and uncheck the Friends of those tagged box.

Clearly, the possibility of not sharing a post with friends of those tagged in the post was not an option in 2013. In fact, if we consider again the scenarios S1–S4, we notice that our answers to the questions about the visibility of the posts in these scenarios remain valid, except for scenario S3: according to the new Facebook’s policy for tagging and posting, the question about whether Peter can see or not the photo in Alice’s timeline depends on whether Alice has checked or not the box ‘Friends of those tagged’ in her photo.

To adjust our SecureUML model of Facebook’s privacy policy to this latest change, we need first to modify our data model for Facebook in order to represent whether or not a post will be visible also to the tagged profile's friends. We do so by adding to the class Post a new Boolean attribute audExt. Then, we modify accordingly the readPost(@post)'s authorization constraint. In particular, we need to replace the last clause in the previous description of the policy for executing the method readPost by the following clause:

- anybody can read any post that has its audience selected to 'Friends' and was created by the owner of the timeline, if he or she is a friend of somebody tagged on the post, unless he or she is blocked by the owner of the timeline or *the owner has unchecked the box 'Friends of those tagged'*.

More formally, the permission to execute the method readPost(@post) will now have the following authorization constraint:

```
(caller = self.profile)
or (caller = @post.creator and self.profile.blocks->excludes(caller))
or (@post.audience = 'Friends' and self.profile.friends->includes(caller))
or (@post.audience = 'FriendsOfFriends' and (self.profile.friends->includes(caller)
or (self.profile.friends.friends->includes(caller) and self.profile.blocks->excludes(caller))
or (@post.audience = 'Public' and self.profile.blocks->excludes(caller))
or (@post.tags.profiling->includes(caller) and self.profile.blocks->excludes(caller))
or (@post.audience = 'Friends' and @post.creator = self.profile
 and @post.tags.profiling.friends->includes(caller)
 and self.profile.blocks->excludes(caller)
// the following conjunct is new
 and @post.audExt).
```

4.4.3 Analyzing Facebook privacy policy

SecureUML has a well-defined semantics that supports formal reasoning about its models. In particular, given a SecureUML model M, we can check that nobody, *for which a given property P holds*, will be allowed to execute a certain method X. Notice that this corresponds to proving that there is no *valid* instance of the underlying data model for which both the method X's authorization constraint and the property P evaluate to true. As explained before, we can automatically transform this type of problems into first-order satisfiability problems, and then use automated SMT solver tools to attempt to solve them.

We have applied this methodology to prove, as an example, that nobody will be allowed to read a post in a timeline if this person is blocked by the timeline's owner. First, we have formalized, using OCL, the properties that every valid instance of our data model for Facebook will have to satisfy, for example:

- If someone is blocked by someone else, then the former can not remain friend of the latter. Formally,

  ```
  Profile.allInstances()->forAll(p, q|
     p.blocks->includes(q) implies p.friends->excludes(q)).
  ```

- Nobody can be blocked by itself. Formally,

  ```
  Profile.allInstances()->forAll(p|p.blocks->excludes(p)).
  ```

Second, we have formalized, using OCL, the property of being blocked by the timeline's owner as follows:

self.profile.blocks−>includes(*caller*),

where *caller* refers to the person who wants to read the post and self refers to the timeline where the posted is posted.

Finally, after generating the corresponding satisfiability problem, we have used the SMT solver Z3 [30] to automatically prove the desired property, i.e., that nobody will be allowed to read a post in a timeline if this person is blocked by the timeline's owner.

4.4.4 Concluding remarks

To the best of our knowledge, no previous attempts have been made to rigorously formalize the Facebook privacy policy and, in particular, its policies for posting and tagging. There are, at least, two good reasons for this. First, as the DPC audit [45] has pointed out, "many policies and procedures that are in operation [in Facebook] are not formally documented." Second, the Facebook privacy policy has significantly changed over the past few years [72], in ways not always well-explained, as Zuckerberg has to admit in his blog [102]: "I'm the first to admit that we've made a bunch of mistakes. In particular (...) poor execution as we transitioned our privacy model two years ago."

Now, assuming that the Facebook privacy policy is formally documented, what will be the challenges for modeling this policy? Basically, as [90] discussed in detail, for modeling social networking privacy it is crucial to use a language able to formalize *fine-grained access control policies*. In other words, a basic role-based access control language, as proposed in [54], will only do part of the job. Thanks to its tight-integration with OCL, the language SecureUML [11] can deal with fine-grained access control policies, as we have shown in our case study. Of course, there are other options, but not many when having a formal semantics becomes a hard requirement. For example, XACML [63], which can be considered the standard choice for describing privacy policies, lacks of a formal semantics. In fact, due to this limitation, [90] uses the language Z [94] for specifying fine-grained access control policies. Although an interesting option, we prefer to use SecureUML instead of Z because SecureUML already has "built-in" the notions of role, permission, methods, resources, and authorization constraints, which, would have to be "encoded" (more or less, naturally) along with the policies, if we were to use Z. Furthermore, SecureUML is designed to support model-driven security [12].

4.5 Checking data invariants preservation

Data-management applications are focused around so-called CRUD actions that create, read, update, and delete data from persistent storage. These operations are the building blocks for numerous applications, for example dynamic websites where users create accounts, store and update information, and receive customized views based on their stored data. Typically, the application's data is required to satisfy some properties, which we may call the application's data invariants.

We introduce here a tool-supported, model-based methodology for proving that all the actions possibly triggered by a data-management application will indeed preserve the application's data invariants. Moreover, we report on our experience applying this

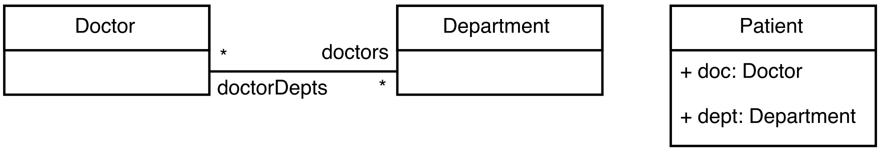


Figure 4.11: EHR: a sample data model.

methodology on the same application for managing medical records that we introduced in in Section 4.1.[5]

In a nutshell, our approach consists of the following three steps. Suppose that we are interested in checking whether a sequence $\mathcal{A} = \langle act_1, \ldots, act_{n-1}\rangle$ of data actions preserves an invariant ϕ of an application's data model $\mathcal{D}$. We proceed as follows: (Step 1) From the data model $\mathcal{D}$, we automatically generate a new data model $\mathrm{Film}(\mathcal{D}, n)$ for representing all sequences of n states of $\mathcal{D}$. Notice that some of these sequences will correspond to executions of $\mathcal{A}$, but many others will not. (Step 2) We constrain the model $\mathrm{Film}(\mathcal{D}, n)$ in such a way that it will represent exactly the sequences of states corresponding to executions of $\mathcal{A}$. We do so by adding to $\mathrm{Film}(\mathcal{D}, n)$ a set of constraints $\mathrm{Execute}(\mathcal{D}, act_i, i)$ capturing the execution of the action act_i upon the i-th state of a sequence of states, for $i = 1, \ldots, n-1$. (Step 3) We prove that, for every sequence of states represented by the model $\mathrm{Film}(\mathcal{D}, n)$ constrained by $\bigcup_{i=1}^{n-1} \mathrm{Execute}(\mathcal{D}, act_i, i)$, if the invariant ϕ is satisfied in the first state of the sequence then it is also satisfied in the last state of the sequence.

4.5.1 Modeling sequences of states

A data model provides a data-oriented view of a system, the idea being that each state of a system can be represented by an *instance* of the system's data model. Here we introduce a special data model: one whose instances do not represent states of a system but instead *sequences of states* of a system.

Example 30 *Consider the data model* EHR *shown in Figure 4.11. It consists of three classes:* Patient*,* Department*, and* Doctor*.*

Patient *It represents patients. The doctor treating a patient is set in the attribute* doc *and the department where a patient is treated is set in the attribute* dept*.*

Department *It represents departments. The doctors working in a department are linked to the department through the association-end* doctors*.*

Doctor *It represents doctor's information. Departments where a doctor works are linked to the doctor's information through the association-end* doctorDepts*.*

□

Example 31 *Consider the object model* Inst_EHR *shown in Figure 4.12. It represents an instance of the data model* EHR *shown in Figure 4.11. In particular,* Inst_EHR *represents a state of the system in which there are only two departments, namely,* Cardiology *and* Digestive*; one doctor, namely,* J_Smith*, working for both departments; and one patient,* M_Perez*, treated by doctor* J_Smith *in the department of* Cardiology*.* □

[5] For the sake of simplicity, however, the underlying data model in this case does not make use of class inheritance.

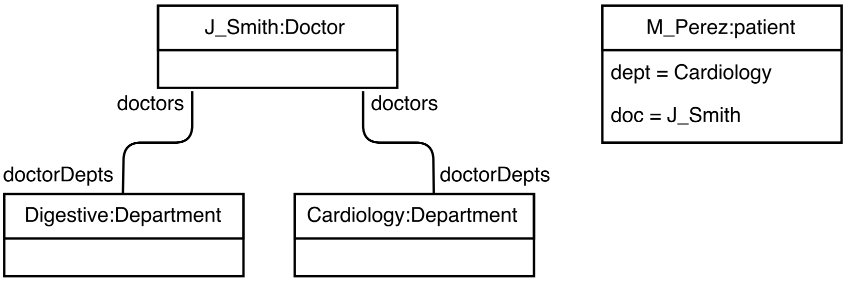


Figure 4.12: Inst_EHR: a sample object model.

Example 32 *Suppose that the following data invariants are specified for the data model* EHR *in Figure 4.11:*

1. Each patient is treated by a doctor.

   Patient.allInstances()->forAll(p|not(p.doc.oclIsUndefined()))

2. Each patient is treated in a department.

   Patient.allInstances()->forAll(p|not(p.dept.oclIsUndefined()))

3. Each patient is treated by a doctor who works in the department where the patient is treated.

   Patient.allInstances()->forAll(p|p.doc.doctorDepts->includes(p.dept))

Clearly, the object model Inst_EHR *in Figure 4.12 is a valid instance of* EHR *with respect to the data invariants (1)–(3), since they evaluate to* true *in* Inst_EHR*.* □

Next, we introduce the notion of *filmstrips* to model sequences of states of a system. Given a data model $\mathcal{D}$, a *$\mathcal{D}$-filmstrip model* of length n, denoted by $\mathrm{Film}(\mathcal{D}, n)$, is a new data model which contains the same classes as $\mathcal{D}$, but now:

- To represent that an object may have different attribute values and/or links in each state, each class c contains n different "copies" of each of the attributes and association-ends that c has in $\mathcal{D}$. The idea is that, in each instance of a filmstrip model, the value of the attribute *at* (respectively, association-end *as*) for an object o in the i-th state of the sequence of states modelled by this instance is precisely the value of the i-th "copy" of *at* (respectively, *as*).

- To represent that an object may exist in some states, but not in others, each class c contains n "copies" of a new boolean attribute `st`. The idea is that, in each instance of a filmstrip model, an object o exists in the i-th state of the sequence of states modelled by this instance if and only if the value of the i-th "copy" of `st` is `true`.

A formal definition of filmstrip models is given here:[6]

[6] For the sake of simplicity, we are not considering here multiplicities or generalizations.

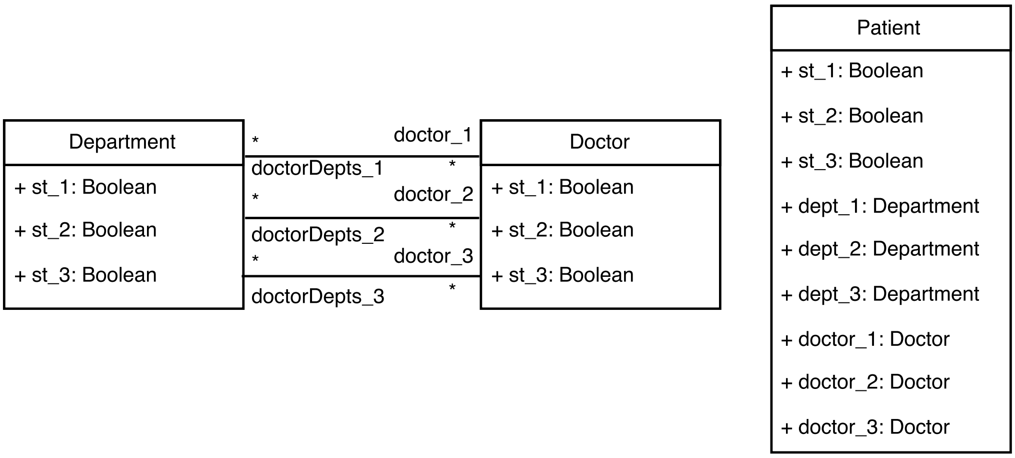


Figure 4.13: Film(EHR,3): a filmstrip model of length 3 of EHR.

Definition 8 (Filmstrip models) *Let $\mathcal{D}$ be a data model, $\mathcal{D} = \langle C, AT, AS, ASO\rangle$. Let n be a positive number. We denote by* $\mathrm{Film}(\mathcal{D}, n)$ *the model of the sequences of length n of $\mathcal{D}$-object models.* $\mathrm{Film}(\mathcal{D}, n)$ *is defined as follows:*

$$\mathrm{Film}(\mathcal{D}, n) = \langle C, (n\times\{\mathrm{st}\}) \cup (n\times AT), (n\times AS), (n\times ASO)\rangle$$

where

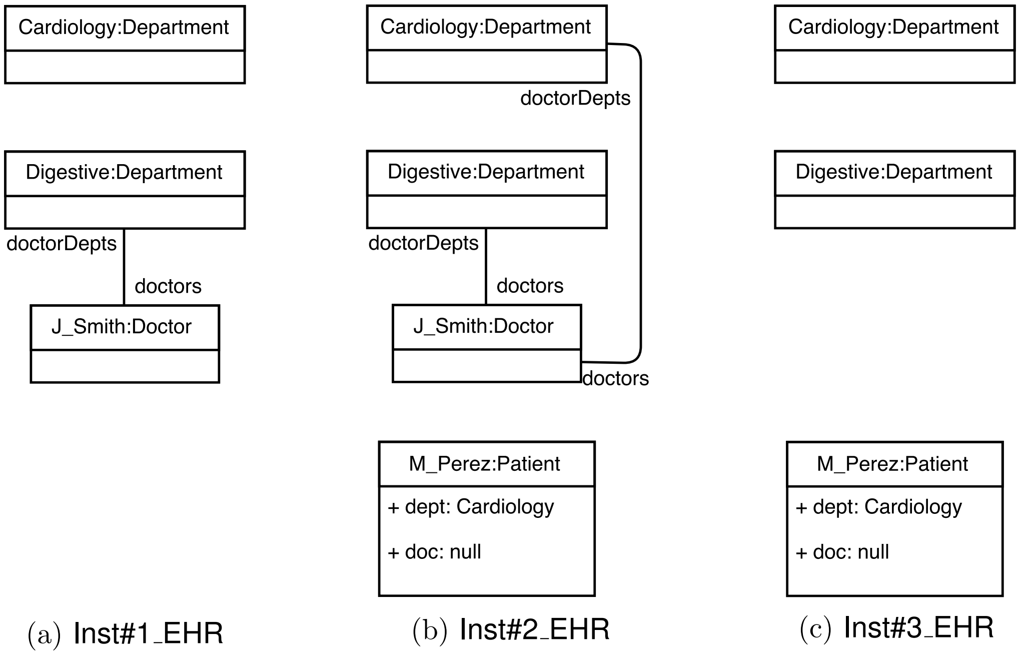
- $(n\times\{\mathrm{st}\}) = \{(\mathrm{st\_}i)_{(c,\mathsf{Boolean})} \mid c \in C \wedge 1 \leq i \leq n\}$.
- $(n\times AT) = \{(at\_i)_{(c,t)} \mid at_{(c,t)} \in AT \wedge 1\leq i\leq n\}$.
- $(n\times AS) = \{(as\_i)_{(c,c')} \mid as_{(c,c')} \in AS \wedge 1\leq i\leq n\}$.
- $(n \times ASO) = \{((as\_i)_{(c,c')}, (as'\_i)_{(c',c)}) \mid (as_{(c,c')}, as'_{(c',c)}) \in ASO \wedge 1\leq i\leq n\}$.

Example 33 *In Figure 4.13 we show the filmstrip model* Film(EHR, 3)*. Consider now the three instances of* EHR *shown in Figure 4.14. The first instance (*Inst#1_EHR*) corresponds to a state where there are two departments,* Cardiology *and* Digestive*, and one doctor,* J_Smith*, working in* Digestive*. The second instance (*Inst#2_EHR*) is like the first one, except that now* J_Smith *also works in* Cardiology *and, moreover, there is a patient,* M_Perez*, who is treated in* Cardiology*, but has no doctor assigned yet. Finally, the third instance (*Inst#3_EHR*) is like the second one, except that it does not contain any doctor. In Figure 4.15 we show how the sequence* ⟨Inst#1_EHR, Inst#2_EHR, Inst#3_EHR⟩ *can be represented as an instance of* Film(EHR, 3)*.* □

Finally, we introduce a function Project(), which we will use when reasoning about filmstrip models. Let $\mathcal{D}$ be a data model and let ϕ be an expression. $\mathrm{Project}(\mathcal{D}, \phi, i)$ "projects" the expression ϕ so as to refer to the i-th state in the sequences represented by the instances of $\mathrm{Film}(\mathcal{D}, n)$, for $n \geq i$.

A formal definition of Project() is given here:

Definition 9 (Project) *Let $\mathcal{D} = \langle C, AT, AS, ASO\rangle$ be a data model. Let n be positive number. Let ϕ be a $\mathcal{D}$-expression. For $1 \leq i \leq n$,* $\mathrm{Proj}(\mathcal{D}, \phi, i)$ *is the* $\mathrm{Film}(\mathcal{D}, n)$*-expression that "projects" the expression ϕ so as to refer to the i-th state in the sequences represented by the instances of* $\mathrm{Film}(\mathcal{D}, n)$*.* $\mathrm{Proj}(\mathcal{D}, \phi, i)$ *is obtained from ϕ by executing the following:*



(a) Inst#1_EHR (b) Inst#2_EHR (c) Inst#3_EHR

Figure 4.14: Three instances of EHR.

- *For every class $c \in C$, replace every occurrence of* c.allInstances() *by* c.allInstances() ->select(o|o.st_(i)).
- *For every attribute $at_{(c,t)} \in AT$, replace every occurrence of .at by .at_(i).*
- *For every link $as_{(c,c')} \in AS$, replace every occurrence of .as by .as_(i).*

Example 34 *Consider the data invariants (1) and (3) presented in the Example 32. Then,*

Project(EHR, (1), 1) =
Patient.allInstances()->select(p|p.st_1)->forAll(p|not(p.doc_1.oclIsUndefined()))
Project(EHR, (3), 1) =
Patient.allInstances()->select(p|p.st_1)
->forAll(p|p.doc_1.doctorDepts_1->includes(p.dept_1))

Recall that Patient.allInstances()->select(p|p.st_1) *refers to the instances of the entity* Patient *which exist in the first state of the sequences of states modelled by* Film(EHR, 3), *while* .doc_1 *and* .doctorDepts_1 *refer, respectively, to the value of the attribute* doc *and the links through the association-end* doctorDepts *of the instances of the entity* Patient *also in the first state of the aforementioned sequences of states.* □

4.5.2 Modeling sequences of data actions

As explained before, given a data model $\mathcal{D}$ and a positive number n, the instances of the filmstrip model Film($\mathcal{D}$, n) represent sequences of n states of the system. Notice, however, that, in the sequence of states represented by an instance of Film($\mathcal{D}$, n), the $(i + 1)$-th state does not need to be the result of executing an atomic data action upon the i-th state.

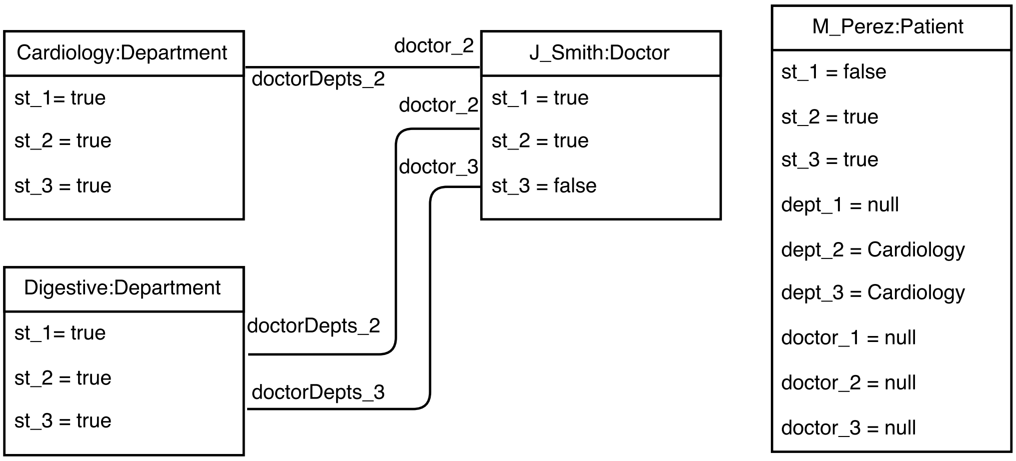


Figure 4.15: An instance of Film($\mathsf{EHR}, 3$).

Let $\mathcal{D}$ be a data model and let *act* be a CRUD data action. In this section we introduce a set of boolean OCL expressions, Execute($\mathcal{D}$, act, i), which capture the relations that hold between the i-th and (i+1)-th states of a sequence, if the latter is the result of executing the action *act* upon the former. We provide first the expressions that capture the differences between the two states, $(i + 1)$-th and i-th, and afterwards the expressions that capture their commonalities.

As expected, we define Execute($\mathcal{D}$, act, i) by cases. We consider the following *atomic data actions*: *create* or *delete* an object of an entity; *read* the value of an attribute of an object; and *add* or *remove* a link between two objects. [7]

Action create. For *act* the action of creating an instance *new* of an entity c, the difference between the states $(i + 1)$-th and i-th can be captured by the following expressions in Execute($\mathcal{D}$, act, i):

- $new.\mathsf{st}\_i = \mathsf{false}$.
- $new.\mathsf{st}\_(i + 1) = \mathsf{true}$.
- $new.at\_(i + 1) = \mathsf{null}$, for every attribute *at* of the entity c.
- $new.as\_(i + 1)\text{->}\mathsf{isEmpty()}$, for every association-end *as* of the entity c.

Action delete. For *act* the action of deleting an instance o of an entity c, the difference between the states $(i + 1)$-th and i-th can be captured by the following expressions in Execute($\mathcal{D}$, act, i):

- $o.\mathsf{st}\_i = \mathsf{true}$.
- $o.\mathsf{st}\_(i + 1) = \mathsf{false}$.
- $o.at\_(i + 1) = \mathsf{null}$, for every attribute *at* of the entity c.

[7] The tool supports also *conditional* data actions, where the conditions are boolean OCL expressions. Notice that, when *act* is a conditional data action, we must also include in Execute($\mathcal{D}$, act, i), the expression that results from "projecting" its condition, using the function Project(), so as to refer to the i-th state in the sequence.

- $o.as\_(i+1)$->isEmpty(), for every association-end as of the entity c.
- c'.allInstances().$as'\_(i+1)$->excludes(o) for every entity c', and every association-end as' between c' and c.

Action update. For act the action of updating an attribute at of an instance o of an entity c with a value v, the difference between the states $(i+1)$-th and i-th can be captured by the following expression in Execute($\mathcal{D}$, act, i):

- $o.at\_(i+1) = v$.

Action add. For act the action of adding an object o' to the objects that are linked with an object o through an association-end as (whose opposite association-end is as'), the difference between the states $(i+1)$-th and i-th can be captured by the following expressions in Execute($\mathcal{D}$, act, i):

- $o.as\_(i+1) = (o.as\_i)$->including(o').
- $o'.as'\_(i+1) = (o'.as'\_i)$->including($o$).

Action remove. For act the action of removing an object o' to the objects that are linked with an object o through an association-end as (whose opposite association-end is as'), the difference between the states $(i+1)$-th and i-th can be captured by the following expressions in Execute($\mathcal{D}$, act, i):

- $o.as\_(i+1) = (o.as\_i)$->excluding(o').
- $o'.as'\_(i+1) = (o'.as'\_i)$->excluding($o$).

Finally, we list below the expressions in Execute($\mathcal{D}$, act, i) that capture the commonalities between the states $(i+1)$-th and i-th, for the case of the action updating an attribute at of an instance o of and entity c; the expressions for the other cases are entirely similar.

- d.allInstances()->select(x|x.st_$(i+1)$) = d.allInstances()->select(x|x.st_i), for every entity d.
- d.allInstances()->select(x|x.st_i)->forAll(x|x.$at'\_(i+1)$ = x.at'_i), for every entity d and every attribute at' of d, such that $at' \neq at$.
- c.allInstances()->select(x|x.st_i)->excluding(o)->forAll(x|x.$at\_(i+1)$ = x.at_i).
- d.allInstances()->select(x|x.st_i)->forAll(x|x.$as\_(i+1)$ = x.as_i) for every entity d, and every association-end as of d.

4.5.3 Checking data invariants preservation

Invariants are properties that are *required* to be satisfied in every system state. Recall that, in the case of data-management applications, the system states are the states of the applications' persistence layer, which can only be changed by executing the sequences of data actions associated to the applications' GUI events. We can now formally define the invariant-preservation property as follows:

Definition 10 (Invariant preservation) *Let $\mathcal{D}$ be a data model, with invariants Φ. Let $\mathcal{A} = \langle act_1, \ldots, act_{n-1}\rangle$ be a sequence of data actions. We say that $\mathcal{A}$* preserves *an invariant $\phi \in \Phi$ if and only if*

$$\forall \mathcal{F} \in [\![\mathrm{Film}(\mathcal{D}, n), \bigcup_{i=1}^{n-1} \mathrm{Execute}(\mathcal{D}, act_i, i)]\!] . \quad [\![\mathrm{Project}(\mathcal{D}, \bigwedge_{\psi\in\Phi} (\psi), 1) \ \textsf{implies}\ \mathrm{Project}(\mathcal{D}, \phi, n)]\!]^{\mathcal{F}} = \mathrm{true}, \tag{4.1}$$

*i.e., if and only if, for every $\mathcal{A}$-*valid *instance $\mathcal{F}$ of* Film$(\mathcal{D}, n)$ *the following holds: if all the invariants in Φ evaluate to* true *when "projected" over the first state of the sequence of states represented by $\mathcal{F}$, then the invariant ϕ evaluates to* true *as well when "projected" over the last state of the aforementioned sequence.*

By Remark 1, we can reformulate Definition 10 as follows: Let $\mathcal{D}$ be a data model, with invariants Φ. Let $\mathcal{A} = \langle act_1, \ldots, act_{n-1}\rangle$ be a sequence of data actions. We say that $\mathcal{A}$ *preserves* an invariant $\phi \in \Phi$ if and only if the following set is unsatisfiable:

$$\mathrm{o2f_{data}}(\mathrm{Film}(\mathcal{D}, n)) \cup \{\mathrm{o2f_{true}}(\gamma) \mid \gamma \in \bigcup_{i=1}^{n-1} \mathrm{Execute}(\mathcal{D}, act_i, i)\} \cup \mathrm{o2f_{true}}(\textsf{not}(\mathrm{Project}(\mathcal{D}, \bigwedge_{\psi\in\Phi} (\psi), 1) \ \textsf{implies}\ \mathrm{Project}(\mathcal{D}, \phi, n))). \tag{4.2}$$

In other words, using our mapping from OCL to first-order logic, we can transform an invariant-preservation problem (4.1) into a first-order satisfiability problem (4.2). And by doing so, we open up the possibility of using SMT solvers to automatically (and effectively) check the invariant-preservation property of non-trivial data-management applications, as we will report in the next section.

Case Study

In this section we report on a case study about using SMT solvers —in particular, Z3 [30]— for proving the invariant-preservation property. All the proofs have been ran on a machine with an Intel Core2 processor running at 2.83 GHz with 8GB of RAM, using Z3 versions 4.3.1 and 4.3.2. The Z3 input files are available at [96] where we also indicate which files are to be ran with which version.

The data-management application for this case study is the eHealth Record Management System (EHRM) developed, using ActionGUI, within the European Network of Excellence on Engineering Secure Future Internet Software Services and Systems (NESSoS) [62]. The EHRM application consists of a web-based system for electronic health record management. The data model contains 18 entities, 40 attributes, and 48 association-ends. It also contains 86 data invariants. For the sake of illustration, we can group the EHRM's data invariants in the following categories:

G1. Properties about the population of certain entities. E.g., *There must be at least a medical center.*

```
MedicalCenter.allInstances()->notEmpty().
```

Sequences	Acts.	Conds.	Invariants			Time		
			affected	preserved	violated	min.	max.	avge.
Create an administrative	8	9	18	18	0	0.03s	0.20s	0.05s
Create a nurse	10	11	22	22	0	0.03s	0.22s	0.06s
Create a doctor	11	12	25	24	1	0.03s	27.00s	0.07s
Reassign a doctor	2	6	2	2	0	6.88s	11.10s	8.94s
Reassign a nurse	2	6	2	1	1	0.10s	17.01s	8.55s
Register patient	30	6	28	26	2	0.03s	0.20s	0.05s
Move a patient	2	3	3	3	0	0.03s	0.03s	0.03s
		Total	100	96	4			

Figure 4.16: EHRM case study: summary.

G2. Properties about the definedness of certain attributes. E.g., *The name of a professional cannot be left undefined.*

```
Professional.allInstances()->forAll(p|not(p.name.oclIsUndefined())).
```

G3. Properties about the uniqueness of certain data. E.g.: *There cannot be two different doctors with the same licence number.*

```
Doctor.allInstances()->forAll(d1,d2|d1<>d2 implies d1.licence<>d2.licence).
```

G4. Properties about the population of certain association-ends. E.g., *Every medical center should have at least one employee.*

```
MedicalCenter.allInstances()->forAll(m|m.employees->notEmpty()).
```

G5. Other properties: E.g., *A patient should be treated in a department where its assigned doctor works.*

```
Patient.allInstances()
  ->forAll(p|p.doctor.doctorDepartments->includes(p.department)).
```

In our case study, we have checked the invariant-preservation property for seven non-trivial sequences of data actions: namely, those that create a new admin staff, a new nurse, or a new doctor; those that reassign a doctor or a nurse to another department; and those that register a new patient, and move a patient to a different ward. The result of our case study is shown in Figure 4.16. In particular, for each of the aforementioned sequences of actions, we indicate:

- The number of data actions (and conditions) in the sequence.

- The number of data invariants (potentially) affected by the actions in the sequence, indicating how many of them we have proved to be preserved by the sequence and how many to be violated.[8]
- The minimum, maximum, and average time taken for proving that the sequence preserves (or violates) each of the (potentially) affected invariants.

4.5.4 Concluding remarks

There are two main lessons that we can learn from this case study. The first lesson is that, when modelling non-trivial data-management applications, it is indeed not difficult to make errors, or at least omissions. In fact, the four *violated* invariants showed in Figure 4.16 arise because the EHRM's modeler inadvertently omitted some conditions for the execution of the corresponding sequence of actions. As an example, for the case of creating a doctor, the invariant that is *violated* is "Every doctor has a unique licence number", and it is so because the modeler omitted a condition for checking that the licence number of the doctor to be created must be different from the licence numbers of the doctors already in the system. As another example, for the case of reassigning a nurse, the invariant that is *violated* is "There should be at least one nurse assigned to each department", and this is produced because the modeler omitted a condition for checking that the department where the nurse to be reassigned currently works must have at least two nurses working in it.

The second lesson that we have learned is that, using our methodology, and, in particular, using Z3 as the back-end prover, the invariant-preservation property can indeed be effectively checked for non-trivial data-management applications. As reported in Figure 4.16, we are able to automatically prove that, for each of the sequences of actions under consideration, all the *affected* invariants are either *preserved* or *violated*. This means that Z3 does not return "unknown" for any of the 100 checks that we have to perform (corresponding to the total number of *affected* invariants), despite the fact that in all these checks there are (many) quantifiers involved. Moreover, regarding performance, Figure 4.16 shows that, in most of the cases we are able to prove the invariant-preservation property in less than 100ms (worst case: 27s). This great performance is achieved even though, for each case, Z3 needs to check the satisfiability of a first-order theory containing on average 190 declarations (of function, predicate and constant symbols), 20 definitions (of predicates), and 550 assertions. Overall, these results improve very significantly those obtained in a preliminary, more simple case study reported in [29], where some checks failed to terminate after several days, and some others took minutes before returning an answer. However, we should take these new results with a grain of salt. Indeed, we are very much aware (even painfully so) that our current results depend on the (hard-won) interaction between (i) the way we formalize sequences of n states, OCL invariants, actions' conditions, and actions' executions, and (ii) the heuristics implemented in the verification back-end we use, namely Z3. This state-of-affairs is very well illustrated by the fact that, as indicated before, we have had to use two different versions of Z3 (4.3.1 and 4.3.2) to complete our case study for the following reason: there are some checks for which one of the versions returns "unknown", while the other version returns either "sat" or "unsat"; but there are some other checks for which precisely the opposite occurs.

[8] Interestingly, when an invariant is violated, Z3 returns also an instance of the given filmstrip model responsibly for this violation. This *counterexample* can then be used to fix accordingly the given sequence of actions.

Chapter 5

Related work

¡UNA TUERCA!
¡OXIDADA! ¿PARA QUÉ DEMONIOS SIRVE?
TODO SIRVE PARA ALGO
PERO NADA SIRVE PARA TODO

© Joaquín S. Lavado, QUINO. Toda Mafalda, Penguin Random House, España

To facilitate the presentation, we will divide this chapter into four sections. First, we will discuss work related to our mapping from OCL queries to the procedural language extensions of SQL (SQL-PL). Secondly, we will discuss work related to our mapping from OCL constraints to many-sorted first-order logic. Finally, we will discuss work related to two of our applications domains, namely, analyzing security models and checking data invariants preservation. (The related work for the other three applications domains is less substantial and was already discussed when presenting the corresponding domains.)

5.1 Mapping OCL as a query language

There have been several interesting attempts to map OCL into SQL, each with its own limitations and shortcomings. [91] supports only OCL class invariants and, partially, the operators forAll, select, and exists. [92] generates SQL code from OCL as a part of Enterprise Architect, but only for simple expressions; in particular, it cannot with deal with OCL iterator expressions or sequences. [7] explores a model transformation approach from UML to CWM and from OCL to a patterns metamodel, but the idea has not been further developed. [19] introduces a different strategy for query translation. Instead of a compile time translation, it proposes a runtime query translation from model level languages like EOL, to persistent query languages like SQL. Each EOL query is splitted up into subexpressions that are handled by the appropriate implementation classes. The idea has not yet applied for translating OCL to SQL. [14] explores how participation constraints defined on binary associations, e.g. ‘xor’ constraint, can be expressed at two different levels, in OCL as a constraint language and as triggers in SQL. No mapping from

Level 1	size, +, -, *, max, min, -(unary), abs, floor, round, size, and, or, xor, implies, not, <=, <,>, <>, concat.
Level 2	mod y div.
Level 3	toInteger, toReal y toString.
Level 4	toBoolean, toUpper, toLower, substring.

Table 5.1: Support of OCL2SQL for primitive operators

OCL to SQL expressions has been proposed yet. [21] proposes OCL transformations rules to SQL for some simple OCL expressions. Complex expressions are not covered yet. [73] proposes a translation from OCL to a logic called Event-Dependency Constraints (EDC). From EDC it generates SQL statements with a pattern-based approach. No information is provided regarding the subset of the OCL language supported by the translation.

The most interesting comparison can be done, however, with the mapping OCL2SQL presented in [87, 31, 32]. OCL2SQL only supports boolean OCL expressions. The main idea behind OCL operators is the following: given a context, all elements of the query that do not satisfy the defined condition are returned. The tansformation rules are described in [87]. Basically, it use a generic OCL pattern to map the expression into a generic MySQL pattern.

- OCL pattern
  ```
  context: Class
  inv: OCL_boolean_expression
  ```

- MySQL pattern
  ```
  select *
  from Class
  where not OCL2SQL(OCL_boolean_expression)
  ```

The mapping proposed there does not faithfully represent some key properties of the evaluation semantics of OCL. In particular, by relying on the SQL `in` operator, it erroneously removes duplicates from bag-collections. In our case, to preserve the evaluation semantics of OCL, we use SQL *left joins* instead of the `in` operator.

In particular, in the implementation of the OCL2SQL tool, we have found we have found that there are for level of supported OCL operators:

- Level 1: are supported by the OCL2SQL parser and the code generated is correct with respect to the semantics of the OCL operators,

- Level 2: are supported by the OCL2SQL parser and the code generated is not correct with respect to the semantics of the OCL operators, or it is not syntactically correct in MySQL,

- Level 3: are supported by the OCL2SQL parser but no code is generated,

- Level 4: are not supported by the OCL2SQL parser.

Tables 5.1 and 5.2 summarize the operators over primitive types and the operators on sets and multi-sets that are currently in each level.

Level 1	includes, excludes, isEmpty, notEmpty.
Level 2	excludesAll, includesAll, symmetricDifference, intersection, union(set,bag.), =(set,bag.).
Level 3	
Level 4	including y excluding

Table 5.2: Support of OCL2FOL for operators over collections

Sorted sets and sequence types are not supported by OCL2SQL. With respect to the iterators, we have noticed that there are some implementation problems: for example, they do not generate code for source–>collect(p|p) or source–>collect(p|p.atr) expressions. Also, if the body of an iterator contains a comparison involving the iterator variable, and that variable occurs on the right side of the comparison, only an incomplete query is obtained.

To the best of our knowledge the idea of mapping OCL iterators to stored procedures was first proposed in [87], but it was not developed afterwards.[1].

Finally, [22] seems to be the only work dealing with the translation from SQL to OCL up to date. It is motivated by the concern of expressing database integrity constraints as business rules in a more abstract language. In the process of business rules identification, it describes the mapping between SQL SELECT statements and certain type of PL blocks, and the equivalent OCL expressions. The mapping focuses in handling SQL projections, joins, conditions, functions, group by and having clauses.

5.2 Mapping OCL as a constraint language

With the goal of providing support for UML/OCL reasoning, different mappings from OCL to other formalisms have been proposed in the past. In each case, the chosen target formalism imposes a different trade-off between expressiveness, termination, automation, and completeness. In particular, most proposals have disregarded OCL undefinedness in order to more easily map OCL to a two-valued formalism.

In Table 5.3 we summarize the different mappings from UML/OCL to other formalism. They are grouped as follows. The first group (G1) includes mappings that *do not not support OCL constraints.* FiniteSAT [56] uses constrained generalization sets for reasoning about finite satisfiability of UML class diagrams. DL [8] encodes the problem of finite model reasoning in UML classes as a constraint satisfaction problem (CSP). MathForm [95] formalizes UML class diagrams using set and partial functions.

The second group (G2) includes mappings that *support OCL constraints, but that do not consider OCL undefinedness.* UMLtoCSP [17] translates UML class diagrams and OCL constraints into CSP. EMFtoCSP [41] is an evolution of UMLtoCSP, which supports EMF models; AuRUS [78, 85] supports verifying and validating UML/OCL conceptual

[1] "Das Ergebnis des hier vorgestellten Abbildungsmusters kann für einen Teilausdruck nicht direkt in das Abbildungsergebnis eines anderen Teilausdrucks eingesetzt werden. Die Kombinationstechnik wird nicht formal beschrieben." [87, pag.59] [. . .] "Es ist in dieser Arbeit nicht gelungen, eine übersichtliche und vollständig formale Darstellung für die prozeduralen Abbildungsmuster zu finden." [87, pag.112].

In our own translation: "The result of the mapping model presented here may not apply a part of the expression directly into the result of another subexpression.The combination technique is not formally described." [. . .] "This work did not succeed to find a concise and complete formal representation for procedural mapping patterns."

	Mapping	**Target formalism**
G1	FiniteSAT [56]	System of Linear Inequalities
	DL [8]	Description Logics, CSP
	MathForm [95]	Mathematical Notation
G2	UMLtoCSP [17]	CSP
	EMFtoCSP [41]	CSP
	AuRUS [78]	FOL
	OCL2FOL [20]	FOL
	OCL-Lite [77]	Description Logics
	BV-SAT [93]	Relation Logic
	PVS [79]	HOL
	CDOCL-HOL [3]	HOL
	KeY [2]	Dynamic Logic
	Object-Z [82]	Object-Z
	UML-B [57]	B
G3	UML2Alloy [4]	Relation Logic
	USE [39]	Relation Logic
G4	HOL-OCL [16]	HOL
	OCL2FOL$^+$ [26]	FOL

Table 5.3: Other mappings from UML/OCL to other formalism.

schemes using first-order logic; OCL2FOL [20] also maps UML/OCL class diagrams to first-order logic. OCL-Lite [77] maps a fragment of OCL to DL, ensuring termination. BV-SAT [93] encodes UML/OCL into bit vectors, and solves UML/OCL verification problems based on Boolean satisfiability. PVS [79] and CDOCL-HOL [3] uses higher-order logic: in particular, they map UML/OCL to the specification languages of the theorem provers PVS and Isabelle, respectively. KeY [2] uses dynamic logic, a multi modal extension of first-order logic; Object-Z [82] maps UML/OCL into Object-Z; and finally UML-B [57] maps UML/OCL to the B formal specification.

The third group (G3) includes mappings that *support OCL constraints and consider* null*-related undefinedness, but not* invalid*-related undefinedness.* UML2Alloy [4] and USE [39, 51] map UML/OCL to relational logic and use the SAT-based constraint solver KodKod for solving UML/OCL verification problems.

The fourth group (G4) includes mappings that *support OCL constraints and OCL undefinedness.* OCL2MSFOL belongs to this group. OCL-HOL [16] embeds UML/OCL in the specification language of the interactive theorem provers Isabelle. It supports the full OCL language, but it requires advanced user interaction to solve UML/OCL verification problems. OCL2FOL$^+$ [26] maps UML/OCL to first-order logic and uses SMT solvers to attempt to solve automatically UML/OCL verification problems.

Next, we compare more closely OCL2MSFOL with the mappings in groups G3 and G4, and, in particular, with USE and HOL-OCL, from a *practical* point of view.

With regard to HOL-OCL, there are two significant differences. On the one hand, HOL-OCL uses an interactive theorem prover, Isabelle, for UML/OCL reasoning, which requires advanced knowledge (and possibly time) from the part of the user. OCL2MSFOL, on the contrary, uses SMT solvers with finite model finding capabilities, which, as we have shown, efficiently support automated UML/OCL reasoning. On the other hand, HOL-OCL supports the full OCL language (in fact, it can be considered as providing 'de facto' formal semantics for OCL), while OCL2MSFOL has a number of limitations, as we have discussed before, in supporting the OCL language.

With regards to USE, the difference, from a practical point of view, is that its mapping is designed for using SAT-based constraint solvers, while ours targets SMT-solvers. In practice, this means that while USE-based proofs are only valid for instances *up to a given size*, our proofs are valid for *all possible instances*. This is indeed the key advantage of using SMT-solvers instead of SAT-based constraint solvers for reasoning about UML/OCL models.

5.3 Analizing security models

Many proposals exist for reasoning about RBAC policies, each one using a different logic or formalism, including the so-called "default" logic [99], modal logic [59], higher-order logic [5], C-Datalog [9], first-order logic [49, 15], and description logic [100]. To the best of our knowledge none of these proposals has been properly extended to cope with fine-grained access control (FGAC) policies. In recent years, however, there has been a growing interest in finding appropriate formalisms and frameworks for specifying and analysing FGAC policies. [42] have proposed an interesting framework for specifying and reasoning about FGAC policies, called Lithium. It is based on a decidable fragment of (multi-sorted) first-order logic. Differently from OCL, this logic does not consider undefined values, which, based on our experience, is something crucial when formalizing properties of the system states. We are not aware of case studies that have been carried out using Lithium. [52, 53] propose a domain-specific language for specifying role-based policies which is based on UML and OCL. For the purpose of analyzing these policies, they propose to use SAT solvers, and, in particular the one implemented in Alloy [47]. Differently from SMT solvers, Alloy requires the search space to be bounded. In the context of XACML [65], there exists a XACML profile for the specification of RBAC policies [64]. However, no methods have been proposed for reasoning about policies written with this profile. Also, it is unclear whether this profile can be extended to cope with fine-grained access control policies. To address the first concern, [44] propose an extension of the XACML profile for RBAC based on OWL. This approach supports the use of an OWL-DL reasoner for deciding about RBAC policies within XACML. More interestingly, [80] have recently proposed a new syntax and semantics for XACML, for the purpose of supporting formal reasoning about XACML policies. One of the challenges here is to formalize the different algorithms for enforcing policy rules which are available in XACML. [80] formalize the majority of these algorithms, and propose two new algorithms (one of which is very close to the semantics of SecureUML.) Another challenge is to formalize the concepts of obligations and advices in XACML, but they are not covered by [80]. Finally, with respect to methods for reasoning about XACML policies, [80] propose to explore the use of SMT solvers, but no experiments are reported yet.

In a nutshell, our proposal differs from other approaches in that: (i) we use SecureUML for modeling FGAC policies, and (ii) we use a mapping from OCL to first-order for reasoning about these policies. In our opinion, our approach has two main advantages: (i) the reasoning about FGAC policies can take into account the properties of the system states, since OCL is the language that we use both for specifying the invariants in the data model and the authorization constraints in the security model; and (ii) the reasoning about FGAC policies can be done automatically (although sometimes may fail to find a result), since the mapping that we use for translating OCL into first-order logic supports the effective application of SMT solvers over the generated formulas.

5.4 Checking data invariants preservation

In the past decade, there has been a plethora of proposals for model-based reasoning about the different aspects of a software system. For the case of the static or structural aspects of a system, the challenge lies in mapping the system's data model, along with its data invariants, into a formalism for which reasoning tools may be readily available. On the other hand, for the case of model-based reasoning about a system's dynamic aspects the main challenge lies in finding a suitable formalism in which to map the models specifying how the system can change over time. To this extent, it is worthwhile noticing the different attempts made so far to extend OCL with temporal features (see [50] and references). In our case, however, we follow a different line of work, one that is centered around the notion of *filmstrips* [33, 97]. A filmstrip is, ultimately, a way of encoding a sequence of snapshots of a system. Interestingly, when this encoding uses the same language employed for modelling the static aspects of a system, then the tools available for reasoning about the latter can be used for reasoning about the former. This is precisely our approach, as well as the one underlying the proposals presented in [48] and [40]. However, the difference between our approach and those are equally important. It has its roots in our different way of mapping data models and data invariants (OCL) into first-order logic [20, 26], which allows us to effectively use SMT solvers for reasoning about them, while [40] and [48] resort to SAT solvers. As a consequence, when successful, we are able to prove that all possible executions of a given sequence of data actions preserve a given data invariant. On the contrary, [40] can only validate that a given execution preserves a given invariant, while [48] can prove that all possible executions of a given sequence of data action preserve a given invariant, but only if these executions do not involve more than a given number of objects and links. Finally, [93] proposes also the use of filmstrip models and SMT solvers for model-based reasoning about the dynamic aspects of a system. This proposal, however, at least in its current form, lacks too many details (including non-trivial examples) for us to be able to provide a fair comparison with our approach.

Chapter 6

Conclusions and future work

Y EL QUE NO HAYA ENTENDIDO, QUE LEVANTE LA MANO
VEAMOS, MANOLITO, ¿QUÉ ES LO QUE NO HAS ENTENDIDO?
DESDE MARZO HASTA AHORA, ¡NADA!
QUINO

© Joaquín S. Lavado, QUINO. Toda Mafalda, Penguin Random House, España

In the work presented here we have defined two key, novel mappings for dealing with UML models (or UML-like models) that use OCL as a constraint and query language. Moreover, we have discussed the applicability and benefits of our mappings with a number of non-trivial benchmarks and case studies.

The first mapping we have introduced is a code-generator from OCL queries to the procedural language extensions of SQL (SQL-PL), which generates queries that can be efficiently executed in the target language. Our mapping follows the seminal ideas presented in our previous work [34, 25], with three substantial changes that greatly increase its applicability:

- Each OCL expression is mapped to a single stored procedure, which can afterwards be executed by a single call-statement.
- When needed, temporary tables are used within store procedures to hold intermediate values.
- The three-valued evaluation semantics of OCL is considered.

Moreover, the definition of our mapping makes it now easier to target different relational database management systems, both open source and proprietary.

The second mapping we have presented here is a translation from OCL constraints to many-sorted first-order logic, which generates logical expressions whose satisfiability can be efficiently checked using Satisfiability Module Theories (SMT) solvers. This mapping follows seminal ideas presented in our previous work [20, 26], but successfully overcomes the limitarjtions encountered in our previous proposals. First, it accepts as input a significantly larger subset of the UML/OCL language; in particular, it supports UML generalization, along with the generalization-related OCL operators. Secondly, it generates as output a class of satisfiability problems that are amenable to checking by using SMT solvers with finite model finding capabilities. This second point has proven to be key in practice, since *pure* SMT solvers would often return unknown when used along our mappings, as a consequence of two facts: first, that non-trivial OCL constraints contain expressions that are naturally mapped to quantified formulas (since they refer to all the objects in a class, for example), and, secondly, that techniques for dealing with quantified formulas in SMT are generally incomplete.

There are many interesting lines of work that we would like to pursue from here.

As for our mapping from OCL to SQL-PL, we plan to integrate it with CASE tools supporting the development life-cycle in the context of UML-like models, as to ease the work of developers and architects. Moreover, we plan to use our mapping as a starting point to address the backward traceability from SQL to OCL, which has been hardly studied so far. Finally, we plan to study the feasibility of mapping OCL to NoSQL databases. Yet, we are aware of the difficulty of such a mapping, given the lack of a common standard among the different NoSQL databases.

Regarding our mapping from OCL to many-sorted first-order logic, we would like to emphasize that our results ultimately depend on the (hard-won) positive logical interaction between (i) our formalization of UML/OCL in MSFOL, and (ii) the heuristics implemented in the SMT solver. This means, in particular, that changes in the SMT solver's heuristics may have consequences (hopefully positive) in the applicability of our mapping. It also means that a deeper understanding from our part of the SMT solver's heuristics may lead us to redefine our mapping in the future. As for the current limitations of our mapping from OCL to many-sorted first-order logic, we would like to comment the following. The key limitation of OCL2MSFOL comes from the fact that expressions defining collections are mapped, as we have explained, to predicates. Although these new predicates are defined so as to capture the property that distinguishes the elements belonging to the given collection, this is not sufficient for reasoning about the *size* of this collection, or about the *multiplicity* or the *ordering* of its elements. Because of this, OCL2MSFOL cannot support, in general, size-expressions or expressions of collection types different from set types. Fortunately, we are not finding this limitation hindering the applicability of our mapping. Other limitations of OCL2MSFOL are mostly due to time constraints, and will be soon corrected, including the current lack of support for attributes of type Boolean and for multiplicities of the form $[n..m]$, where n, m are natural numbers. In the first case, the corresponding terms t of type Boolean would be replaced by formulas of the form $t = \top$. In the second case, the data model would be extended with the corresponding invariants. Notice that it is also fairly trivial to extend our mapping to support n-ary associations.

Finally, in the area of application domains, we plan to define, following our methodology for analyzing security policies, formal mappings between the FGAC languages and frameworks supported by commercial DBMS (e.g., Oracle, IBM/DB2, Microsoft SQL Server and Teradata) and SecureUML. These mappings will allow us to apply our methodology also when reasoning about FGAC policies in commercial DBMS. Also, building upon our methodology for analyzing privacy policies, we envision the design and development of new, more powerful privacy tools which, as requested by the DPC audit to Facebook[45], will provide an "enhanced ability for users to make their own informed choices based on the available information". These tools will help to carry out, among other things, the rigorous comparisons between privacy policies of different social networking sites. Finally, we plan to extend our methodology for checking data invariants preservation to deal also with complex, non-atomic data action. The idea is to model the execution of these complex actions using OCL, as we have done for the case of CRUD actions. A more challenging goal, however, is to extend our methodology to deal with *iterations* over collection of data elements. The idea here is to integrate in our methodology the notion of *iteration invariant*, taking advantage of the fact that the collection over which the sequence of data actions must be iterated can be also specified using OCL.

Bibliography

[1] ActionGUI Project, 2016. http://www.actiongui.org/.

[2] W. Ahrendt, T. Baar, B. Beckert, M. Giese, E. Habermalz, R. Hähnle, W. Menzel, W. Mostowski, and P. H. Schmitt. The KeY system: Integrating object-oriented design and formal methods. In *FASE 2002, Grenoble, France, Proceedings*, volume 2306 of *LNCS*, pages 327–330. Springer, 2002.

[3] T. Ali, M. Nauman, and M. Alam. An accessible formal specification of the UML and OCL meta-model in isabelle/HOL. In *Multitopic Conference, 2007. INMIC 2007. IEEE.*, pages 1–6. IEEE, 2007.

[4] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. UML2Alloy: A challenging model transformation. In *MoDELS 2007, Nashville, USA, Proceedings*, volume 4735 of *LNCS*, pages 436–450. Springer, 2007.

[5] A. W. Appel and E. W. Felten. Proof-carrying authentication. In J. Motiwalla and G. Tsudik, editors, *ACM Conference on Computer and Communications Security*, pages 52–62. ACM, 1999.

[6] M. Arjona, R. Harjani, A. Muoz, and A. Maa. An Engineering Process to Address Security Challenges in Cloud Computing. In *3rd ASE International Conference on Cyber Security*, 2014.

[7] A. Armonas and L. Nemuraitė. Pattern Based Generation of Full-Fledged Relational Schemas From UML/OCL Models. *Information Technology and Control*, 35(1), 2006.

[8] A. Artale, D. Calvanese, and Y. A. Ibáñez-García. Checking full satisfiability of conceptual models. In *DL 2010, Waterloo, Ontario, Canada*, volume 573 of *CEUR Workshop Proceedings*, 2010.

[9] J. Bacon, K. Moody, and W. Yao. A model of OASIS role-based access control and its support for active security. *ACM Trans. Inf. Syst. Secur.*, 5(4):492–540, 2002.

[10] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *LNCS*, pages 171–177. Springer, 2011.

[11] D. Basin, J. Doser, and T. Lodderstedt. Model driven security: from UML models to access control infrastructures. *ACM Trans. on Software Engineering and Methodology*, 15(1):39–91, 2006.

[12] D. A. Basin, M. Clavel, and M. Egea. A decade of model-driven security. In R. Breu, J. Crampton, and J. Lobo, editors, *SACMAT*, pages 1–10. ACM, 2011.

[13] D. A. Basin, M. Clavel, M. Egea, M. A. G. de Dios, and C. Dania. A model-driven methodology for developing secure data-management applications. *IEEE Trans. on Software Engineering*, 40(4):324–337, 2014.

[14] D. Berrabah and F. Boufarès. Constraints checking in UML class diagrams: SQL vs OCL. In R. Wagner, N. Revell, and G. Pernul, editors, *Database and Expert Systems Applications, 18th International Conference, DEXA 2007, Regensburg, Germany, September 3-7, 2007, Proceedings*, volume 4653 of *LNCS*, pages 593–602. Springer, 2007.

[15] E. Bertino, B. Catania, E. Ferrari, and P. Perlasca. A logical framework for reasoning about access control models. *ACM Trans. Inf. Syst. Secur.*, 6(1):71–127, 2003.

[16] A. D. Brucker and B. Wolff. HOL-OCL: A formal proof environment for UML/OCL. In *FASE 2008, Budapest, Hungary. Proceedings*, volume 4961 of *LNCS*, pages 97–100. Springer, 2008.

[17] J. Cabot, R. Clarisó, and D. Riera. UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In *Conference on ASE 2007, Atlanta, Georgia, USA*, pages 547–548. ACM, 2007.

[18] D. Calvanese, S. Hartmann, and E. Teniente. Automated Reasoning on Conceptual Schemas (Dagstuhl Seminar 13211). *Dagstuhl Reports*, 3(5):43–77, 2013.

[19] X. D. Carlos, G. Sagardui, and S. Trujillo. MQT, an approach for run-time query translation: From EOL to SQL. In A. D. Brucker, C. Dania, G. Georg, and M. Gogolla, editors, *Proceedings of the 14th International Workshop on OCL and Textual Modelling co-located with 17th International Conference on Model Driven Engineering Languages and Systems (MODELS 2014), Valencia, Spain, September 30, 2014.*, volume 1285 of *CEUR Workshop Proceedings*, pages 13–22. CEUR-WS.org, 2014.

[20] M. Clavel, M. Egea, and M. A. G. de Dios. Checking unsatisfiability for OCL constraints. *Electronic Communications of the EASST*, 24:1–13, 2009.

[21] S. C. Cortázar. Transformación de las restricciones OCL de un esquema UML a consultas de SQL. trabajo de fin de grado. Technical report, Universidad Carlos III de Madrid, 2012. http://e-archivo.uc3m.es/bitstream/handle/10016/16799/TFG_Sergio_Casillas_Cortazar.pdf?sequence=1&isAllowed=y.

[22] V. Cosentino. *A model-based approach for extracting business rules out of legacy information systems.* PhD thesis, École des mines de Nantes, France, 2013.

[23] CUMULUS Project. http://cumulus-project.eu/.

[24] D4.2: Tools supporting CUMULUS-aware engineering process v1. http://cumulus-project.eu/index.php/public-deliverables.

[25] C. Dania. MySQL4OCL: Un compilador de OCL a MySQL, 2011. Master thesis. Universidad Complutense de Madrid.

[26] C. Dania and M. Clavel. OCL2FOL$^+$: Coping with undefinedness. In *OCL@MoDELS*, volume 1092 of *CEUR Workshop Proceedings*, pages 53–62, 2013.

[27] C. Dania and M. Clavel. OCL2MSFOL: a mapping to many-sorted first-order logic for efficiently checking the satisfiability of OCL constraints. In B. Baudry and B. Combemale, editors, *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, Saint-Malo, France, October 2-7, 2016*, pages 65–75. ACM, 2016.

[28] C. Dania and M. Egea. SQLPL*4*OCL tool, 2016. http://software.imdea.org/~dania/tools/sqlpl4ocl.

[29] M. A. G. de Dios, C. Dania, D. Basin, and M. Clavel. Model-driven development of a secure eHealth application. In *Engineering Secure Future Internet Services*, volume 8431 of *LNCS*, pages 97–118. Springer, 2014.

[30] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[31] B. Demuth and H. Hußmann. Using UML/OCL Constraints for Relational Database Design. In R. B. France and B. Rumpe, editors, *UML*, volume 1723 of *LNCS*, pages 598–613. Springer, 1999.

[32] B. Demuth, H. Hußmann, and S. Loecher. OCL as a Specification Language for Business Rules in Database Applications. In M. Gogolla and C. Kobryn, editors, *UML*, volume 2185 of *LNCS*, pages 104–117. Springer, 2001.

[33] D. D'Souza and A. Wills. Catalysis. Practical Rigor and Refinement: Extending OMT, Fusion, and Objectory. Technical report, http://catalysis.org, 1995.

[34] M. Egea, C. Dania, and M. Clavel. MySQL4OCL: A stored procedure-based MySQL code generator for OCL. *ECEASST*, 36, 2010.

[35] Facebook. Facebook Help Center. 2013. `http://www.facebook.com/help`.

[36] Facebook. Facebook Help Center. 2014. `http://www.facebook.com/help`.

[37] I. O. for Standardization. ISO/IEC 9075-(1–10) Information technology – Database languages – SQL, 2011. `http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=63555`.

[38] M. Gogolla, F. Büttner, and J. Cabot. Initiating a benchmark for UML and OCL analysis tools. In *TAP 2013, Budapest, Hungary. Proceedings*, volume 7942 of *LNCS*, pages 115–132. Springer, 2013.

[39] M. Gogolla, F. Büttner, and M. Richters. USE: A UML-based specification environment for validating UML and OCL. *SCP*, 69(1-3):27–34, 2007.

[40] M. Gogolla, L. Hamann, F. Hilken, M. Kuhlmann, and R. B. France. From application models to filmstrip models: An approach to automatic validation of model dynamics. In H. Fill, D. Karagiannis, and U. Reimer, editors, *Modellierung*, volume 225 of *LNI*, pages 273–288. GI, 2014.

[41] C. A. González, F. Büttner, R. Clarisó, and J. Cabot. EMFtoCSP: a tool for the lightweight verification of EMF models. In S. Gnesi, S. Gruner, N. Plat, and B. Rumpe, editors, *Proceedings of FormSERA 2012, Zurich, Switzerland, June 2, 2012*, pages 44–50. IEEE, 2012.

[42] J. Y. Halpern and V. Weissman. Using first-order logic to reason about policies. *ACM Trans. Inf. Syst. Secur.*, 11(4), 2008.

[43] R. Harjani, M. Arjona, A. Muoz, and A. Maa. Towards an Engineering Process for Certified Multilayer Cloud Services, Layered Assurance Workshop. In *ASAC*, 2013.

[44] N. Helil and K. Rahman. Extending XACML profile for RBAC with semantic concepts. 2010.

[45] Irish Data Protection Commissioner. Facebook Ireland Ltd. Report of Audit, December 2011. `http://www.dataprotection.ie/documents/facebook%20report/final%20report/report.pdf`.

[46] Irish Data Protection Commissioner. Facebook Ireland Re-Audit Report, September 2012. `http://www.dataprotection.ie/documents/press/Facebook_Ireland_Audit_Review_Report_21_Sept_2012.pdf`.

[47] D. Jackson. Alloy: a new technology for software modelling. In J. Katoen and P. Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002, Proceedings*, volume 2280 of *LNCS*, page 20. Springer, 2002.

[48] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[49] S. Jajodia, P. Samarati, M. L. Sapino, and V. S. Subrahmanian. Flexible support for multiple access control policies. *ACM Trans. Database Syst.*, 26(2):214–260, 2001.

[50] B. Kanso and S. Taha. Temporal constraint support for OCL. In K. Czarnecki and G. Hedin, editors, *Software Language Engineering*, volume 7745 of *LNCS*, pages 83–103. Springer Berlin Heidelberg, 2013.

[51] M. Kuhlmann and M. Gogolla. From UML and OCL to relational logic and back. In R. B. France, J. Kazmeier, R. Breu, and C. Atkinson, editors, *Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012. Proceedings*, volume 7590 of *LNCS*, pages 415–431. Springer, 2012.

[52] M. Kuhlmann, K. Sohr, and M. Gogolla. Comprehensive two-level analysis of static and dynamic RBAC constraints with UML and OCL. In *Fifth International Conference on Secure Software Integration and Reliability Improvement, SSIRI 2011*, pages 108–117. IEEE, 2011.

[53] M. Kuhlmann, K. Sohr, and M. Gogolla. Employing UML and OCL for designing and analysing role-based access control. *Mathematical Structures in Computer Science*, 23(4):796–833, 2013.

[54] J. Li, Y. Tang, C. Mao, H. Lai, and J. Zhu. Role based access control for social network sites. In *Pervasive Computing (JCPC), 2009 Joint Conferences on*, pages 389 –394, dec. 2009.

[55] MagicDraw Modelling Tool. `http://www.nomagic.com/products/magicdraw.html`.

[56] A. Maraee and M. Balaban. Efficient reasoning about finite satisfiability of UML class diagrams with constrained generalization sets. In *ECMDA-FA 2007, Haifa, Israel, Proccedings*, volume 4530 of *LNCS*, pages 17–31. Springer, 2007.

[57] R. Marcano-Kamenoff and N. Lévy. Using B formal specifications for analysis and verification of UML/OCL models. In *Workshop on consistency problems in UML-based software development. UML Conference*, Dresden, Germany, 2002.

[58] MariaDB, 2016. `https://mariadb.org/`.

[59] F. Massacci. Reasoning about security: a logic and a decision method for role-based access control. In D. M. Gabbay, R. Kruse, A. Nonnengart, and H. J. Ohlbach, editors, *Qualitative and Quantitative Practical Reasoning, First International Joint Conference on Qualitative and Quantitative Practical Reasoning ECSQARU-FAPR'97, Proceedings*, volume 1244 of *LNCS*, pages 421–435. Springer, 1997.

[60] Microsoft. SQL Server, 2016. `https://www.microsoft.com/es-es/server-cloud/products/sql-server/overview.aspx`.

[61] MySQL 5.7 Reference Manual. `http://dev.mysql.com/doc/refman/5.7/`.

[62] NESSoS. The European Network of Excellence on Engineering Secure Future internet Software Services and Systems, 2010. `http://www.nessos-project.eu`.

[63] OASIS. eXtensible Access Control Markup Language (XACML), 2010. `http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-cs-01-en.pdf`.

[64] OASIS. XACML core and hierarchical role-based access control. `http://docs.oasis-open.org/xacml/3.0/`, 2010.

[65] OASIS. Extensible access control markup language (XACML). `http://docs.oasis-open.org/xacml/3.0/`, 2013.

[66] Object Management Group. Model Driven Architecture Guide v. 1.0.1. Technical report, OMG, 2003. OMG documento disponible en `http://www.omg.org/cgi-bin/doc?omg/03-06-01`.

[67] Object Management Group. *Unified Modeling Language*, Mar 2011. OMG documento disponible en `http://www.omg.org/spec/UML/2.4`.

[68] Object Management Group. Object constraint language specification version 2.4. Technical report, OMG, 2014. `http://www.omg.org/spec/OCL/2.4`.

[69] OCL2FOL+ Project, 2014. `http://software.imdea.org/~dania/tools/ocl2folplus`.

[70] OCL2MSFOL Project, 2016. `http://software.imdea.org/~dania/tools/ocl2msfol`.

[71] Object Management Group. `http://www.omg.org`.

[72] N. O'Neill. Infographic: The History of Facebook's Default Privacy Settings. `http://www.allfacebook.com`.

[73] X. Oriol and E. Teniente. Incremental checking of OCL constraints through SQL queries. In A. D. Brucker, C. Dania, G. Georg, and M. Gogolla, editors, *Proceedings of the 14th International Workshop on OCL and Textual Modelling co-located with 17th International Conference on Model Driven Engineering Languages and Systems (MODELS 2014), Valencia, Spain, September 30, 2014.*, volume 1285 of *CEUR Workshop Proceedings*, pages 23–32. CEUR-WS.org, 2014.

[74] PARIS Project. `http://www.paris-project.org/`.

[75] PL/pgSQL - SQL procedural language, 2016. `https://www.postgresql.org/docs/9.2/static/plpgsql.html`.

[76] N. Przigoda, F. Hilken, J. Peters, R. Wille, M. Gogolla, and R. Drechsler. Integrating an smt-based modelfinder into USE. In M. Famelis, D. Ratiu, and G. M. K. Selim, editors, *Proceedings of the 13th Workshop on Model-Driven Engineering, Verification and Validation co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MODELS 2016), Saint-Malo, France, October 3, 2016.*, volume 1713 of *CEUR Workshop Proceedings*, pages 40–45. CEUR-WS.org, 2016.

[77] A. Queralt, A. Artale, D. Calvanese, and E. Teniente. OCL-Lite: Finite reasoning on UML/OCL conceptual schemas. *DKE*, 73:1–22, 2012.

[78] A. Queralt, G. Rull, E. Teniente, C. Farré, and T. Urpí. AuRUS: Automated reasoning on UML/OCL schemas. In *Conceptual Modeling - ER 2010, Vancouver, BC, Canada. Proceedings*, pages 438–444, 2010.

[79] L. A. R. Mapping from OCL/UML metamodel to PVS metamodel.

[80] C. D. P. K. Ramli, H. R. Nielson, and F. Nielson. The logic of XACML. *Sci. Comput. Program.*, 83:80–105, 2014.

[81] A. J. Reynolds. *Finite model finding in satisfiability modulo theories.* PhD thesis, University of Iowa, 2013.

[82] D. Roe, K. Broda, and A. Russo. Mapping UML models incorporating OCL constraints into Object-Z. Technical report, Imperial College of Science, Technology and Medicine, 2003.

[83] J. Ruiz, A. Rein, M. Arjona, A. Maa, A. Monsifrot, and M. Morvan. Security Engineering and Modelling of Set-Top Boxes. In *Proc. of ASE/IEEE BioMedCom*, 2012.

[84] J. F. Ruiz, A. Maa, M. Arjona, and J. Paatero. Emergency Systems Modelling using a Security Engineering Process. In *Proc. of 3rd Int. Conf. SIMULTECH*. SciTePress, 2013.

[85] G. Rull, C. Farré, A. Queralt, E. Teniente, and T. Urpí. AuRUS: explaining the validation of UML/OCL conceptual schemas. *SoSyM*, 14(2):953–980, 2015.

[86] A. Sánchez. *Formal Verification of Temporal Properties for Parametrized Concurrent Programs and Concurrent Data Structures.* PhD thesis, Universidad Politécnica de Madrid, Spain, September 2015.

[87] A. Schmidt. Untersuchungen zur Abbildung von OCL-ausdrücken auf SQL. Master's thesis, Institut für Softwaretechnik II - Technische Universität Dresden, Germany, 1998.

[88] SecFutur Project. `http://www.secfutur.eu/`.

[89] SecProver, 2014. `http://actiongui.org/`, see SecProver project.

[90] A. Simpson. On the Need for User-Defined Fine-Grained Access Control Policies for Social Networking Applications. In *Proceedings of the workshop on Security in Opportunistic and SOCial networks*, SOSOC '08, pages 1:1–1:8, New York, NY, USA, 2008. ACM.

[91] N. Siripornpanit and S. Lekcharoen. An adaptive algorithms translating and back-translating of object constraint language into structure query language. In *International Conference on Information and Multimedia Technology, 2009. ICIMT'09.*, pages 149–151. IEEE, 2009.

[92] P. Sobotka. Transformation from OCL into SQL, 2012. Master thesis. Charles University in Prague. `https://is.cuni.cz/webapps/zzp/download/120076745`.

[93] M. Soeken, R. Wille, M. Kuhlmann, M. Gogolla, and R. Drechsler. Verifying UML/OCL models using boolean satisfiability. In *DATE 2010, Dresden, Germany.*, pages 1341–1344. IEEE, 2010.

[94] J. M. Spivey. *The Z notation: a reference manual.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[95] M. Szlenk. Formal semantics and reasoning about UML class diagram. In *DEPCOS-RELCOMEX*, pages 51–59, Washington, DC, USA, 2006. IEEE.

[96] Traces Analyzer Project, 2015. `http://software.imdea.org/~dania/tools/tracesAnalyzer.html`.

[97] R. Wieringa. A survey of structured and object-oriented software specification methods and techniques. *ACM Comput. Surv.*, 30(4):459–527, 1998.

[98] SQL Dialects Reference, 2016. `https://en.wikibooks.org/wiki/SQL_Dialects_Reference/Print_version`.

[99] T. Y. C. Woo and S. S. Lam. Authorizations in distributed systems: A new approach. *Journal of Computer Security*, 2(2-3):107–136, 1993.

[100] C. Zhao, N. Heilili, S. Liu, and Z. Lin. Representation and reasoning on RBAC: a description logic approach. In D. V. Hung and M. Wirsing, editors, *Theoretical Aspects of Computing - ICTAC 2005, Second International Colloquium, Proceedings*, volume 3722 of *LNCS*, pages 381–393. Springer, 2005.

[101] M. Zuckerberg. Facebook and the Irish Data Protection Commission. The Facebook Blog, Dec. 2011. `https://blog.facebook.com`.

[102] M. Zuckerberg. Our Commitment to the Facebook Community. The Facebook Blog, Nov. 2011. `https://blog.facebook.com`.

BASTA!
QUINO