Universidad de Oviedo
Universidá d'Uviéu
University of Oviedo

RESUMEN DEL CONTENIDO DE TESIS DOCTORAL

1.- Título de la Tesis	
Español/Otro Idioma: Métodos para la asistencia y agilización del diseño y desarrollo de clientes ricos robustos.	Inglés: Methods for the assistance and speed-up of the design and development of robust rich clients

2.- Autor	
Nombre: Manuel Quintela Pumares	DNI/Pasaporte/NIE:
Programa de Doctorado: Informática	
Órgano responsable: Comisión académica de doctorado del departamento de informática	

RESUMEN (en español)

Los clientes ricos son capaces de descargar parte del modelo de objetos del servidor y manipularlo localmente sin notificar cada cambio al mismo. Esto proporciona una experiencia de usuario más interactiva, reduce la carga de comunicación y el tiempo de espera. Su desarrollo es, sin embargo, una tarea compleja. Cliente y servidor deben tener diferentes modelos de dominio que sean coherentes entre sí, y definir restricciones apropiadas para garantizar la robustez de la aplicación.

El diseño de ambos modelos de dominio generalmente implica el uso de diagramas de clases UML así como las restricciones OCL. El diseñador debe definir tanto el modelo de dominio global (Global Domain Model, GDM), que estará ubicado en el servidor, como el modelo de dominio del cliente (Client Domain Model, CDM). El CDM suele ser un subconjunto del GDM, donde el diseñador debe repetir y adaptar partes del GDM para crearlo. Esta tarea también implica decidir si hay restricciones definidas en el GDM que también deberían estar presentes en el CDM. Verificar el mayor número de restricciones posibles en el cliente, incluso cuando también están presentes en el servidor, da lugar a un cliente más robusto y a una experiencia de usuario más fluida, ya que el cliente detectaría las inconsistencias sin esperar a comunicarse con el servidor.

Debido a que el diseño del CDM y sus restricciones dependen del GDM, su diseño requiere un esfuerzo considerable en tareas que suelen ser repetitivas, tediosas y propensas a errores. Dado que la información requerida para tomar estas decisiones generalmente está disponible en un formato estructurado como parte del GDM, este trabajo analiza y se ocupa de la automatización de la creación del CDM a partir de la información del servidor.

Contribuciones de este trabajo.

Este trabajo propone un método que resuelve parcialmente el objetivo de automatizar la creación del modelo de dominio de cliente a partir de la información del modelo de dominio del servidor. Dado un GDM completo con

sus restricciones representadas por medio de OCL y el conjunto de entidades seleccionadas por el diseñador para crear el CDM, la propuesta aborda la generación automática del CDM y detecta las restricciones de OCL del GDM que pueden ser aplicadas al CDM. También detecta las restricciones que requerirían un procesamiento manual adicional por parte de los desarrolladores para adaptarlas al CDM.

Para las restricciones que requieren intervención manual, el método también genera documentación que proporciona información útil sobre cada una de ellas. Esto incluye una representación visual de la restricción que ayuda a comprender los elementos involucrados en su verificación, métricas objetivas que describen la complejidad de la restricción, y la clasificación de las mismas. Todos estos elementos combinados ayudan al desarrollador en la toma de decisiones, proporcionando un acceso fácil a información que no es trivial tal y como se representa en UML y OCL.

Este trabajo presenta las siguientes contribuciones:

— Un algoritmo para la generación automática del CDM a partir del GDM basado en técnicas de Model Slicing.

— Una propuesta para la representación visual de los elementos involucrados en la verificación de una restricción OCL, llamada árboles de instancia.

— Algoritmos para la generación automática de árboles de instancia a partir del árbol de sintaxis abstracta (AST) de una expresión OCL, y para la extracción de métricas objetivas sobre restricciones de árboles de instancia.

— Algoritmos que utilizan las métricas extraídas para una restricción para clasificarlas en función de la variedad de elementos afectados por ella y su nivel de dependencia del servidor.

— Descripciones formales de todos los procesos y algoritmos descritos en este trabajo, independientemente de cualquier tecnología específica.

— Un prototipo de trabajo que implementa la propuesta basada en las descripciones formales y utilizando la tecnología existente.

RESUMEN (en Inglés)

Rich client applications are able to download part of the object model from the server and manipulate it locally without having to notify back every change to the server. This provides a more interactive and responsive user experience, reduces the client-server communication load and the perceived delay. Their development is, however, a complex task. Client and server must have different domain models that are consistent with each other, and define appropriate constraints to ensure the robustness of the application.

Designing these domain models usually involves using UML class

diagrams as well as the OCL constraints. The designer must delimit both the global domain model (GDM) —that will be located on the server— and the client domain model (CDM). The CDM is usually a subset of the GDM, where the designer must repeat and adapt parts of the GDM to create it. This task also involves deciding if there are constraints defined on the GDM that should also be present on the CDM. Verifying as many constraints as possible on the client, even when they are also present on the server, would lead to a more robust client and provide a more fluid user experience, since the client would prevent inconsistencies without waiting for their reintegration onto the server.

Since the design of the CDM and its constraints is dependent on the GDM, designing it requires a considerable effort in tasks that are usually repetitive, tedious and error-prone. Given that the information required to take this decisions is usually available in a structured format as part of the GDM, this work analyzes and deals with the potential automation of the creation of the CDM based on the server information.

Contributions of this work

This work proposes a method that partially solves the goal of automating the creation of the client domain model from the information present on the server domain model. Given a complete GDM with its restrictions represented by means of OCL and the set of entities selected by the designer to create the CDM, the proposal deals with the automatic generation of the CDM and detects the OCL constraints from the GDM that can be present on the CDM. It also detects the constraints that would require further manual processing from the developers to be adapted to the CDM.

For those constraints that require manual intervention, the method also generates documentation that provides helpful information about each one of them. This includes a visual representation of the constraint that aids in understanding the elements involved in its verification, objective metrics describing the complexity of the constraint, and its classification. All these elements combined help the developer in the decision-making process providing easy access to information that is not trivial to understand as represented in UML and OCL.

The work presents the following constributions:

— An algorithm for the automatic generation of the CDM from the GDM based on Model Slicing techniques.

— A proposal for the visual representation of the elements involved in the verification of an OCL constraint, called instance trees

— Algorithms for the automatic generation of instance trees from the abstract syntax tree (AST) of an OCL expression and the extraction of objective metrics about constraints from instance trees.

— Algorithms that uses the metrics extracted for a constraint to classify

them based on the variety of elements affected by it and their level of dependency from the server.

— Formal descriptions of all the processes and algorithms described in this work, independent of any specific technology.

— A working prototype that implements the proposal based on the formal descriptions and using existing technology.

SR. PRESIDENTE DE LA COMISIÓN ACADÉMICA DEL PROGRAMA DE DOCTORADO EN _____

Universidad de Oviedo

MÉTODOS PARA LA ASISTENCIA Y AGILIZACIÓN DEL DISEÑO Y DESARROLLO DE CLIENTES RICOS ROBUSTOS

Manuel Quintela Pumares

Departamento de Informática

Tesis presentada para la obtención del título de
Doctor por la Universidad de Oviedo

Dirigida por los Drs.
Daniel Fernández Lanvin
Alberto Manuel Fernández Álvarez

Noviembre de 2018

AGRADECIMIENTOS

A mi familia, por su apoyo y ánimos durante todo este tiempo.

A mis directores, por su ayuda y disposición constantes, y su concienzuda implicación en todos los aspectos de este trabajo.

A todos los compañeros del departamento en Oviedo, en particular a todos los que hemos compartido espacio y horas en el laboratorio. También a los compañeros del grupo REFLECTION, y especialmente a Francisco Ortín, cuya ayuda a lo largo de estos años ha sido inestimable.

Métodos para la asistencia y agilización del diseño y desarrollo de clientes ricos robustos

Manuel Quintela Pumares

Noviembre 2018

RESUMEN

Los clientes ricos han emergido como una propuesta sólida y popular en aplicaciones tanto web como nativas. Su capacidad para gestionar su propio modelo de dominio y verificar restricciones localmente proporciona una experiencia de usuario más robusta e interactiva. Su modelo de dominio local es habitualmente un subconjunto del modelo global ubicado en el servidor, por lo que ambos extremos deben gestionar de forma consistente las entidades, relaciones y restricciones que comparten. Diseñar clientes con estas características manualmente implica replicar partes del modelo de dominio del servidor, requiriendo además ser adaptadas teniendo en cuenta aquellos elementos del servidor que no están presentes en el cliente, así como identificar cuáles de las restricciones del servidor pueden ser verificadas localmente. Esta tarea es compleja y propensa a introducir errores, y además cualquier modificación en el modelo del servidor implica revisar todo el trabajo realizado para el cliente. La contribución de este trabajo para afrontar este problema es la propuesta de un proceso que, a partir de la información del modelo del servidor y sus restricciones, genera de forma automática el modelo de dominio del cliente y detecta las restricciones que pueden ser aplicadas al mismo. Dado que algunos aspectos del proceso no son automatizables, el modelo de dominio y restricciones son analizados para generar documentación que permita asistir en la toma de decisiones. Esta documentación incluye métricas acerca de cada restricción y una clasificación en función de su tipo y nivel de dependencia con el servidor. La propuesta se describe de un modo formal e independiente de tecnologías específicas, y basándose en ella se ha desarrollado un prototipo funcional.

PALABRAS CLAVE

Restricciones, Clientes Ricos, Modelos de Dominio, UML, OCL, Model Slicing

Métodos para la asistencia y agilización del diseño y desarrollo de clientes ricos robustos

Manuel Quintela Pumares

Noviembre 2018

ABSTRACT

Rich clients have emerged as a solid and popular approach in both web and native applications. Their capability to manage their own domain model and locally verify constraints provides a more responsive and robust user experience. Its local domain model is usually a subset of the global model located on the server, so both ends must consistently manage the entities, relationships and constraints they share. Designing clients with these characteristics manually implies replicating part of the server domain model Furthermore, this subset also may require to be adapted considering those elements of the server that are not present in the client, as well as identifying which of the server's restrictions can be verified locally. This is a complex and error-prone task. Any further modification in the server model involves reviewing all the work done for the client. The contribution of this work to address with this problem is the proposal of a process that, based on the information of the server model and its restrictions, automatically generates the client domain model and identifies the restrictions that can be applied to it. Since some aspects of the process are not automatable, the domain model and constraints are analyzed to generate documentation to assist in the decision-making. This documentation includes metrics about each restriction and a classification based on its type and level of dependency with the server. The proposal is described formally and independently from specific technologies, and a functional prototype has been developed based on it.

KEYWORDS

Constraints, Rich Clients, Domain Models, UML, OCL, Model Slicing

CONTENIDOS

LISTA DE TABLAS

LISTA DE FIGURAS

LISTA DE EJEMPLOS DE CÓDIGO

LISTA DE DESCRIPCIONES FORMALES

1 INTRODUCTION

Web-based rich client applications have become very popular and widespread. In contrast to classic web applications in which each interaction from the user involves a request/response call to the server (and the subsequent reloading of the whole page), rich clients are focused on a different interaction model. In order to improve user experience, by reducing response times during interaction, a rich client downloads part of the object model from the server and manipulates it locally without having to notify back every change [1]. Once the transformation is finished, it delivers the new version back to the server. This provides a more interactive and responsive user experience, reduces the client-server communication load and the perceived delay [2]. These improvements in user experience are the main reason for rich clients to become quite popular in different platforms, mainly as web applications for browsers or native applications for different smartphone operating systems.

1.1 Description of the problem

The client-server strategy involves some drawbacks from the design point of view that turns its development into a complex and error-prone task. During the download-transformation-delivery cycle, the object model on the server can also be modified by different clients or processes. The transformations on each client can lead to inconsistencies with respect to the new state of the server object model. So there must be clear constraints defined on the server model that are verified every time a client object model is reintegrated into the server object model.

Designing the domain model of an application with such behavior usually involves using UML class diagrams as well as the OCL standard to define the constraints. The designer must delimit both the global domain model (GDM) —that will be located on the server— and the client domain model (CDM) —a GDM subset that must be replicated in the client application. That involves not only identifying the classes —something trivial—, but also the constraints that can be checked on the client [3], the way they must be checked (completely or partially), and again, which of them can be checked on the server side once the model is delivered back to the server. This constraint management is not trivial at all.

The simplest strategy to face that temporal duplication of an object model subset would be delaying the constraint check until the object sub-model is integrated back into the global object model. However, to guarantee a consistent local manipulation on the client —so that it can be reintegrated properly on the server while maintaining a responsive user experience— it would be desirable to check as many constraints as possible on the client, even when they are also present on the server [4][5]. This would lead to a more robust client and provide a more fluid user experience, since the client would detect and prevent inconsistencies without waiting for their reintegration onto the server. Therefore, the designer must analyze and adjust the GDM constraints to select and adapt those that can and/or must be checked in the client application.

Generally, only a subset of the GDM classes will be needed on the CDM. So, firstly, the designer must decide how to adapt these GDM classes considering that some of their relationships are linked to other classes that are not required on the CDM. After that, the designer must identify the OCL constraints that can be checked directly in the client application, and which of them must be adapted or split up beforehand. For example, it is not unusual to have an OCL constraint that involves multiple classes, some of which can be beyond the scope of the CDM. In this case, the designer must decide if it may be adapted or ignored on the CDM [6].

In summary, the slicing of the GDM into a CDM requires a considerable design effort in tasks that are usually repetitive, tedious and error-prone. Furthermore, this partial model replication involves a considerably higher product maintenance complexity, which increases the workload and the risk of adding errors [7].

After analyzing the procedure that the designer will apply to solve this issue, it happens that the decisions he/she will take regarding the development of the CDM are mostly based in the information self-contained in the GDM that affects to the entities in the scope of the CDM and their relationships. These two elements determine the applicability of any GDM restriction. Given that all this information is usually available in a structured format as part of the GDM, this work analyzes and deals with the potential automation of this GDM slicing. This automation would not only lighten the work of the designer, but reduce the

development and maintenance cost, at the time it removes the potential errors in the project that the duplicity of constraint checking would involve.

1.2 Proposal

According to the ideas summarized in section 1.1, this work proposes a method that partially solves this goal of automating the model slicing activity by facing the first stages of the whole process. Basically, given a complete GDM with its restrictions represented by means of OCL and the set of entities that the designer requires to create the CDM, the proposal deals with the automatic generation of the CDM and classifies the OCL constraints from the GDM informing the developer about which of them can be automatically checked in the CDM without supervision and which would require further manual processing from the developers.

Specifically, the classification indicates (i) the elements they affect, (ii) which ones can be checked on the CDM without needing communication with the server, (iii) which are not related to the CDM at all, and (iv) which can be checked on the CDM but may require communication with the server. This last set cannot be determined just with the information contained in the GDM. Therefore, this task requires developer's intervention to analyze the requirements of the system and decide if they would require communication with the server or not. To ease this task, the method also generates documentation that provides helpful information about each constraint. This includes a visual representation of the constraint that aids in understanding the elements involved in its verification, objective metrics describing the complexity of the constraint, and, of course, its classification. All these elements combined help the developer in the decision making process providing easy access to information that is not trivial to understand as represented in UML and OCL.

The proposed method would support the automation of the CDM design and its maintenance through the development cycle, and would help developers to design CDMs maximizing the local checking of GDM constraints insofar as possible.

1.3 Objectives

Assuming the previous existence of a GDM, with its UML class model and OCL constraints, the proposal presents the following objectives:

- The CDM creation should be based on the available information from the GDM and as automatable as possible.
- The stages of the process that are not automatable should be supported by the generation of documentation that classifies the constraints and assists the designer on the decision-making.

— The proposal should be implementable on existing platforms or technologies, without forcing a specific one.

1.4 Contributions of this work

This work presents the following contributions:

— An algorithm for the automatic generation of the CDM from the GDM based on *Model Slicing* techniques.

— A proposal for the visual representation of the elements involved in the verification of an OCL constraint, called instance trees, and an algorithm for their automatic generation from the abstract syntax tree (AST) of an OCL expression.

— An algorithm to extract objective metrics about constraints from instance trees.

— An algorithm that uses the metrics extracted for a constraint to classify it based on the variety of elements affected by it.

— An algorithm that uses the information from the CDM and the metrics extracted for a constraint to classify it based on its level of dependency from the server.

— A formal description of UML Class models and Instance trees based on graph theory.

— Formal descriptions of all the algorithms to generate the CDM, instance trees, metric extraction and classification of constraints presented on this work.

— A working prototype that implements the proposal based on the formal descriptions and using existing technology.

1.5 Research context

This work has been funded by the European Union, through the European Regional Development Funds (ERDF); and the Principality of Asturias, through its Science, Technology and Innovation Plan (grant GRUPIN14-100), whose PI is Francisco Ortín Soler.

The research has been developed within the *Domain Modeling Research Group*, in the context of an ongoing research involving automatic error recovery techniques [8], optimization of the verification of OCL constraints [9]. Rich client robustness has also been a focus of this group [10] [11] where the Phd candidate has been highly involved.

1.6 Publications

In Journal

— *Manuel Quintela Pumares; Daniel Fernández Lanvin; Alberto Manuel Fernández Álvarez*; **Domain Model Slicing and Constraint Classification for Local Validation on Rich Clients**. *Journal of Systems and Software (2017)*. [12]

In relevant conferences

— *Manuel Quintela Pumares; Daniel Fernández Lanvin; Alberto Manuel Fernández Álvarez; Raúl Izquierdo Castanedo*. **Automatic Classification of Domain Constraints for Rich Client Development**. *ICSEA 2014*. [13]

Related publications

While the following publications do not address directly the contents of this document, they have influenced it or are closely related. They are enclosed in the same research group and may give insight on future lines of research.

— *Alberto Manuel Fernández Álvarez, Daniel Fernández Lanvin, Manuel Quintela Pumares*; **Extending the Consistency Property of Software Transactional Memory Systems with business constraints in OCL and incremental checking**. *IMCIC 2017*. [14]

— *Alberto Manuel Fernández Álvarez; Daniel Fernández Lanvin; Manuel Quintela Pumares*. **Invariant Implementation for Domain Models Applying Incremental OCL Techniques**. *Software Technologies (2016)*. [9]

— *Manuel Quintela Pumares; Bruno Cabral; Daniel Fernández Lanvin; Alberto Manuel Fernández Álvarez*. **Integrating automatic backward error recovery in asynchronous rich clients**. *ICSE 2016*. [11]

— *Alberto Manuel Fernández Álvarez; Daniel Fernández Lanvin; Manuel Quintela Pumares*. **OCL for rich domain models implementation. An incremental aspect based solution**. *ICSOFT 2015*. [15]

— *Manuel Quintela Pumares; Daniel Fernández Lanvin; Raúl Izquierdo Castanedo; Alberto Manuel Fernández Álvarez*. **Implementing automatic error recovery support for rich Web clients**. *WISE 2010*. [10]

1.7 Organization

This work is structured as follows:

Chapter 2 describes the problems associated with designing rich clients and local validation of constraints. Chapter 3 presents the general objectives and proposal. Chapter 0 details the characteristics of rich clients. Chapter 0 describes the UML and OCL standards for defining domain models. Chapter 0 looks into

current techniques being used for distributing the responsibilities between client and server. Chapter 0 explains the sequence of steps proposed in the proposal. Chapter 0 describes the first step of the process, the generation of explicit constraints and simplification of constraints. Chapter 0 details the automatic generation of the CDM through *Model Slicing* techniques. Chapter 0 shows how constraints are analyzed and how Instance Trees are created. Chapter 0 describes how objective metrics are extracted from instance trees, and how this information is used to classify the constraints. Chapter 1 develops the whole proposal formally. Chapter 1 describes the implemented prototype. Chapter 1 presents the conclusion and future work. Finally, chapter 0 is dedicated to the acknowledgments while chapter 0 lists the bibliography.

Following the normative for the International PhD Mention, this chapter and chapter 1 are both written in English, while the rest of the document is written in Spanish. For a more detailed description in English, the main publications detailing this work are [12] and [13].

2 MOTIVACIÓN Y PRESENTACIÓN DEL PROBLEMA

La arquitectura de las aplicaciones web ha ido evolucionando continuamente desde la popularización de los primeros sistemas basados en scripts CGI [16]. De esos primeros modelos arquitectónicos donde toda la lógica de la aplicación se implementaba en el servidor, se ha avanzado hacia un modelo donde el cliente ha ido cobrando más peso en pos de una mejor experiencia de usuario. Con ánimo de evitar los tiempos de espera inherentes a esos primeros modelos arquitectónicos, se ha ido gradualmente responsabilizando al cliente de parte de la gestión de los datos, reduciendo la interacción con el servidor y, por lo tanto, agilizando la interacción con el usuario.

En los primeros tiempos esta tendencia se vio impulsada por la popularización de tecnologías web como Ajax, y en la actualidad continúa evolucionando potenciada principalmente por las actuales plataformas de desarrollo de aplicaciones móviles nativas. Estos avances permitieron evolucionar gradualmente a una arquitectura más distribuida en la que el cliente puede tener también su propio modelo de datos y lógica de negocio, y por tanto posibilitando interfaces más interactivas y tiempos de respuesta menores.

El desarrollo de clientes ricos presenta numerosas ventajas, pero su desarrollo plantea numerosos retos que deben ser abordados si se desea obtener

un producto robusto que maneje el tráfico de datos entre cliente y servidor de forma consistente.

2.1 Características de los clientes ricos

En las primeras aplicaciones basadas en tecnologías Web, el cliente se limitaba a mostrar las vistas producidas por el servidor y a proporcionar una interacción muy limitada. Desde el punto de vista del usuario, cada nueva interacción con la aplicación implicaba una nueva comunicación con el servidor y posterior refresco de la vista. La aparición gradual de nuevas tecnologías fue permitiendo desarrollar clientes más interactivos, capaces de almacenar y gestionar datos localmente y de disponer de su propia lógica [17]. A lo largo del tiempo han surgido numerosas opciones tecnológicas que permiten desarrollar este tipo de clientes. Los primeros clientes ricos se basaban en *plugins* para el navegador, y posteriormente el avance de los estándares Web permitió la implementación de estos de forma nativa. Más allá del navegador web tradicional, hoy este tipo de cliente es particularmente popular en los dispositivos móviles, en forma de aplicaciones nativas que se conectan a través del protocolo HTTP.

Por ello, en adelante y en el contexto de este trabajo el término "cliente rico" se referirá a cualquier tipo de aplicación, independientemente de la tecnología usada para su desarrollo o la plataforma en la que se ejecute, que posea las siguientes características:

- Es capaz de comunicarse con un servidor para enviar y recibir información de forma tanto síncrona como asíncrona.
- Contiene una lógica propia, que le permite manipular información local de forma independiente del servidor.
- Ofrece interactividad al usuario sin necesidad de cargar de nuevo la interfaz.
- No es estrictamente necesario notificar al servidor de cada acción individual que sucede en el cliente.

El principal beneficio que ofrecen los clientes ricos es una mejor experiencia de usuario, ya que la independencia del servidor proporciona las siguientes ventajas [18]:

- Permite al usuario llevar a cabo acciones más interactivas.
- Puede dar respuestas más inmediatas a las acciones del usuario.
- Proporciona un mayor control sobre cuándo se producen las comunicaciones entre cliente y servidor, que a menudo redunda en un tráfico de comunicaciones más eficiente.
- La comunicación asíncrona permite hacer que el usuario perciba los tiempos de espera entre comunicaciones como más cortos, o incluso

puede permitir la posibilidad de llevar a cabo otras acciones mientras se espera la respuesta del servidor.

— Puede disponerse de funcionalidades offline, o de estrategias para entornos con conectividad limitada o intermitente.

Hoy en día, el usuario es cada vez más exigente respecto a la experiencia que le ofrece una aplicación. El mercado actual le ofrece una gran oferta de sitios web y aplicaciones móviles que compiten entre ellas, y es sencillo y rápido probar varias de ellas y seleccionar la más satisfactoria. En este contexto, la capacidad de proporcionar al usuario una sensación de respuesta inmediata y fluidez en la interacción es uno de los criterios que impactan en la toma de decisiones del usuario medio [19].

2.2 Robustez en un cliente rico

Un cliente rico debería ser capaz de llevar a cabo todas las acciones mencionadas anteriormente sin que las acciones del usuario se vean detenidas por errores, ni permitiendo que la información del sistema pierda su integridad.

Aunque este aspecto es fundamental en cualquier tipo de software, los clientes ricos se enfrentan a retos particulares en este sentido. Al existir una colaboración entre cliente y servidor la información contenida en ambos está relacionada y ocasionalmente repetida, por lo que la robustez del sistema en su conjunto depende de que ambos manejen esta información correctamente [20]. Paralelamente, durante la comunicación entre ambos pueden surgir problemas, desde enviar información inapropiada que no se puede integrar correctamente, a problemas de transmisión en la red durante el envío de los datos. Es crítico que ambas partes del sistema sean capaces de responder de una forma apropiada a estas situaciones, dando continuidad a las funcionalidades del sistema y manteniendo su información en un estado consistente.

La primera ocasión para afrontar esta dificultad se da durante el diseño del sistema. Es necesario tener un modelo de dominio correctamente definido tanto para el cliente como para el servidor, con restricciones que definan qué condiciones debe cumplir el grafo de objetos en tiempo de ejecución para poder considerarse como íntegro.

2.3 El modelo de dominio de un cliente rico

Para crear aplicaciones conectadas que ofrezcan al usuario interacción a través de un cliente rico, tendremos por un lado el modelo del servidor, y por otro el del cliente.

El modelo del servidor se trata de un modelo global al que los clientes podrán hacer peticiones. Su modelo detallará las entidades que lo componen, sus relaciones, atributos, funcionalidades, y las restricciones que todos esos elementos deben cumplir en todo momento.

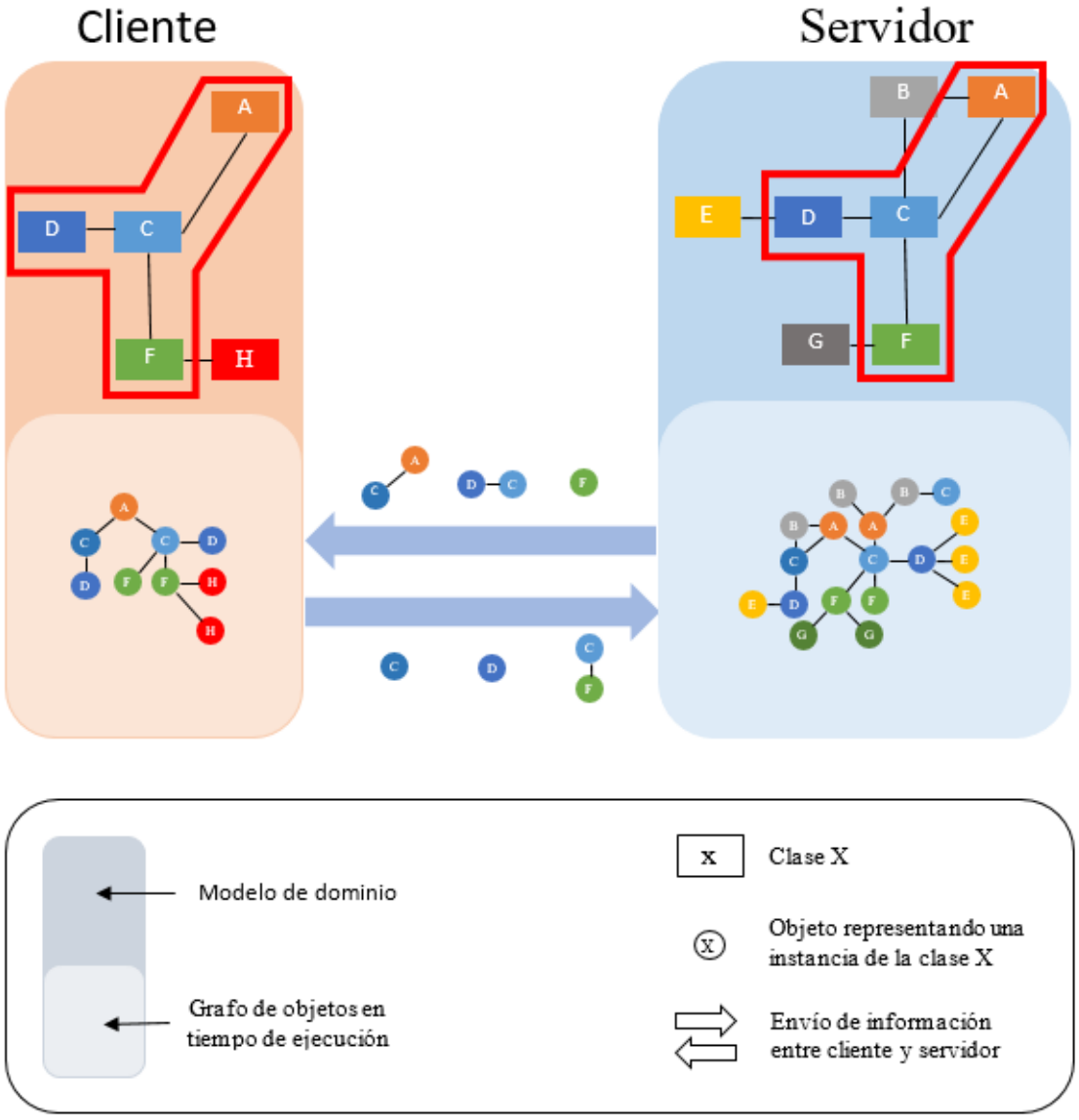


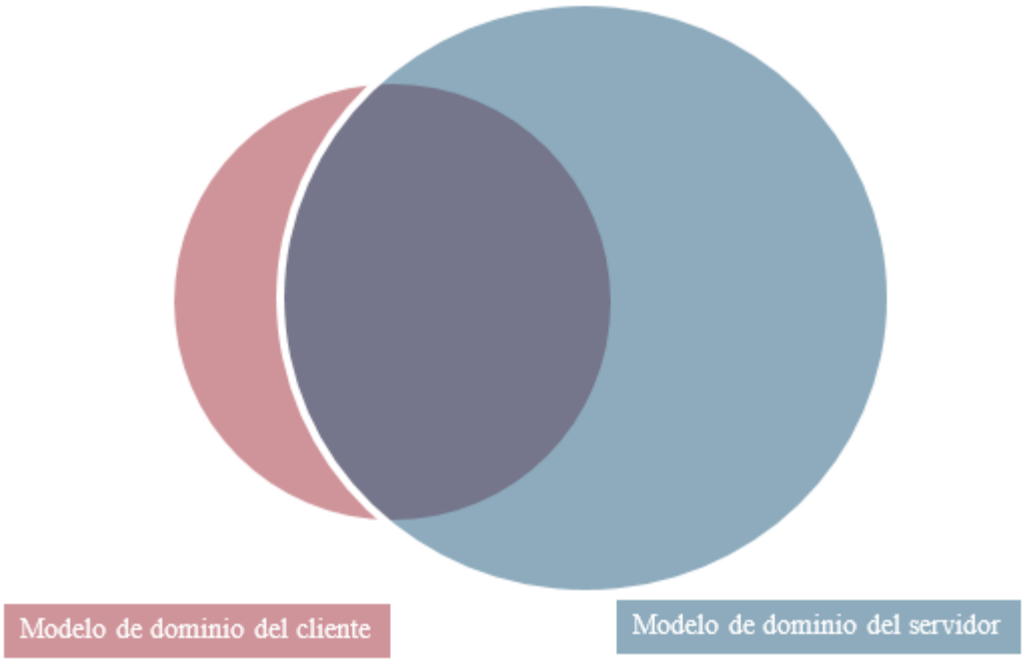
Fig 1: Cliente y servidor tienen cada uno su propio modelo de dominio, su propio estado, y existe un tráfico de información entre ellos.

Por su lado, un cliente que realice peticiones a dicho servidor y pretenda trabajar con cierta independencia del mismo debe ser capaz de gestionar y manipular datos localmente. Para ello debe tener su propio modelo de dominio y lógica de negocio, con sus correspondientes restricciones [21][20][6][22].

Dado que está diseñado para trabajar con el servidor, el modelo local que maneja el cliente suele contener un subconjunto de los elementos del modelo presente en el servidor.

De esta forma, el cliente puede realizar peticiones al servidor, incorporar la información recibida en su propio grafo de objetos, y manipularlo localmente. Posteriormente a la manipulación local, la información puede ser enviada de vuelta al servidor para que se actualice el grafo de objetos global que éste aloja. Esta comunicación es posible gracias a los elementos comunes del modelo de dominio, que permiten que la lógica de negocio de cada uno de los extremos trabaje sobre el mismo tipo de información de una forma consistente.

Fig 2: Parte del modelo de dominio del cliente estará solapado con el del servidor.



2.4 Gestión de restricciones en clientes ricos

Como se ha mencionado, el modelo del servidor incorpora una serie de restricciones que aseguren la integridad del mismo.

Si tenemos un cliente que implementa un subconjunto del modelo de dominio del servidor, éste puede verse también sometido a algunas de esas mismas restricciones.

La forma más sencilla de solventar este problema es delegar la verificación de todas las restricciones al servidor. En una arquitectura en la que solo el servidor verifica las restricciones, el cliente tiene dos opciones [7] [20]:

— Comunicar cada acción individual al servidor, y esperar que este verifique que ha sido correcta antes de continuar realizando otras.

— Permitir realizar series de acciones localmente y enviar el producto final al servidor para que valide el conjunto.

Si bien son efectivas, ambas aproximaciones entran en conflicto con una de las principales ventajas que conlleva el uso de clientes ricos: una respuesta ágil y fluida que mejore la experiencia de usuario. La primera opción limita esta

experiencia al introducir el tiempo de espera de la comunicación entre acción y acción. La segunda conlleva el riesgo de que, ante una violación de restricción durante la manipulación local, el usuario vea como el resto de sus acciones serán descartadas una vez se chequeen las restricciones en el servidor.

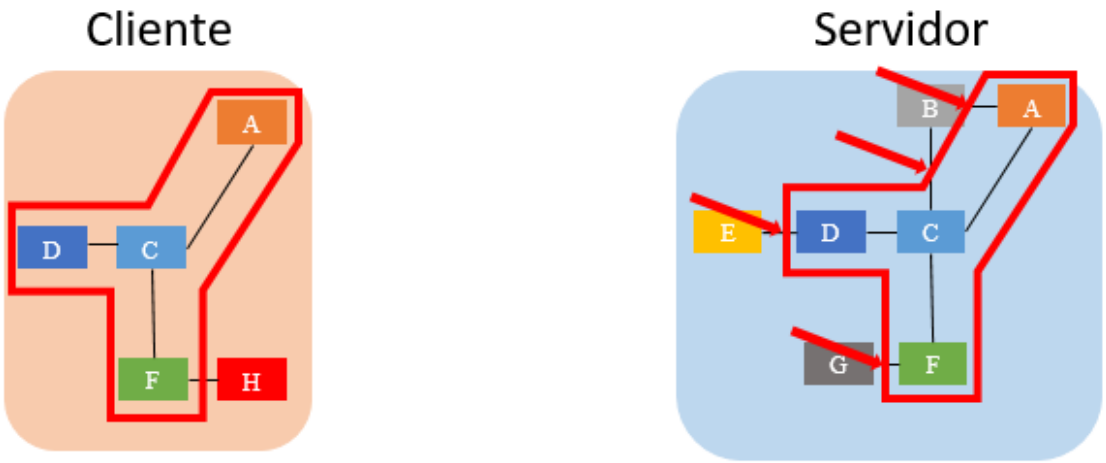
Ambos problemas se pueden evitar si el propio cliente pudiera verificar las restricciones localmente, informando al usuario del problema en el mismo instante que se produce y sin necesidad de consultar con el servidor. Si bien es posible diseñar un cliente que incorpore restricciones locales de una forma consistente con las del servidor, realizar esto manualmente plantea una serie de retos que hacen que la tarea sea compleja, como se describirá a continuación.

2.5 Presentación del problema

División del modelo

Cuando se desea diseñar clientes capaces de verificar restricciones localmente de una forma que sea consistente con el modelo del servidor, surgen varios retos.

Fig 3: Al crear un subconjunto del modelo original del servidor, hay relaciones que se rompen.



Habitualmente, el modelo del cliente solo contendrá algunas de las clases existentes en el servidor, es decir, un subconjunto del modelo de éste. Es probable que, en casos en los que dos o más clases del servidor tienen relaciones entre ellas, unas sean relevantes para el modelo del cliente mientras que otras no. En ese caso, es necesario decidir cómo trasladar eso al cliente, ya que llevar solo una de las partes implicaría romper esas relaciones. Dependiendo de los tipos de relaciones que están uniendo a las clases y su semántica, en algunos casos será posible romper dicha relación sin problemas, mientras que en otros implicaría la necesidad de llevarse al cliente clases que inicialmente no parecían necesarias. Del mismo modo, una clase trasladada al cliente puede definir métodos que hagan en su signatura alusión a otras clases que solo están en el servidor. En esas

circunstancias cabe plantearse si es necesario añadir dichas clases al cliente, o eliminar ese método del mismo.

Detección de restricciones aplicables en el cliente

Una vez se ha definido un modelo de clases válido para el cliente, no todas las restricciones presentes en el servidor serán aplicables localmente. Las restricciones se pueden definir con diferentes grados de complejidad, afectando a más o menos elementos del modelo. Son numerosas las propuestas sobre clasificación de restricciones en función de los elementos que son necesarias para verificarlas [23] [7] [24] [25]. Para los propósitos de este trabajo, se utilizará la siguiente clasificación basada en varios de los elementos comunes de otras propuestas:

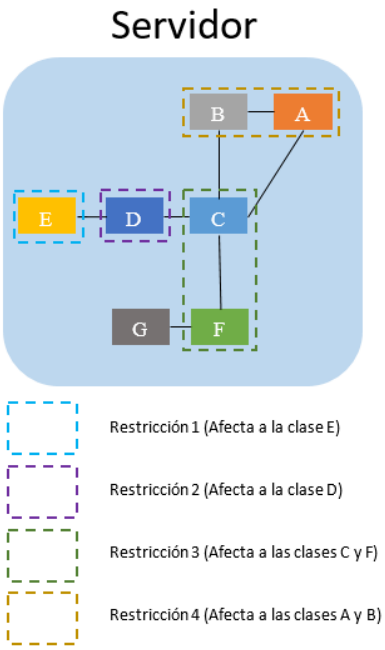


Fig 4: Elementos a los que afecta cada restricción.

- Atributo: Es necesario un único atributo de un objeto. Ejemplo: *Todos los Conductores deben tener una edad superior a 18 años*.
- Objeto: Son necesarios varios atributos del mismo objeto. Ejemplo: *Todos los Empleados deben ser mayores de 18 años y ser varones*.
- Clase: Son necesarios varios objetos de la misma clase. Ejemplo: *No puede haber dos Empleados con el mismo DNI*.
- Dominio: Son necesarios varios objetos de diferentes clases. Ejemplo: Todos los Conductores deben ser mayores de 18 años y tener un seguro activo.

Cada restricción definida para el servidor afectará a distintos elementos del modelo, como se puede observar en la Fig 4. De esas restricciones, aquellas que afectan a elementos que se encuentran también en el cliente son candidatas a poder ser aplicadas en él. Pero como se muestra en la Fig 5, algunas de esas restricciones pueden estar afectando tanto a elementos que se encuentran en el cliente como a otros que están únicamente en el servidor. Esto hace que sea necesario valorar cada restricción individualmente y analizar las posibilidades que tiene de ser evaluada en el cliente. A continuación, se detallan los distintos escenarios posibles.

Restricciones aplicables en el cliente sin modificaciones.

Algunas de las restricciones aplicadas sobre el modelo del servidor podrán ser aplicadas también en el cliente sin necesidad de realizar ninguna modificación. Esto sucede si todos los elementos afectados por la restricción que se ha definido para el servidor existen también en el cliente. Por ejemplo, en la Fig 5 la restricción que afecta a la clase D, y la que afecta a las clases C y F podrían ser candidatas a ser aplicables en el cliente.

No obstante, este es solo el primer requisito para que la restricción pueda evaluarse localmente. Es necesario tener además otras consideraciones. Además, el grafo de instancias del cliente debe garantizar la existencia de todos los elementos necesarios para realizar la evaluación.

Por ejemplo, la restricción *No puede haber dos Empleados con el mismo DNI* podría ser candidata a evaluarse localmente si la clase *Empleado* existe tanto en el cliente como en el servidor. No obstante, solo podrá evaluarse de forma totalmente independiente del servidor si el cliente dispone de todas las instancias de tipo *Empleado* en su grafo de objetos. Si solo dispone de algunas instancias, y el resto se encuentran en el servidor, sería necesario o bien solicitar las restantes al servidor, o bien delegar la evaluación enteramente a él.

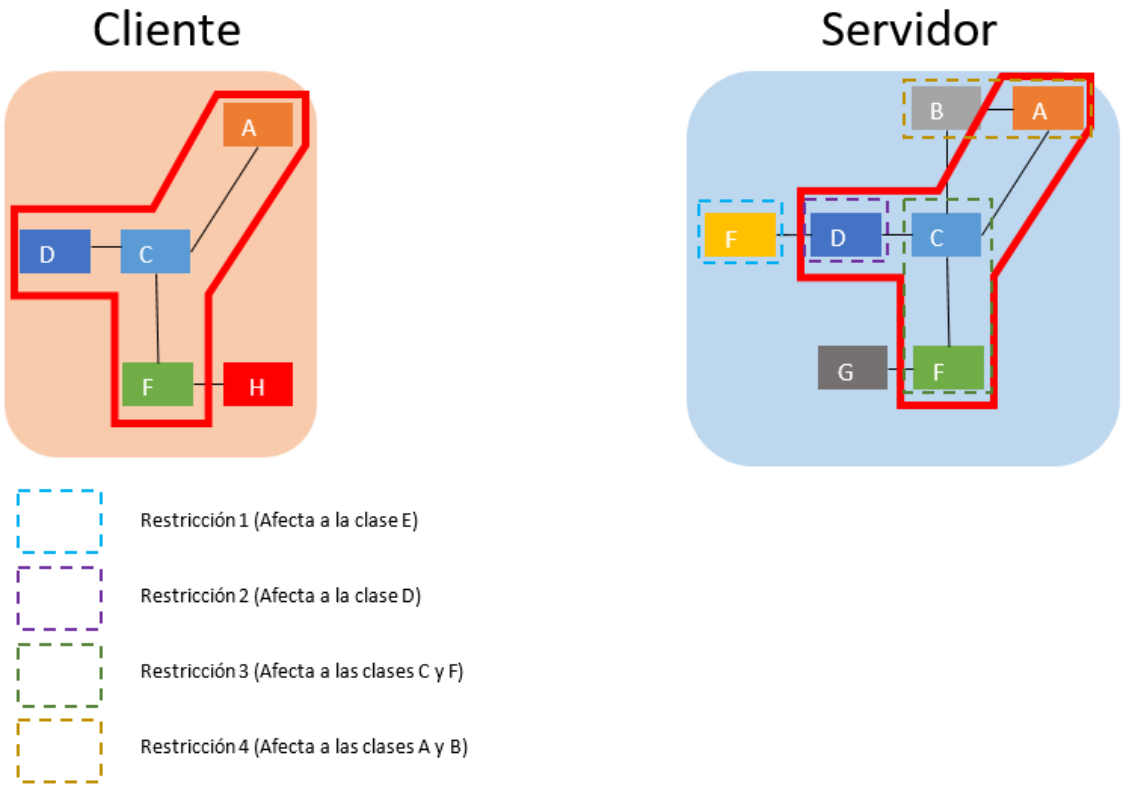


Fig 5: Algunas restricciones afectan a elementos que también están en el cliente.

Se puede ver que, incluso en los casos en los que aparentemente el modelo de dominio del cliente contiene todo lo necesario para evaluar una restricción, identificar si puede aplicarse al cliente o no requiere un nivel de análisis sobre el modelo de dominio y las funcionalidades del cliente que no son en absoluto triviales.

El siguiente escenario analiza qué se puede hacer con aquellas restricciones que no pueden evaluarse de forma totalmente independiente.

Restricciones aplicables en el cliente con modificaciones

Es posible que existan restricciones que, tal y como han sido formuladas para servidor, no son aplicables localmente, pero pueden ser utilizadas como base para la creación de otras nuevas adaptadas para el CDM.

En la Fig 5 se puede observar que la restricción que afecta a las clases A y B en principio no podría aplicarse al cliente, dado que carece de la clase B. Pero dependiendo de cómo esté formulada la restricción, quizás podría ser posible quedarse solo con la parte de la restricción que implica a la clase A, presente en ambos extremos.

Por ejemplo, la restricción "*Los Empleados deben tener al menos 18 años y disponer de una acreditación de manipulador de alimentos*" no sería aplicable en un cliente que contiene la clase *Empleado* pero no la clase *Acreditación*. Sería posible sin embargo tomar únicamente la parte de la restricción que afecta a la clase

Empleado e ignorar la otra, para aplicarla en el cliente como "*Los Empleados deben tener al menos 18 años*", para al menos ser capaces de evaluar ese aspecto localmente.

De nuevo, para tomar estas decisiones es necesario analizar cada restricción de forma individual e interpretar el modelo de dominio cuidadosamente.

Restricciones no aplicables en el cliente

Finalmente, algunas restricciones del servidor no tendrán relación alguna con el cliente. En la Fig 5 se puede observar como la restricción que afecta a la clase E nunca será evaluable localmente dado que ésta no existe en el cliente.

Además de casos triviales como el anterior, también pueden entrar en esta categoría restricciones que afectan a elementos existentes en el cliente, pero que en última instancia no es posible adaptar de forma efectiva para su evaluación local. Una restricción como *"Dos Empleados distintos no pueden disponer del mismo coche"* implica a las clases *Empleado* y *Coche*, pero en este caso no existe una forma de reformular ningún aspecto de la misma para su aplicación total o parcial en el cliente.

Aunque existen restricciones no aplicables fáciles de detectar, en los casos en los que algunos de los elementos que referencian existen en el cliente, es necesario analizar la restricción y comprobar si algunos aspectos de la misma son aplicables en el cliente antes de descartarla definitivamente, lo cual implica una tarea de análisis compleja y en la que es posible cometer errores.

Retos del análisis manual

El hecho de que los modelos de cliente y servidor estén muy relacionados no hace la aplicación de las restricciones del servidor en el cliente trivial, sino que en ocasiones plantea cuestiones difíciles de resolver.

Estos problemas se afrontan habitualmente con soluciones ad hoc, en las que el diseñador debe tomar todas las decisiones de un modo manual, realizando un análisis cuidadoso y asegurándose caso a caso que el modelo que está adaptándose al cliente es consistente con el del servidor, y que se pueden evaluar localmente la mayor cantidad de restricciones posible.

Este análisis no es sencillo dado que el diseñador debe evaluar información presentada de formas muy distintas, tanto visual en el caso de los modelos de clases UML, como la textual que describe las restricciones, habitualmente en forma de código OCL. Este tipo de código puede ser complejo de interpretar, debido a que su representación impide en ocasiones identificar de un vistazo a qué elementos del modelo está afectando una restricción determinada [26] [27] [28]. Todos estos factores hacen que el proceso sea complicado, y exige al diseñador ser muy meticuloso comprobando todos los casos posibles, algo que tiende a ser fuente de descuidos y equivocaciones.

A esto hay que añadir que si, como es habitual, hay diferentes equipos de trabajo para el desarrollo del cliente y el servidor, los problemas de coordinación entre equipos humanos pueden llevar a inconsistencias entre ambas partes. Este problema adquiere una dimensión aún mayor en la fase de mantenimiento del producto, dado que cualquier modificación en el servidor implicará una revaluación de las restricciones necesarias para el cliente [7]. Todos estos problemas hacen que el diseño y correcto mantenimiento de las restricciones sea una tarea muy compleja, tediosa, y propensa a errores, especialmente a medida que se avanza en el ciclo de vida del desarrollo.

3 EXPOSICIÓN DE OBJETIVOS

El desarrollo de clientes ricos robustos plantea retos en la etapa de diseño que a día de hoy son solucionados principalmente mediante soluciones ad hoc que consumen mucho tiempo y recursos, y son propensas a la introducción de errores.

El propósito de este trabajo es explorar las posibilidades a la hora de dar soporte y agilizar el desarrollo de este tipo de clientes mediante la propuesta de métodos para la automatización de aquellas tareas de diseño susceptibles de serlo.

3.1 Propuesta

Este trabajo propone un método que analiza la información ya existente en el modelo del servidor y sus restricciones, para utilizarla en la generación automática del modelo de dominio del cliente, al tiempo que detecta cuáles de las restricciones del servidor son aplicables localmente. Para ello, el único trabajo requerido por parte del diseñador es identificar de qué clases son las instancias del grafo del servidor que necesitarán ser enviadas al cliente durante las comunicaciones cliente-servidor.

Dada la limitación descrita en la sección 1.1 por la que en determinados casos no será posible detectar si una restricción es aplicable al cliente de forma totalmente automatizada, se propone además un mecanismo de generación automática de documentación que asista al diseñador en el análisis y toma de decisiones.

Esta solución está orientada a escenarios en los que la funcionalidad del cliente está estrechamente relacionada con el modelo de dominio del servidor, planteando un modelo de desarrollo en el que inicialmente se define el modelo del servidor al completo y es utilizado para automatizar el diseño del cliente.

3.2 Objetivos

Asumiendo la existencia previa de un modelo de dominio para el servidor, se plantean los siguientes objetivos:

— La creación del modelo para el cliente estará basada en información existente y será automatizable en la medida de lo posible.

— Las partes del proceso que no sean automatizables recibirán soporte mediante la generación automática de documentación que clasifique las restricciones y asista al diseñador en la toma de decisiones.

— La propuesta debe ser implementable en plataformas o tecnologías existentes, sin forzar la elección de una concreta.

La creación del modelo para el cliente estará basada en información existente y será automatizable en la medida de lo posible

El método propuesto se basará en el modelo descrito para el servidor, y una selección inicial de las clases que el diseñador identifica como necesarias en la comunicación con el cliente. Una vez hecho esto, el método deberá ser capaz de generar un modelo de dominio para el cliente consistente con el del servidor. Si en la selección el diseñador omite clases o relaciones fuertemente dependientes con el resto de las elegidas e importantes para la consistencia del modelo, se deberán detectar y añadir automáticamente, aunque el diseñador no las estimase como necesarias inicialmente.

Todo este proceso deberá definirse basándose en reglas objetivas que puedan aplicarse de forma automatizada. El método debe ser compatible con las herramientas existentes de modelado para su ejecución automática.

La información del cliente generado se usará junto a la del servidor para detectar automáticamente cuáles de las restricciones del servidor son también aplicables al cliente. No obstante, solo se incorporarán automáticamente al cliente aquellas para las que esté garantizado que se puedan verificar de forma independiente.

Las partes del proceso que no sean automatizables recibirán soporte mediante la generación automática de documentación que clasifique las restricciones

Como se ha explicado anteriormente, existen algunos escenarios en los que no será posible detectar de forma automatizada si una restricción puede aplicarse al

cliente o no. En este contexto, la toma de decisiones respecto a esas restricciones recae en el diseñador.

En estos casos, se proporcionará información adicional para asistirle en esta tarea. Esta documentación se generará automáticamente en base a la información ya existente en el modelo de clases y el código de las restricciones, y estará formada por métricas cuantificables y objetivas.

Disponiendo de estas métricas, se propondrá además una clasificación automática de las restricciones que permita al diseñador analizar el problema de una forma estructurada.

El código de las restricciones OCL utiliza como base la información existente en el modelo de clases UML. Esto implica que para comprender qué elementos afectan no basta con la restricción, sino que es necesario considerar los atributos, clases y relaciones presentes en el modelo.

Para facilitar esta tarea, se propondrá una representación visual de las restricciones que agilicen la tarea de comprender los elementos afectados por una restricción y la complejidad de la misma. Este tipo de información es útil a la hora de entender si es aplicable al cliente en los casos donde el método automatizado no llega, especialmente considerando que realizar este análisis no es intuitivo para el diseñador si solo dispone del código de la restricción.

Esta representación debe ser generable automáticamente en base a la información existente en el código de la restricción y el modelo de clases UML.

La propuesta debe ser implementable en plataformas o tecnologías existentes, sin forzar la elección de una concreta

La propuesta evitará imponer metodologías de desarrollo adicionales al desarrollador. Los modelos producidos como resultado de la aplicación de este método deberían ser útiles independientemente de la metodología elegida, siguiendo el principio de no intrusividad. Aunque los modelos producidos podrían integrarse con técnicas y herramientas de diseño dirigido por modelos, el alcance del trabajo se limita a la generación de los modelos de una forma agnóstica.

Asimismo, para evitar la dependencia con tecnologías específicas para la implementación de esta propuesta, la descripción de la solución se proporcionará formalmente para que ésta pueda ser aplicada sobre cualquier tecnología.

Hoy en día el estándar de facto a la hora de representar modelos de dominio es UML, con el lenguaje OCL para la definición de las restricciones. La formalización tendrá especialmente en cuenta las características de estos estándares y se basará principalmente en ellos, pero aplicando principios e ideas de carácter generalista de forma que puedan ser adaptados a otras propuestas de modelado alternativas, así como a otros lenguajes de definición de predicados.

Aunque la propuesta se realizará de un modo agnóstico e independiente de cualquier plataforma tecnológica, los principios descritos en este trabajo deberán ser implementables en las tecnologías existentes. Por ello, a modo de demostración, se desarrollará un prototipo que cumpla los objetivos descritos en este apartado y que permita demostrar la viabilidad de la propuesta.

4 TÉCNICAS ACTUALES PARA EL DISEÑO Y DESARROLLO DE CLIENTES RICOS

Las propuestas tecnológicas para desarrollar aplicaciones más interactivas en el navegador web han evolucionado con el tiempo, con propuestas como Adobe Flash[29], Flex[30], Microsoft Silverlight [31], Java Applets [32] o JavaFX[33]. Aunque este tipo de tecnologías fueron las primeras en popularizar el concepto de cliente rico en la Web, más adelante, el avance de los estándares web combinados con Javascript y Ajax fue ganando popularidad y desplazando el uso de plugins. Paralelamente, la popularización de los dispositivos móviles conectados a internet, en los que las aplicaciones nativas hacen uso del protocolo HTTP para conectarse a servidores web, hacen que las plataformas de desarrollo de aplicaciones móviles sean unas de las más utilizadas a la hora de desarrollar clientes ricos.

A continuación, se describirán las principales prácticas a la hora de desarrollar este tipo de clientes.

4.1 Evolución de la tecnología de desarrollo de clientes ricos en La Web

En sus inicios, los primeros sitios web eran simples documentos que presentaban información al usuario, y apenas proporcionaban interacción más allá de navegar

hacia otra página web. El servidor se limitaba a almacenar estas páginas, y el navegador (el cliente) se limitaba a pedirlas al servidor cuando el usuario navegaba de una página a otra.

En 1993 se comenzaron a utilizar los primeros scripts CGI en el lado del servidor que permitieron añadir lógica y crear sitios web dinámicos. El servidor podía generar dinámicamente documentos HTML específicos en función de las peticiones del usuario, dando lugar a una web más interactiva y participativa. En este escenario, toda la lógica era ejecutada por el servidor, y el cliente se limitaba a recibir una vista con la interfaz y los datos correspondientes. Cada nueva interacción realizaba una nueva petición al servidor, que tras realizar las tareas correspondientes enviaba una nueva vista actualizada al cliente.

La popularización de las primeras tecnologías destinadas a ofrecer clientes ricos para la Web estaban basadas en tecnologías propietarias, con los Applets de Java [32] y los componentes ActiveX de Microsoft [34]. Los applets de Java surgían en 1995 y se trataban de *plugins* dependientes de la máquina virtual de java, mientras que los componentes ActiveX se anunciaban en 1996, siendo una evolución de los componentes COM y DCOM de Microsoft. En este tipo de soluciones, el cliente se descargaba las aplicaciones a través de la web, y una vez cargada esta se visualizaba incrustada en la página web.

También en 1995 el navegador Netscape introdujo el lenguaje Javascript como un mecanismo para crear páginas HTML más dinámicas mediante scripts locales que permitían modificar la visualización de la página y añadir lógica a la misma [35].

Macromedia Flash surgió en 1996 y gozó de una gran popularidad. Surgido inicialmente como un plugin para realizar animaciones y visualizar vídeo, la inclusión de su propio lenguaje de scripting (ActionScript) popularizó también su uso para el desarrollo de clientes ricos [29].

En sus primeras etapas, las posibilidades técnicas que ofrecía el uso de Javascript en conjunto con HTML eran aún muy limitado, por lo que a la hora de desarrollar clientes ricos el uso de plugins era la norma. En 1998 con la introducción de la tecnología AJAX (aunque este término no se acuñaría hasta más adelante) que permitía realizar peticiones asíncronas desde el navegador sin la necesidad de usar plugins, fue posible realizar clientes ricos mediante el uso de HTML y Javascript. Sin embargo, la falta de estandarización, la escasa optimización de los motores de Javascript y de manipulación del DOM, y la competitividad de las plataformas de desarrollo de plugins hicieron que el uso de estos estándares no se popularizara hasta más tarde [36].

En 2007, mientras la tecnología de Flash Player mantenía aún una gran popularidad, Microsoft presentó la plataforma Silverlight basada en .NET, que pretendía proporcionar un modelo de desarrollo más sólido y con más posibilidades que el proporcionado por los estándares web del momento, al

tiempo que pretendía resolver las limitaciones y problemas de los plugins existentes [31].

La tendencia en el uso de plugins comenzó a declinar lentamente con la aparición de nuevos navegadores. Estos crearon un mercado más competitivo que impulsó la consolidación y avance de los estándares web, así como la optimización de los motores de Javascript en una carrera por disponer de navegadores más rápidos [37]. Esto, unido al auge de los primeros smartphones, con capacidades más limitadas que se veían lastrados por el uso de plugins pesados como Flash o Silverlight, hizo perder popularidad de estos en favor de un desarrollo basado en estándares web. En la actualidad la mayoría de clientes ricos para el navegador web se implementan utilizando HTML5 y Javascript, a menudo con la ayuda de alguno de los numerosos *frameworks* de desarrollo existentes para esta plataforma.

Esta tecnología ha permitido que en los dispositivos de escritorio haya habido un uso cada vez mayor de aplicaciones web basadas en clientes ricos para la realización de tareas que tradicionalmente han sido soportadas por aplicaciones de escritorio. Clientes de correo online, calendarios o incluso aplicaciones online de edición de documentos son claros ejemplos de esta tendencia. Las principales empresas tecnológicas ofrecen todo tipo de servicios relacionados con la productividad y la colaboración, como Google (Gmail, Google Calendar, Google Docs) o Microsoft (Outlook365).

En el ámbito de los dispositivos móviles, aunque las principales empresas suelen ofrecer clientes ricos basados en estándares web adaptados al navegador móvil, la principal forma de publicar este tipo de servicios es a través de aplicaciones nativas. Estas aplicaciones se ejecutan con más fluidez en los dispositivos móviles, pueden hacer un uso mayor de sus características, son más sencillas de desarrollar y las tiendas online de aplicaciones hacen fácil su distribución, instalación y monetización.

Aunque tradicionalmente el termino cliente rico en este contexto se ha asociado a aplicaciones web ejecutadas en un navegador, las aplicaciones nativas para dispositivos móviles comparten sus mismas características. La mayoría de ellas además se basan también en una arquitectura cliente-servidor, donde habitualmente la tecnología utilizada en el lado del servidor es la misma que se utilizaría para una aplicación web, y los métodos de comunicación siguen basándose en los mismos protocolos. A efectos de desarrollo, es habitual que un mismo servidor reciba peticiones de aplicaciones nativas y de clientes basados en estándares web para navegadores.

Las tecnologías que han dado soporte a este tipo de clientes han variado con el tiempo, y otras nuevas surgirán en el futuro. No obstante, las dificultades que plantea su desarrollo identificadas en este trabajo aún continúan vigentes.

4.2 Distribución de responsabilidades entre cliente y servidor

La primera dificultad encontrada al plantear el desarrollo de un cliente rico es determinar cómo se distribuirán las responsabilidades entre cliente y servidor.

En los inicios del desarrollo de este tipo de clientes, la mayor parte de la lógica de negocio permanecía en el servidor, mientras que el cliente se limitaba a mostrar la información enviada por éste. El cliente solo explotaba su capacidad de procesamiento local para realizar visualizaciones gráficas, animaciones y presentaciones más efectivas. Cada modificación era inmediatamente enviada al servidor para persistirla, y cada nueva acción realizaba algún tipo de petición al mismo. Esta aproximación ignoraba dos posibilidades: la capacidad de manejar y persistir un modelo de datos local, y la posibilidad de tener una lógica propia que trabajase con dicho modelo [3].

Respecto a la persistencia del modelo, se pueden presentar las siguientes aproximaciones:

— No persistente en el cliente: el cliente descarga un modelo de datos durante su conexión con el servidor, pero al terminar la sesión este estado no se almacena.

— Persistente en el cliente: el cliente mantiene el estado del modelo de datos localmente.

— No persistente en el servidor: cuando el cliente se conecta al servidor este crea un modelo de datos que se descarta cuando finaliza la sesión.

— Persistente en el servidor: el servidor posee un modelo de datos cuyo estado se mantiene.

Estas alternativas obligan a tomar decisiones acerca de cómo mantener la consistencia de los datos entre cliente y servidor para cada una de las combinaciones, y cuándo se debe realizar la sincronización en caso de que se produzcan cambios en una de las dos partes.

Respecto a la lógica de negocio, de nuevo existen varias posibilidades:

— Ubicar toda la lógica de negocio en el cliente: el servidor actúa únicamente como un sistema de persistencia al que el cliente realiza consultas y envía datos.

— Ubicar toda la lógica de negocio en el servidor: el cliente actúa únicamente como una capa de presentación.

— Mixta: tanto cliente como servidor poseen lógica de negocio, y pueden llevar a cabo tareas con los datos que poseen. Dentro de esta aproximación, las tareas también pueden distribuirse siguiendo el mismo esquema: tareas que se realizan completamente en cliente o servidor, o tareas divididas en varias partes que se distribuyen entre servidor y cliente.

Estas opciones plantean dificultades, y pueden influir sobre qué elementos del modelo son necesarios en el cliente dependiendo de las funcionalidades que deban llevar a cabo.

Ante estas decisiones, varios autores proponen tratar de maximizar las responsabilidades del cliente y así aprovechar sus beneficios [18][3][4]. También se sugiere iniciar el diseño de una forma global y ajena a la posterior división de responsabilidades entre cliente y servidor, para luego decidir cómo repartir estas responsabilidades[3].

Otros aspectos a considerar son que un servidor pueda servir a distintos tipos de cliente con diferentes objetivos y funcionalidades, así como la posibilidad de que un cliente pueda comunicarse con diferentes servidores totalmente independientes entre ellos, con un modelo de datos y funcionalidades que se sirve de diferentes fuentes.

4.3 Mecanismos de comunicación entre cliente rico y servidor

Dependiendo del modelo de distribución de responsabilidades entre cliente y servidor seleccionado para el sistema existirán diferentes necesidades respecto a la comunicación.

Un cliente rico con su propio modelo de datos y lógica de negocio tiene la posibilidad de realizar menos peticiones al servidor, pero implica decidir cuándo los datos entre ambos deben sincronizarse.

En el contexto actual, donde los dispositivos móviles son ubicuos, las aplicaciones desarrolladas para estos son clientes ricos que pueden verse afectados por una conectividad intermitente. Aunque las conexiones ofrecidas por los operadores móviles tienen cada vez más cobertura y calidad, y el acceso a redes wifi es habitual, no son infrecuentes las ocasiones en las que la conexión no está disponible temporalmente.

Por este motivo, es ideal que el cliente sea capaz de realizar una serie de modificaciones que se acumulen sobre el modelo local sin necesidad de notificar cada una de ellas individualmente al servidor. Respecto a los medios para sincronizar la comunicación, se pueden plantear tres mecanismos principales [20]:

— Paso de mensajes: El cliente envía mensajes al servidor y espera su respuesta. En caso de que la conexión no esté disponible, éstos se encolan en el cliente y se espera a poder enviarlos al servidor. En esta aproximación, mientras se espera la respuesta del servidor, no se pueden acumular más cambios al modelo local hasta que el servidor haya sincronizado los anteriores.

— Replicación del estado: Se trabaja con un modelo local que se modifica sin preocuparse de si es posible sincronizarse con el servidor o no. Cuando

hay una conexión disponible, se notifican los cambios al servidor. Cuando no hay conexión, las modificaciones en el modelo se realizan localmente, y son registradas en orden para notificarlas al servidor posteriormente.

— Repetición de métodos: En este escenario, cliente y servidor disponen de los mismos métodos para modificar el modelo. Cada vez que el modelo local se modifica a través de un método, se notifica al servidor qué método y parámetros se han usado, que ejecuta el mismo método. Cuando no hay conexión disponible, estos métodos se encolan, para ser reproducidos en orden posteriormente. El resultado es similar al de la replicación de estado, pero su implementación no necesita hacer un seguimiento de las modificaciones del estado, sino de los métodos que se ejecutan. Puede implicar menos tráfico de datos, aunque en determinadas circunstancias puede requerir mayor cantidad de procesamiento en el servidor.

Para que estos mecanismos sean efectivos, los modelos de cliente y servidor han de estar correctamente diseñados, y los elementos en común de ambos serán los que definan qué posibilidades hay a la hora de comunicarse y sincronizar la información.

4.4 Características de un cliente rico robusto

Para asegurar la integridad del sistema no es suficiente que cliente y servidor tengan formas apropiadas de comunicarse, ambos deben velar por realizar las modificaciones sobre sus respectivos modelos de una forma apropiada, y deben permitir al usuario continuar su trabajo ante posibles eventualidades. Un cliente rico robusto debería ser capaz de continuar su funcionamiento ante:

— Introducción de valores no válidos en el modelo por parte del usuario, o tras interacciones de usuario incorrectas o inesperadas.

— Errores inesperados en el funcionamiento de la aplicación.

— Pérdida de conexión con el servidor.

— Errores en la comunicación con el servidor. Envío o recepción de datos corruptos o manipulados por intermediarios.

Si el cliente realiza modificaciones que el modelo del servidor no va a aceptar, el trabajo realizado en el cliente habrá sido en vano, especialmente en los casos en los que se han acumulado varias modificaciones antes de notificar al servidor.

Cuando un cliente tiene libertad para operar con su modelo local comunicándose solo esporádicamente con el servidor, las consecuencias de que éste rechace sus cambios por no respetar la integridad del modelo se hacen más notables para el usuario, que puede perder una mayor cantidad del trabajo realizado. Por otro lado, una aproximación en la que se consulta más a menudo

al servidor tiende a garantizar más la integridad, pero limita la interactividad y fluidez que el cliente ofrece al usuario.

Verificación local de restricciones

Tener en cuenta las restricciones que se han definido para el modelo del servidor es fundamental cuando se diseña el cliente, ya que permiten definir los límites de las modificaciones locales que son posibles sin que causen problemas al sincronizarse con el servidor. Como se ha mencionado en capítulos anteriores, es posible mantener un cliente con una independencia razonable del servidor y que al mismo tiempo mantenga la integridad con este si trata de realizar la mayor cantidad posible de verificaciones localmente [7].

Respecto a las restricciones que afectan a elementos del modelo que se repiten en cliente y servidor, no es viable ubicarlas únicamente en uno de los dos modelos, sino que deben replicarse en ambos. El servidor tiene la responsabilidad de verificar siempre las actualizaciones que reciba de los clientes que se le conecten, con independencia de que ellos ya hayan realizado una verificación. Esta aproximación es necesaria por motivos de seguridad [7], ya que es posible que el cliente haya visto comprometida su propia seguridad, sus funcionalidades hayan sido modificadas, o la comunicación haya sido interceptada y modificada en algún punto intermedio de la comunicación. También es posible que clientes de diferente naturaleza sean desarrollados con independencia del servidor, sin garantías de que vayan a verificar restricciones localmente.

Esta necesidad de mantener comprobaciones en ambos extremos conlleva dificultades en el caso de restricciones que tienen en cuenta varios elementos, algunos de los cuales sólo se encuentran en el servidor. En la actualidad, todas estas decisiones se llevan a cabo manualmente, y el diseñador debe reinterpretar dicha restricción para que las partes que afectan a los elementos del cliente puedan validarse localmente.

5 DEFINICIÓN DE MODELOS DE DOMINIO Y RESTRICCIONES

Cualquier software es el producto del código que define su comportamiento, escrito en cualquiera de los lenguajes de programación existentes y ejecutándose en una determinada plataforma. Cuando se implementan productos software de gran envergadura o complejidad, su código es igualmente extenso y complejo, y es necesario abordarlo siguiendo una metodología propia de la ingeniería del software.

Uno de los pilares fundamentales para esto es el uso de modelos de dominio [38]. Se trata de representar una parte del conocimiento que tienen los expertos en el problema (dominio) que se trata de resolver, realizando una abstracción que representa las partes relevantes del sistema a desarrollar [39]. Este modelo representa los conceptos principales y cómo estos se relacionan desde el punto de vista de su creador, siendo útil para documentar el problema o una posible solución al mismo. Un modelo de dominio bien diseñado ayuda al entendimiento, reflexión y comunicación de los aspectos esenciales del negocio.

La idea principal es que el modelo surge del conocimiento de los expertos en el dominio, y es continuamente refinado a lo largo del ciclo de desarrollo. El código de la implementación final trata de plasmar este modelo de forma fiel al diseño, que a su vez refleja el modelo mental de los expertos en el negocio, ejerciendo de eje de unión que comparten todos los participantes en el proyecto.

5.1 Modelos de clase UML el diseño de modelos de dominio

El uso de modelos de dominio es una práctica asentada en la industria, y ha evolucionado de la mano de los lenguajes de orientación a objetos y tecnologías relacionadas. El paradigma de la orientación a objetos ha demostrado su utilidad

a la hora de resolver aplicaciones de negocios y cuenta con multitud de técnicas, patrones, métodos y enfoques bien establecidos.

Un modelo de dominio se puede representar de múltiples formas, con descripciones textuales, diagramas informales, diagramas Entidad-Relación, etc. Sin embargo, a la hora de diseñar un modelo de dominio, el uso de los conceptos de la orientación a objetos resulta muy adecuado, ya que permite representar los aspectos del modelo en términos directamente trasladables al lenguaje en que se implementarán. Así, el estándar de facto en la industria es el uso del *Unified Modeling Language* (UML) como lenguaje gráfico para la expresión de modelos orientados a objetos. Aunque UML permite representar diversas vistas del modelo subyacente mediante distintos tipos de diagramas, el de dominio es habitualmente representado por diagramas de clases.

Esta aproximación es ventajosa porque permite plasmar un modelo mental del usuario en elementos visuales del lenguaje. En este estándar, los conceptos se representan con clases, y las propiedades de estos conceptos se definen con atributos. Permite definir diferentes tipos de relaciones entre los conceptos del modelo (clases), como asociaciones, generalizaciones y composiciones, con diferentes significados semánticos.

El modelo incluye además en las clases los métodos (funciones) que definen su comportamiento. Esto permite encapsular datos y comportamiento que hacen que el modelo de dominio incluyan la lógica de negocio, siendo, por tanto, más que simples modelos de datos [40] como puede suceder en otros sistemas basados en diagramas.

Finalmente, cuando los conceptos de un modelo de dominio se expresan en un diagrama de clases UML, estos pueden implementarse de una forma natural utilizando un lenguaje orientado a objetos. Si tanto el modelo como la implementación se realizan basándose en los conceptos propuestos por la orientación a objetos, ambas representaciones están claramente relacionadas, y los distintos miembros involucrados en el desarrollo tienen un lenguaje común que les permite comunicarse con precisión sobre los diferentes elementos del modelo.

Tipos de relaciones en modelos de clases UML

El método que se propone en este trabajo tiene en consideración los distintos tipos de relaciones a la hora de generar el modelo del cliente, por lo que es interesante detallar las características de las mismas. UML permite establecer varios tipos de relación: asociación, agregación, composición, herencia, y dependencia. Estas relaciones sirven para indicar cómo una instancia de una clase se puede comunicar y relacionar con instancias de otras clases. Cada uno de estos tipos de relación tienen una semántica diferente, que condiciona el modo en que se puede producir esta comunicación, y en algunos tipos de relación cada

una de las instancias relacionadas puede adoptar un rol diferente. Las relaciones existentes en UML son las siguientes:

- Asociación: Una asociación indica una relación estructural que describe una conexión entre objetos, y les permite comunicarse y colaborar. No se trata de una relación fuerte, ya que el ciclo de vida de un objeto no depende del otro. Esta relación puede ser bidireccional, es decir, que ambos objetos pueden comunicarse entre ellos, o tener una navegación unidireccional, en la que uno de los objetos puede comunicarse con el otro, pero no al revés.
- Composición: La composición permite establecer una relación de tipo parte/todo. Se cataloga como relación fuerte, en la que el ciclo de vida de las partes está ligado al ciclo de vida del todo.
- Agregación: La agregación, de forma similar a la composición, permite establecer una relación del tipo "tiene un". En este caso no se trata de una relación fuerte, ya que los ciclos de vida de los objetos no están ligados. Se suele considerar que la agregación es simplemente una forma de interpretar una asociación, sin ningún efecto práctico sobre el modelo, más allá de permitir al diseñador añadir un matiz a la interpretación de la asociación[41]. En esta Tesis, hablaremos únicamente de relaciones de asociación, considerando que las agregaciones son asociaciones a todos los efectos.
- Generalización / Especialización: Una relación de especialización entre dos clases establece una relación con dos roles, una clase padre y una clase hija, que también puede entenderse como una clase general y otra especializada. En este tipo de relación, se entiende que la clase hija es un subtipo de la clase padre, y posee las mismas características que esta, pudiendo añadir otras adicionales.
- Dependencia: La dependencia es el tipo de relación más débil de todas, e indica que en alguna de las operaciones de una de las clases se hace uso de instancias de la otra.

Además, algunos tipos de relación permiten definir cardinalidades en cada extremo, indicando con cuantas instancias de la otra clase se puede relacionar, algo también relevante a la hora de analizar las restricciones del modelo.

5.2 Definición de restricciones con OCL

OCL (Object Constraint Language) es un lenguaje formal de definición de restricciones. Fue definido por el OMG (Object Management Group) y forma parte del estándar UML desde su versión 1.1. El último estándar publicado para este lenguaje es el 2.4.1 [42].

Los modelos de dominio expresados mediante diagramas UML tienen ciertas limitaciones que no permiten expresar las restricciones del modelo de una forma precisa. OCL permite expresar restricciones sobre modelos UML existentes. Aunque la razón principal de su creación era la definición de restricciones, el lenguaje permite definir varios tipos de expresiones sobre un modelo de objetos (consultas, inicializaciones, definir propiedades derivadas, reglas de negocio, …).

Las expresiones OCL permiten hacer uso de la lógica de primer orden y la teoría de conjuntos. Sin embargo, la sintaxis se aleja de las expresiones formales matemáticas, y es más cercana a la de los lenguajes de programación modernos. Esto pretende hacer el lenguaje más familiar para los programadores, que, en última instancia, son los responsables de implementar estas expresiones en un lenguaje de programación.

En OCL las expresiones tienen tres partes: el contexto, el tipo de expresión y el cuerpo.

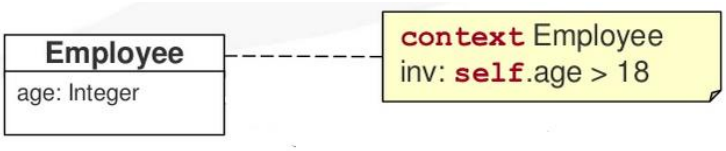


Fig 6: Partes de una restricción en OCL: contexto (Employee), tipo de restricción (inv) y cuerpo (self.age > 18).

Las expresiones están siempre definidas en el contexto de un tipo (un *Classifier* de UML): una clase, un atributo, un método, etc.; llamado el "tipo contexto" (*Employee* en la Fig 6). Los tipos contexto provienen del modelo UML al que se complementa.

La expresión que establece la restricción es denominada el cuerpo de la restricción (*self.age > 18*). En el caso de una expresión de tipo invariante (*inv*), el cuerpo es siempre una expresión booleana y debe ser satisfecha por todas las instancias del tipo contexto.

Desde el cuerpo de una expresión OCL siempre se puede acceder a la variable *self*. Esta variable representa la instancia raíz de la expresión, un objeto de la clase definida en el contexto de la expresión. Se puede navegar desde una instancia hacia sus relaciones y atributos utilizando el operador punto ".", de forma similar a como se realiza en algunos lenguajes de programación orientados a objetos.

```
context Person inv: self.partner.job.salary > 0
```

Ejemplo de código 1: Ejemplo de restricción OCL.

Los tipos de expresión que soporta OCL son:

— Invariantes sobre clases y tipos en un modelo de clases: *inv*.

```
context Company inv: self.noEmployees <= 50
```

Ejemplo de código 2: Ejemplo de invariante en OCL.

— Pre y poscondiciones sobre métodos: pre y post.

```
context Employee::income(): Integer
          pre: self.age >= 18
          post: result < 5000
```

Ejemplo de código 3: Ejemplo de pre y poscondiciones en OCL.

— Declaraciones de expresiones para ser reusadas en otras expresiones OCL: *def*.

```
context Employee def: hasTitle(t: String): Boolean =
      self.degrees->exists(title = t)

context Employee inv:
      self.job = Job::ADMINISTRATOR
      implies
      self.hasTitle( 'ECONOMY' )
```

Ejemplo de código 4: Ejemplo de definición de expresiones en OCL.

— Declaraciones de variables y su valor para ser usadas en una expresión: *let*.

```
context Person inv:
      let numberJobs: Integer = self.job->count() in
      if isUnemployed then
        numberJobs = 0
      else
        numberJobs > 0
      endif
```

Ejemplo de código 5: Ejemplo del uso de la expresión let en OCL.

— Definiciones de invariantes sobre valores iniciales y derivados: *init y derive*.

```
context Person::isMarried: Boolean
    init: self.isMarried = false
```

```
context Company::noEmployees: Integer
    derive: self.employees->size()
```

Ejemplo de código 6: Ejemplos de init y derive en OCL.

— Definiciones del cuerpo de métodos de consulta utilizando *body*.

```
context Person::income(): Integer
     body: self.job.salary->sum()
```

Ejemplo de código 7: Ejemplo de métodos de consulta en OCL.

Las expresiones OCL hacen referencia a los elementos definidos en el modelo de clases UML, pero se evalúan contra el modelo de instancia (grafo de

objetos) que existirá en tiempo de ejecución. Este modelo de instancia estará derivado del modelo de clases que ejerce como metamodelo de este.

5.3 Herramientas de soporte a OCL

En la actualidad existen varias herramientas tanto en la industria como en el ámbito académico que dan soporte a OCL. Estas herramientas facilitan de diversas formas tanto el uso de este lenguaje como la implementación de las restricciones definidas con él. Algunas de las áreas a las que dan soporte son:

- Edición: proveen validación sintáctica y sintaxis realzada.
- Validación semántica: de las expresiones contra el modelo permitiendo detectar errores de nombres de clases, atributos, etc.
- Ejecución: permiten cargar un grafo de instancias y chequear sobre él todas las restricciones para validarlo. Estas comprobaciones se hacen contra todas las instancias de objetos del modelo.
- Traducción de expresiones OCL a lenguajes de programación (Java, por ejemplo): lo que permite ejecutar las validaciones con mejor eficiencia que en el punto anterior, ya que es código compilado frente a un intérprete. Esta facilidad permite también completar el ciclo en procesos MDD o MDA ya que el modelo y sus restricciones se terminan traduciendo a código ejecutable final.
- Traducción a SQL: En aquellas aplicaciones que se vayan a dedicar a sistemas de información que usen una base de datos relacional, se puede generar el esquema de tablas a partir del UML y traducir a SQL las consultas expresadas en OCL.

Dentro del ámbito experimental y académico, algunas de las propuestas más relevantes son las siguientes [43]:

	Sintaxis/ edición	**Semántica**	**Ejecución**	**Java**	**SQL**	**Versión de OCL**
USE	Sí	Sí		Sí		2.0
Octopus	Sí			Sí		2.0
OCLE	Sí			Sí		2.0
Dresden OCL Toolkit	Sí	Sí	Sí	Sí	Sí	2.3
Oclarity	Sí					2.3
Eclipse OCL	Sí	Sí	Sí	Sí		2.4

Tabla 1: Herramientas de OCL y sus características.

Dentro de la industria se han desarrollado herramientas de diversa solidez para dar soporte a OCL, principalmente en la plataforma Eclipse. El proyecto Eclipse es la principal implementación de referencia de las especificaciones de OCL y dispone de varios proyectos para uso en modelado.

EMF (*Eclipse Modeling Framework*) [44] es el *framework* base de todos estos proyectos y da soporte a la definición de metamodelos estructurales y a la instanciación de estos en modelos. EMF también soporta la generación de código Java equivalente a los metamodelos. Este *framework* base se complementa con otros *frameworks*/proyectos en distinto estado de madurez. En ese entorno, OCL se convierte en un lenguaje de modelado que permite añadir validaciones a todos los metamodelos EMF. Permite expresar de forma compacta restricciones, consultas, inicializaciones y operaciones derivadas. Las últimas versiones del *framework* permiten hacer validación estática y semántica de las expresiones. En la actualidad permite:

- Editar: ya sea incrustando expresiones en el metamodelo, añadiendo restricciones en documentos adjuntos al modelo, o interactivamente a través de una consola.
- Ejecutar: en modo interprete, lo que permite interacción desde una consola o en modo compilado, traduciendo el código a Java.
- Depurar.
- Testear, verificar que los modelos resultado de haber aplicado transformaciones son compatibles con las restricciones que se les imponen.

La comunidad académica propone regularmente prototipos y mejoras para facilitar la adopción de este lenguaje, mientras que, en el ámbito de la industria, principalmente desde el proyecto Eclipse, las herramientas disponibles amplían gradualmente sus funcionalidades. Aunque esto evidencia la demanda de mejoras y soporte para el uso de este lenguaje, OCL dista aún de gozar de un uso generalizado. Aún hay un amplio margen de mejora tanto en cantidad de herramientas como en la madurez de las mismas para su uso en la industria, algo que puede ayudar a impulsar la definición de restricciones de una forma precisa sobre los modelos, y en consecuencia a promover prácticas más sólidas a la hora de diseñar software robusto.

6 UBICACIÓN DE RESPONSABILIDADES EN EL CLIENTE Y TÉCNICAS PARA DARLE SOPORTE

Los problemas relacionados con el manejo de modelos de dominio en cliente y servidor han sido identificados desde hace tiempo, y han tratado de afrontarse a través de diferentes enfoques.

En este capítulo se realizará un repaso de algunas de las técnicas relevantes para este problema, pasando por herramientas para la verificación local de restricciones, técnicas para la redistribución de las funcionalidades de una aplicación, modificaciones automáticas sobre modelos de dominio existentes, o manipulación de restricciones.

Estas propuestas sirven para dar una perspectiva de la relevancia del problema, a la vez que ofrecen inspiración a la hora de buscar la forma de afrontar los problemas presentados en esta tesis.

6.1 Técnicas para la verificación de restricciones en el cliente

Desde el comienzo de la popularización de los clientes ricos, se ha sugerido la necesidad de ubicar más responsabilidades en el cliente. Varios trabajos como los publicados por Zhang [21] o Leff y Rayfield [20] sugieren que las aproximaciones habituales para el desarrollo de clientes ricos no explotan su potencial al completo. Según estos autores, las arquitecturas más populares a menudo delegan demasiadas funcionalidades al servidor. Las consecuencias de esta

aproximación no sólo implican una peor experiencia de usuario, sino que además conllevan arquitecturas y ciclos de desarrollo demasiado complejos [21] o aplicaciones que no son capaces de responder apropiadamente en situaciones de baja conectividad [20]. Aunque en estos trabajos se propone el manejo local de la robustez y las restricciones, no se ofrece ningún medio específico para facilitar estas tareas.

En la industria existen soluciones para implementar las validaciones en el cliente, como las herramientas de validación de Struts [45], jQuery Validation Plugin[46] o Simfatic [47]. Estas herramientas, y otras similares, gozan de popularidad en el ámbito del desarrollo web, y son eficaces a la hora de validar campos de formularios antes de enviar la petición al servidor, y de este modo poder dar notificaciones instantáneas de error al usuario, pero no están diseñadas para enfrentarse a las complejidades de las reglas de negocio de una aplicación completa.

Se han realizado propuestas que tratan de cubrir algunas de las carencias de este tipo de herramientas. En el ámbito de los servicios web, Hallé y Vllenmarie [22] sugieren definir contratos que se evalúen localmente antes de realizar la petición al servidor. Estas reglas se describirían en un lenguaje que soporta la lógica de primer orden, y en el cliente se dispondría de un applet que evalúa estas reglas localmente antes de realizar la petición al servidor. En caso de que éstas no se cumplan, la petición no se realiza y se evita tanto que el servidor realice comprobaciones innecesarias, como que el cliente espere la respuesta de peticiones destinadas a fallar. Dado que Javascript es el principal lenguaje de desarrollo en el navegador, Heidegger y Thiemann [48] han propuesto métodos para definir pre y post condiciones en el mismo, integrando en él un lenguaje basado en atributos. La principal ventaja de estas propuestas es que ofrecen medios más sofisticados para ayudar al desarrollador a definir e implementar las restricciones del cliente. Sin embargo, la tarea de detectar y definir qué restricciones son necesarias para el cliente y cómo definirlas correctamente ha de realizarse manualmente, por lo que los problemas asociados a identificar qué restricciones del servidor son aplicables al cliente y cómo se hará, seguirían presentes.

Liang y Jianling [7] reconocen la necesidad de disponer de restricciones coordinadas entre cliente y servidor. Para ello han propuesto sistemas en los que las restricciones se definen en un fichero XML que se ubicará en el servidor, y permite manejar algunas de ellas en los formularios utilizados en el cliente. Estas evaluaciones locales están limitadas a campos de formularios (restricciones de atributo o de objeto). Esta aproximación automatiza la implementación de parte de las verificaciones del lado del cliente y facilita el mantenimiento. No obstante, los autores recomiendan explícitamente delegar siempre al servidor las restricciones más complejas, las de clase y dominio, por lo que las posibilidades de verificación en el cliente continúan limitadas en esta propuesta.

Schmidt, Stühmer y Stojanovic [49] proponen utilizar motores de reglas en el cliente para la verificación de restricciones. Las reglas para cliente y servidor se definen en un fichero que se ubicará en el servidor, y el motor aplicará el algoritmo RETE [50] para verificarlas. Este sistema permite definir reglas complejas que se evaluarán automáticamente en tiempo de ejecución, pero de nuevo se deben especificar manualmente las que se aplicarán en el cliente.

Louwsma et al. [6] identifican la importancia de la detección y clasificación de restricciones aplicables al cliente, de modo que se permita una edición robusta incluso cuando no hay conexión con el servidor. Proponen una metodología basada en UML y OCL para definir restricciones para clientes ricos para GIS (Geographic Information System, Sistema de Información Geográfica). Este trabajo propone aplicar las restricciones OCL para definir cómo se pueden editar los distintos elementos geográficos en el terreno, identifica diferentes tipos de restricciones en función de su complejidad, e incide en la utilidad de verificar restricciones localmente en el cliente. Sin embargo, la propuesta se limita a proponer una metodología sobre cómo definir estas restricciones, y en última instancia éstas deben identificarse e implementarse manualmente.

6.2 Transformaciones automáticas sobre elementos existentes

El problema de distribuir las funcionalidades de una aplicación que se encuentra dividida en varias partes que se comunican es común, y algunas propuestas tratan de afrontar estos problemas de una forma más automatizada.

Existen investigaciones centradas en distribuir automáticamente la lógica de dominio entre cliente y servidor. Una vez se ha construido una aplicación completa, su código se analiza y tratan de redistribuir automáticamente sus componentes entre cliente y servidor, con herramientas como J-Orchestra de Tilevich y Smaragdakis [51], Coign de Hunt y Scott [52], o la plataforma diseñada por Yang et al. basada en el lenguaje Hilda [53]. Estas soluciones están motivadas por algunos de los mismos problemas que se han identificado en este trabajo, pero están más centradas en la optimización de aplicaciones ya construidas. Esta optimización y redistribución de las funcionalidades de la aplicación se hace en función de diferentes criterios en cada caso, como por ejemplo tratar de optimizar el uso de memoria, adaptarse a la capacidad del hardware del cliente, o llevar al cliente las funcionalidades que más se utilizan dejando en el servidor las que tienen un uso esporádico. Todas estas aplicaciones se basan en *frameworks* o plataformas de desarrollo específicas, trabajan directamente sobre la aplicación una vez esta ha sido implementada y ninguna se centra específicamente en el problema de verificar localmente las restricciones.

Fuera del entorno del desarrollo de clientes ricos, se han propuesto técnicas para adaptar automáticamente restricciones OCL con diferentes propósitos. Hassam et al. [54] definen técnicas para mantener automáticamente la

consistencia de las restricciones OCL después de aplicar modificaciones al modelo de clases UML. Por cada modificación individual aplicada sobre el modelo, su herramienta identifica las restricciones OCL afectadas por dicha modificación, y decide si es necesario eliminarla (en caso de que ya no sea relevante), o si debe modificarse automáticamente para adaptarse el nuevo modelo. Cabot y Teniente [55] también han desarrollado técnicas para modificar automáticamente restricciones y modelos de dominio automáticamente para mejorar la eficiencia de la verificación de las restricciones. Para ello, proponen técnicas para simplificar las restricciones OCL, identificar qué operaciones activan la verificación de ciertas restricciones, y reformulan las mismas para que estás se verifiquen de forma más eficiente en función de las operaciones encontradas en el modelo.

Estas propuestas tratan de afrontar el problema de delegar en el diseñador la tarea de revisar las restricciones OCL existentes para obtener ciertos objetivos, cuando la información necesaria para ello ya se encuentra en el propio modelo UML. Aunque estas soluciones están diseñadas para resolver otro conjunto de problemas distintos a los propuestos en este trabajo, los principios puestos en práctica para identificar qué restricciones OCL necesitan revisión pueden ser útiles para los propósitos de este trabajo.

6.3 Model Slicing

La idea de dividir un modelo de dominio en distintas partes ha demostrado ser una aproximación útil para afrontar varias tareas, como identificar conjuntos de elementos interdependientes, una mejor visualización de las partes de un modelo, o evaluar subconjuntos de un modelo mayor y más complejo. Este tipo de técnicas se conocen como *Model Slicing* (Corte del modelo), un proceso de desglose para extraer e identificar partes del modelo relevantes, o elementos relacionados dentro de un modelo, en función de una serie de criterios de corte (*slicing criteria*) [56].

El proceso de *Model Slicing* consta de 3 etapas:

— Selección del criterio de corte: Un proceso de *model slicing* siempre comienza con un criterio de corte. El criterio de corte puede variar enormemente en función del propósito del proceso. Puede consistir en seleccionar algunos de los elementos iniciales del modelo original, obtener un modelo cuyos elementos cumplan una serie de propiedades, o que se ajuste a determinadas métricas.

— Extracción inicial de elementos del modelo: Se genera un submodelo inicial que cumpla el criterio de corte. Este submodelo solo debe incluir los elementos mínimos que cumplan el criterio de corte, no siendo necesario que sea la solución final.

— Refinamiento del modelo: El submodelo se refina añadiendo o eliminando elementos respecto a la extracción inicial. El submodelo producido se reevalúa respecto al criterio de corte, y se produce un submodelo nuevo. Este proceso se repite de forma iterativa hasta que el producto final se ajusta de la mejor forma posible al criterio de corte.

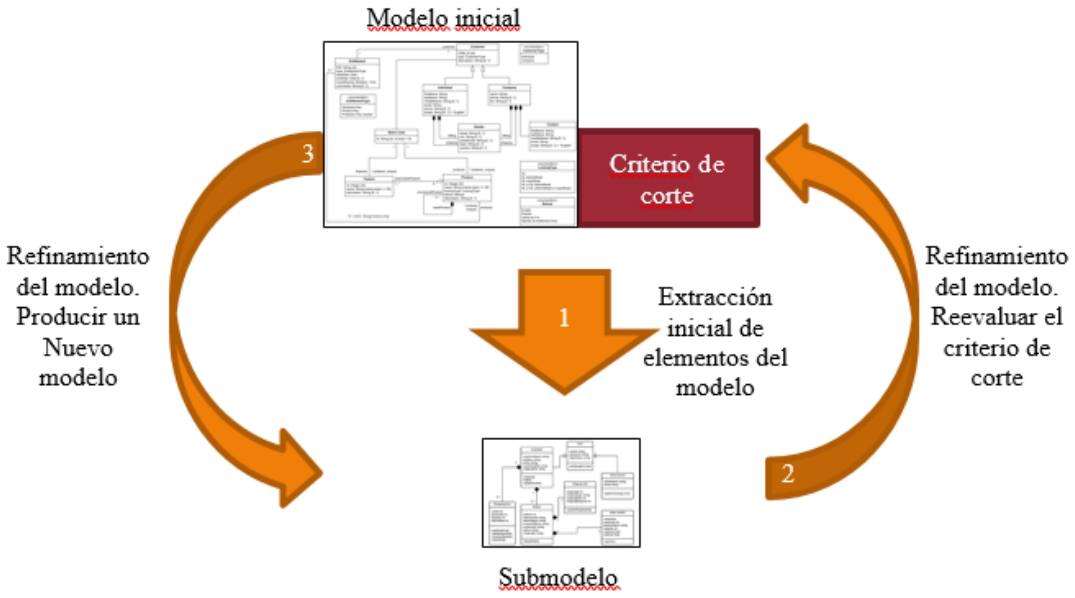


Fig 7: Proceso de Model Slicing.

Algunos trabajos proponen usar estas técnicas para ayudar en la visualización de modelos extensos y complejos. Tratan de identificar automáticamente diferentes subconjuntos del modelo que puedan ser presentados visualmente manteniendo suficiente cohesión, siguiendo criterios seleccionados por el usuario. El objetivo final en estos casos es que el usuario pueda contemplar partes específicas del modelo, de una forma que garantice que no se omite información importante para comprenderlo, al tiempo que se ignoran los elementos que no son relevantes para ello. Kagdi, Maletic y Sutton [57] utilizan un conjunto inicial de elementos del modelo seleccionados por el usuario como criterio de corte. El método determina los elementos dependientes de los seleccionados y que deberían ser mostrados junto a estos. Kollman y Gogolla [58] sugieren realizar un análisis inicial del modelo para extraer métricas que sirvan para detector el nivel de acoplamiento y dependencia entre los diferentes elementos del modelo. Esta información se utiliza para generar automáticamente los distintos submodelos que se pueden mostrar. Lallchadani y Mall [59] definen un método para analizar las distintas vistas UML de modelo de una arquitectura, tanto estáticas como de comportamiento, para generar un grafo de dependencias intermedio que relaciona los elementos de diferentes vistas entre sí. Este grafo intermedio es utilizado posteriormente para crear distintos tipos de diagramas

UML basados en una selección inicial de elementos del modelo como criterio de corte.

Además de para una mejor visualización y análisis de modelos complejos, las técnicas de *Model Slicing* son útiles para optimizar la evaluación de la validez de modelos de dominio. Un modelo de clases junto a sus correspondientes restricciones OCL puede ser validado en tiempo de diseño para detectar inconsistencias antes de comenzar su implementación. Este tipo de técnicas tienden a requerir un tiempo de computación considerable, y escalan pobremente a medida que el modelo a analizar se vuelve más complejo y con más elementos, algo que sucede incluso si los nuevos elementos añadidos no afectan a las restricciones ya definidas. La solución que proponen Shaikh, Wiil y Memon [60] es usar las restricciones OCL como criterio de corte, para generar solo el submodelo afectado por las mismas. De esta forma las evaluaciones se realizan sobre estos submodelos. Aplicar las técnicas de evaluación sobre todos estos submodelos generados lleva menos tiempo que aplicarla una única vez sobre el modelo completo.

Como se puede observar, varias de estas propuestas utilizan conjuntos iniciales de clases como criterio de corte, pero no entran a considerar si el modelo producido es funcional a la hora de usarlo como base para desarrollar una aplicación. Están destinados a mejorar la visualización por parte del usuario de una parte del modelo, o para mejorar la eficiencia en un proceso de evaluación, pero no para generar nuevos elementos de diseño para una aplicación. Aunque estos algoritmos sirven como inspiración, en ciertas circunstancias los modelos generados con ellos dejan fuera partes fundamentales del modelo para ser funcionales. Además, ninguna de estas propuestas es aplicada sobre las propias restricciones OCL. Aunque Shaikh Will y Memon [60] usan las restricciones OCL para generar submodelos, no llevan a cabo el proceso inverso, útil para nuestro caso, que es en función de un modelo de clases, identificar qué restricciones OCL son relevantes para este. No obstante, su aproximación es útil ya que su análisis de los elementos de un diagrama afectados por una restricción OCL puede ser usado para inferir si dicha restricción es válida o no para un determinado submodelo.

7 PROPUESTA

Para describir la propuesta, se denominará de ahora en adelante al modelo de dominio del servidor como GDM (Global Domain Model, Modelo de Dominio Global), ya que en tiempo de ejecución el servidor almacenará el estado global de la aplicación para todos los clientes que se conecten a ella. Se denominará al modelo del dominio del cliente como CDM (Client Domain Model, Modelo de Dominio del Cliente), ya que en tiempo de ejecución cada cliente tendrá su propio estado local.

La propuesta de esta tesis consiste en un método que permita, a partir de un GDM existente, automatizar el proceso de creación del CDM hasta donde sea posible. Superado el alcance de la automatización, el diseñador tendrá que tomar decisiones para completar el diseño. Para asistir en esta parte manual del proceso, se analizará el modelo y las restricciones para generar automáticamente documentación que permita realizar una toma de decisiones más informada. La documentación proporcionará información basada en métricas, representaciones visuales y clasificaciones de las restricciones.

El método parte de dos entradas: un GDM (Consistente en un modelo de clases y un conjunto de restricciones aplicadas al mismo) y una selección por parte del diseñador de las clases del mismo que deben estar presentes en el cliente. Una vez finalizado el método, el producto resultante será un CDM que incluirá las restricciones del GDM que son aplicables localmente en el cliente, y documentación que incluirá para cada una de las restricciones información basada en métricas, una representación visual de los elementos del modelo afectados por la misma, y una clasificación en función de su tipo y su grado de dependencia con el servidor.

El método constará de 4 fases.

— Extracción de restricciones implícitas.

— Simplificación de restricciones.

— Generación del CDM.

— Análisis de restricciones, generación de documentación y clasificación.

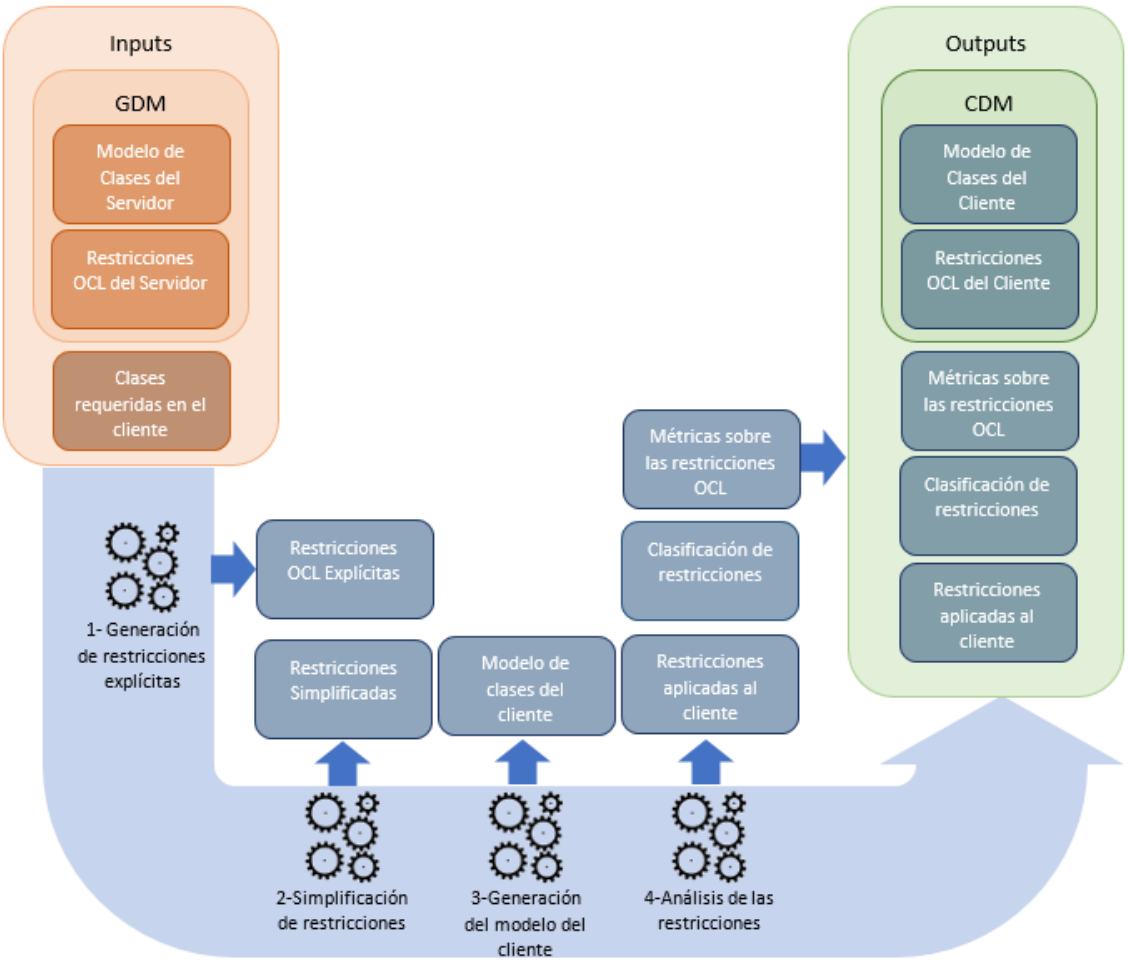


Fig 8: Esquema del proceso propuesto.

En la primera, se analizará el modelo de clases del GDM en busca de restricciones implícitas descritas en el propio modelo (como, por ejemplo, cardinalidades en las relaciones), y estas se representarán de forma explícita del mismo modo que el resto de restricciones. De esta forma todas ellas se pueden tratar de una forma uniforme y consistente.

Una vez se dispone de todas las restricciones expresadas en OCL, durante la segunda fase se aplicarán sobre ellas una serie de simplificaciones que generarán expresiones OCL semánticamente equivalentes, pero más sencillas de analizar.

Durante la tercera fase se aplicarán técnicas de *Model Slicing* para generar automáticamente un modelo de clases del CDM consistente con la selección inicial de clases por parte del diseñador.

Durante la última fase se extrae información de todas las restricciones del GDM, incluyendo a qué clases y atributos afectan, el rango de la cantidad de

instancias que pueden verse involucradas en su verificación, y cómo se navegan las relaciones entre las mismas para evaluar la restricción. Esta información es utilizada para generar documentación de las restricciones, representaciones visuales de las mismas, y es contrastada con el CDM para detectar automáticamente cuáles se pueden aplicar directamente al mismo, mientras que el resto son clasificarlas automáticamente en función de su grado de dependencia respecto al servidor.

A continuación, se detallará el papel de cada una de estas cuatro fases en el método propuesto, y por qué son necesarias en el mismo.

7.1 Fase 1: Extracción de restricciones implícitas a partir del modelo de clases

Un modelo de clases define las entidades del modelo de dominio. Cada clase viene definida por sus atributos y métodos, y sus relaciones con otras clases. UML permite establecer varios tipos de relación (asociación, agregación, composición, herencia, …), que indican cómo una instancia de una clase se puede comunicar y relacionar con instancias de otras clases. Cada uno de estos tipos de relación tiene una semántica diferente, que condiciona el modo en que se puede producir esta comunicación. Algunos tipos de relación permiten definir cardinalidades en cada extremo para indicar con cuantas instancias del extremo opuesto se puede relacionar una del extremo de referencia. Toda esta información expresada en el modelo añade restricciones al modelo de dominio, las cuales no vienen expresadas explícitamente en forma de predicados OCL, sino que están implícitas en los tipos de relación utilizados y sus cardinalidades. Por ejemplo, una relación 1 a 1 entre dos clases, impone una restricción que limita la cantidad de instancias con las que pueden comunicarse respectivamente.

Este tipo de restricciones son suficientemente simples y localizadas como para expresarlas sobre el propio modelo. Esto hace más fácil para las personas comprender esta información, y es más práctico e intuitivo que expresarlas en forma de predicados. En cambio, otras restricciones más elaboradas no pueden ser expresadas con la semántica ofrecida por los modelos de clases UML y necesitan de un lenguaje más expresivo. En esos casos, la solución propuesta en el estándar UML es la utilización de predicados OCL. Aunque disponer de restricciones expresadas en dos lenguajes distintos es útil para comprender y trabajar con el modelo, a la hora de aplicar métodos automatizados es más útil disponer de todas ellas expresadas en el mismo lenguaje.

Por eso el primer paso para el método propuesto será analizar el modelo de clases, y generar expresiones OCL equivalentes a las restricciones expresadas implícitamente sobre el diagrama. Una vez generadas es posible analizar el conjunto de expresiones OCL y procesar todas las restricciones aplicadas al

modelo sin necesidad de aplicar métodos distintos en función del lenguaje en el que el diseñador haya elegido expresarlas inicialmente.

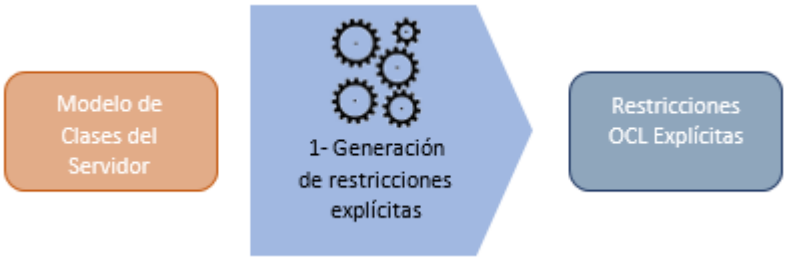


Fig 9: Entradas y salidas de la primera fase.

7.2 Fase 2: Simplificación de las restricciones

OCL es un lenguaje flexible que permite definir la misma restricción de formas muy diferentes, pero lógicamente equivalentes. Esto es posible gracias a las equivalencias lógicas existentes entre sus distintas expresiones, lo que permite al diseñador expresar una restricción del modo que personalmente le resulte más intuitivo o claro. Por ejemplo, una restricción R1 podría expresarse de varias formas, todas ellas lógicamente equivalentes:

```
context Employee inv R1:
self.name.size() <= 45 and self.name.toUpperCase() = name
```

```
context Employee inv R1:
not (self.name.size() > 45 or self.name.toUpperCase() <>
name)
```

```
context Employee inv R1:
 not Employee.allInstances()
->exists( name.size() > 45 or name.toUpperCase() <>
name)
```

Ejemplo de código 8: Ejemplo de restricciones lógicamente equivalentes.

Como consecuencia, esta flexibilidad permite expresar restricciones que pueden no ser óptimas en términos de eficiencia a la hora de ser evaluadas. Más allá de eso, una restricción puede expresarse en formatos que la complican innecesariamente, dificultando tanto su comprensión por parte de otras personas, como la aplicación de métodos automatizados de análisis sobre ella.

Es posible aplicar reglas de equivalencia sobre las expresiones para simplificarlas y acercarlas a una forma más canónica. Esto puede resultar en expresiones computacionalmente menos costosas, así como menor variedad en el tipo de expresiones encontradas, lo cual hace más fácil su análisis automático en fases posteriores.

Estas simplificaciones están recogidas en el trabajo ya realizado por Cabot y Teniente [61], que de un modo similar las aplican para facilitar el análisis posterior de las restricciones. Entre las transformaciones que se aplicarán en esta

fase se incluye la aplicación automática de reglas de equivalencia lógicas, la simplificación de expresiones con operadores de colecciones, la eliminación de expresiones *let*, o la eliminación cuando sea posible del uso de *allInstances*. Finalmente, también se aplicará una transformación a forma normal conjuntiva, en la cual determinadas restricciones consistentes en varias expresiones unidas por disyunciones se pueden transformar en varias restricciones más simples. Por ejemplo, se podría transformar la expresión R1 en dos expresiones R1a y R1b:

```
context Employee inv R1:
self.name.size() <= 45 and self.salary > 1000
```

```
context Employee inv R1a:
   self.name.size() <= 45
```

```
context Employee inv R1b:
     self.salary > 1000
```

Ejemplo de código 9: Ejemplo de la división de la restricción R1 en R1a y R1b.

Fig 10: Entradas y salidas de la segunda fase..

Estas simplificaciones no solo facilitan su posterior análisis, sino que al aumentar la granularidad de las mismas aumentan las probabilidades de que algunas sean aplicables al CDM. Una restricción para el GDM que haga referencia tanto a elementos que estarán presentes en el CDM, como elementos que no lo están, no podría ser aplicada en el CDM. Sin embargo, mediante simplificaciones esta restricción podría verse dividida en varias de alcance más limitado. Esta división puede resultar en que el alcance de alguna de las nuevas restricciones generadas se limite a elementos del CDM, siendo por tanto aplicable al mismo, algo imposible en su forma previa.

7.3 Fase 3: Automatización de la generación del modelo de dominio del cliente (CDM)

Las técnicas de *Model Slicing* son procesos iterativos que permiten dividir un modelo en varias partes. El proceso se puede dividir en tres fases principales [62]. Durante la primera fase, se selecciona el criterio de corte. Es decir, qué selección de elementos de entrada inicial van a utilizarse para comenzar a aplicar el método. La segunda fase consiste en generar una versión inicial del submodelo, con los principales elementos que son relevantes para la división. Durante la tercera fase, se detectan aquellos elementos que están fuertemente acoplados a estos elementos iniciales, y se añaden automáticamente. Este último proceso se repite de forma iterativa hasta que no se detecta la necesidad de añadir más elementos.

Fig 11: Entradas y salidas de la tercera fase.

Gracias a este proceso, el trabajo del diseñador se limita a identificar las necesidades básicas del cliente, y el proceso incluirá todos los elementos necesarios para tener un CDM consistente. Para esto el diseñador selecciona las clases clave del GDM que son necesarias en el cliente, y durante la segunda fase del proceso de *Model Slicing* se generará una versión inicial del modelo de clases. Durante la tercera fase del proceso se añadirán clases adicionales necesarias para mantener la consistencia del modelo. Como hemos visto anteriormente, existen distintos tipos de relaciones en UML, y al crear un submodelo inevitablemente algunas de esas relaciones se romperán. Sin embargo, no todas las relaciones tienen la misma semántica, y algunas implican una relación más fuerte entre clases que otras. Es por eso que, durante la tercera fase del proceso, estas relaciones son revisadas para comprobar si es necesario incluir ambas partes de la relación, incluso si una de ellas no estaba incluida en la selección inicial.

Una vez este proceso se completa y se dispone del modelo de clases del CDM completo y consistente, la siguiente fase de la propuesta consiste en analizar las restricciones del GDM y clasificar cuáles de ellas pueden ser aplicables a dicho cliente y cuáles no.

7.4 Fase 4: Generación de documentación y detección automática del nivel de dependencia de las restricciones del cliente con el servidor

Una vez se dispone del CDM, es posible comenzar a determinar qué restricciones del GDM se pueden aplicar a él. Para esto es necesario analizar cada restricción y contrastar su información con el CDM. Esto implica saber a qué clases accede la restricción, así como a qué atributos, y el número mínimo de instancias que intervienen en su evaluación.

Con el análisis de cada restricción se genera documentación que incluye una representación visual, y métricas acerca de la misma. No obstante, habrá restricciones para las que esta información no es suficiente para detectar de forma automatizada si son verificables localmente, y requieren la intervención del diseñador.

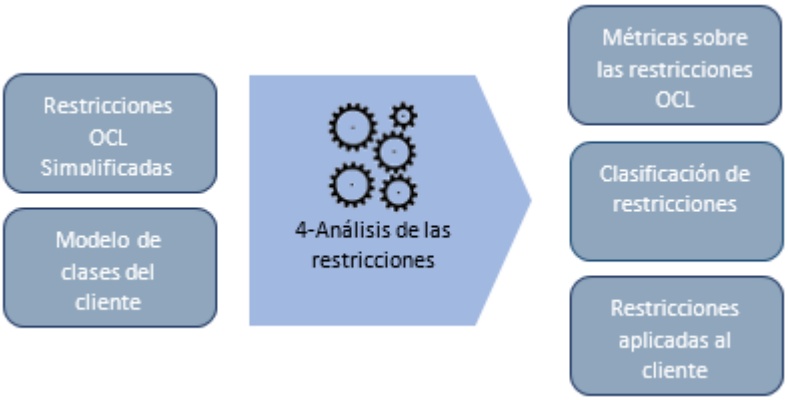


Fig 12: Entradas y salidas de la cuarta fase.

Para facilitar su tarea, la información extraída de las restricciones se utiliza para clasificarlas de forma automatizada en base a dos criterios diferentes: su nivel de dependencia respecto al servidor, y la variedad de elementos requeridos para su evaluación. Con estas clasificaciones, el diseñador puede detectar más fácilmente los casos en los que debe intervenir, al tiempo que comprende mejor por qué la restricción puede ser problemática a la hora de ser verificada localmente.

Por un lado, se clasifican en tres niveles de independencia respecto al servidor:

— Dependiente del servidor.

— Potencialmente dependiente del servidor.

— Independiente del servidor.

Por otra parte, también se clasifican en función de la variedad de elementos requeridos para su evaluación como:

— Restricción de atributo.

— Restricción de objeto.

— Restricción de clase.

— Restricción de dominio.

Las restricciones que únicamente hacen referencia a elementos que no existen en el CDM son *Dependientes del servidor*, nunca tendrá sentido su verificación en el cliente.

Las restricciones que únicamente hacen referencia a elementos que están presentes en el CDM pueden ser *Independientes del Servidor*, o bien *Potencialmente Dependientes*. Por ejemplo, si un cliente pide objetos al servidor de forma asíncrona, y una restricción requiere acceder a varios objetos del mismo tipo para ser evaluada, cabe la posibilidad de que no todos ellos estén presentes en el cliente en ese instante, por lo que sería una restricción potencialmente dependiente. Si bien las operaciones de la restricción pueden realizarse en el cliente, depende de la información de la que disponga en ese instante, y en caso de no disponer de toda ella se necesitaría pedir dicha información al servidor.

Si una restricción puede verificarse en cualquier momento sin necesidad de consultar con el servidor, se trata de una restricción independiente. Por ejemplo, una restricción que solo requiere acceder al valor de los atributos de un único objeto para ser verificada siempre será independiente, ya que en el momento que dicho objeto se encuentre en el cliente, se podrá evaluar el valor de sus atributos localmente. Todas las restricciones clasificadas como independientes se añadirán automáticamente al CDM.

Estas clasificaciones se realizan en base a las representaciones visuales y las métricas generadas tras el análisis. Además de eso, esta documentación complementa a la clasificación y proporciona información adicional útil para la toma decisiones por parte del diseñador.

7.5 Formalización

Además de desarrollar un prototipo funcional que permita realizar el proceso descrito, se desarrollará una descripción formal de la propuesta que permita describir el ámbito del problema y los métodos evitando ambigüedades, con independencia de soluciones tecnológicas específicas. Si bien esta propuesta parte de la base de los estándares UML y OCL, la formalización deberá ser suficientemente genérica como para permitir adaptarla a las características de otros estándares de modelado basados en diagramas y predicados, como podrían ser diagramas entidad-relación, o los diagramas utilizados en metodologías como *Domain-driven Design*.

Esta formalización incluirá un modelo formal para describir los modelos de clases UML, que servirá como base para elaborar el resto de la formalización.

Basándonos en éste, se desarrollará también un modelo formal para describir los elementos afectados por las restricciones OCL y sea fácilmente trasladable al lenguaje visual, al tiempo que permita definir de una forma precisa cómo se obtienen las métricas asociadas a las mismas. También se describirán formalmente los criterios de clasificación para las restricciones.

7.6 Caso de ejemplo

A continuación, se expondrá un modelo de ejemplo que servirá para ilustrar cada uno de los pasos del método propuesto a lo largo de los próximos capítulos.

Se trata de una adaptación del ejemplo propuesto por Shaikh et al [60], que

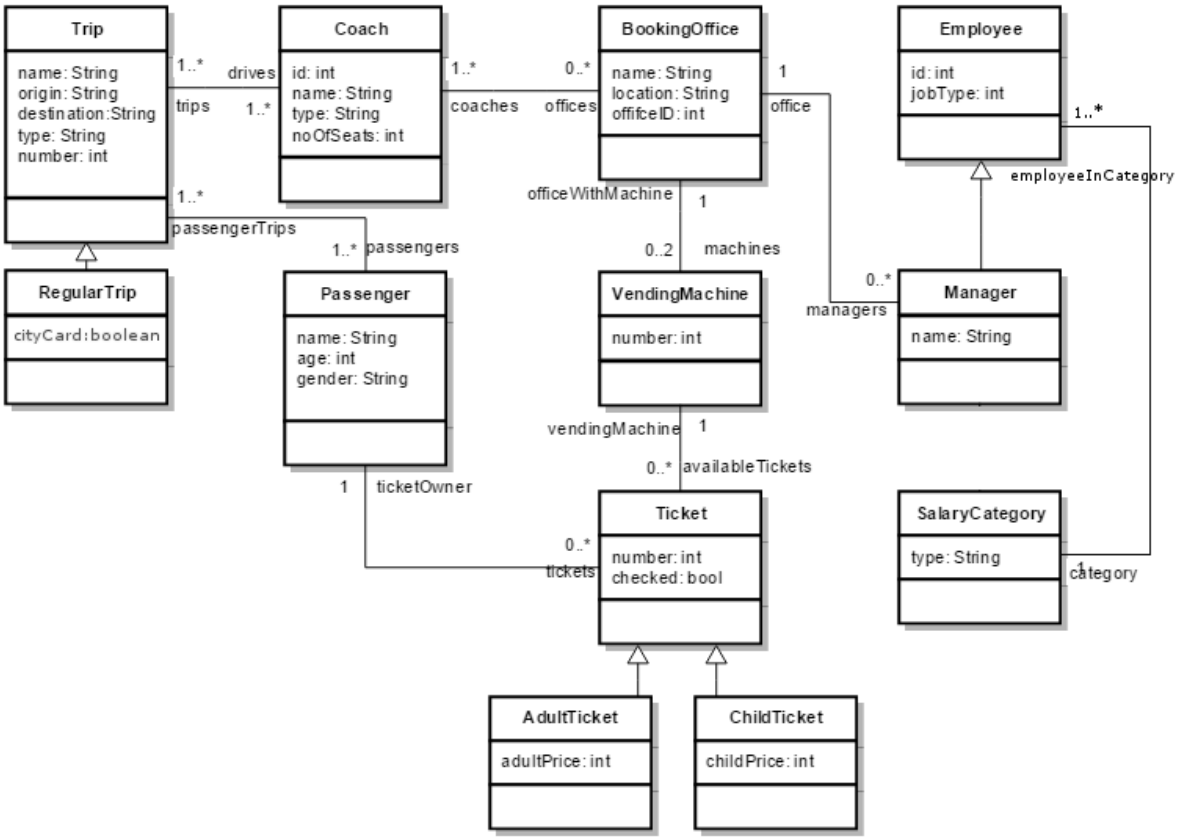


Fig 13: GDM de ejemplo que modela un sistema de gestión de billetes de autobús.

modela de forma simplificada un sistema de gestión de billetes de autobús. Utilizaremos este modelo como base para el GDM.

El modelo contempla la organización de viajes, oficinas, managers, autobuses, pasajeros y venta de billetes. Incluye además varias restricciones OCL que deberían respetarse en todo momento:

La edad de los pasajeros debe ser positiva.

```
context Passenger inv nonNegativeAge:
      self.age > 0
```

Ejemplo de código 10: Restricción 1, nonNegativeAge.

Si el tipo de autobús es grande ("*big*") el número de asientos debe ser superior a 30.

```
context Coach inv typeOfCoach:
      self.type = 'big' implies self.noOfSeats>30
```

Ejemplo de código 11: Restricción 2, typeOfCoach.

Cada billete debe tener un identificador único.

```
context Ticket inv ticketNumberPositive:
      Ticket.allInstances()->forAll(a, b | a<>b implies a.number<>b.number)
```

Ejemplo de código 12: Restricción 3, ticketNumberPositive.

Los billetes infantiles deberían estar asociados a pasajeros con edad inferior a 12.

```
context ChildTicket inv correctAgeChild:
      self.ticketOwner.age < 12
```

Ejemplo de código 13: Restricción 4, correctAgeChild.

Los pasajeros que tengan una edad inferior a 12 deben tener billetes infantiles

```
context Passenger inv correctAgeChild2:
      self.age<12 implies self.tickets->forAll(a|a.oclIsKindOf(ChildTicket))
```

Ejemplo de código 14: Restricción 5, correctAgeChild2

Un viaje no debe tener más pasajeros que la suma de todos los asientos disponibles en todos los autobuses involucrados.

```
context Trip inv enoughSeats:
      self.passengers->size() <= self.drives.noOfSeats->sum()
```

Ejemplo de código 15: Restricción 6, enoughSeats.

La suma de todos los asientos de todos los autobuses asociados a una oficina de billetes debería ser igual o mayor que el número de billetes disponibles.

```
context BookingOffice inv enoughTickets:
     self.coaches.noOfSeats->sum() >= self.machines.availableTickets->size()
```

Ejemplo de código 16: Restricción 7, enoughTickets.

Sobre este GDM, se propone la premisa de tratar de crear un CDM para una aplicación móvil que utilizarán los conductores de los autobuses para verificar los billetes de los pasajeros. Dicho cliente solo necesitará algunos de los elementos existentes en el GDM en su modelo, pero al mismo tiempo debería ser lo más independiente posible del servidor y ser capaz de verificar la mayor cantidad posible de restricciones localmente.

Es importante destacar que el objetivo de este GDM de ejemplo y sus restricciones no es proporcionar una solución completa o realista para un sistema de este tipo, sino ofrecer un ejemplo simplificado que permita ilustrar de una

forma comprensible los detalles técnicos de la propuesta durante los capítulos siguientes.

8 PREPROCESADO DE RESTRICCIONES

Los estándares UML y OCL están diseñados para facilitar la tarea del diseñador, y para ser comprendidos sin esfuerzo por los usuarios de los mismos. Sin embargo, algunas de las características que facilitan esto a las personas, pueden complicar ciertos procesos automatizados.

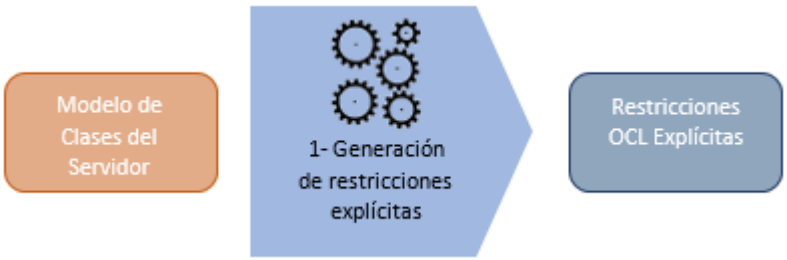


Fig 14: Proceso de generación de restricciones explícitas a partir del GDM.

Los diagramas de clases UML permiten añadir ciertos tipos de restricciones sobre el propio modelo, ya que la descripción de las relaciones entre clases y sus cardinalidades describen implícitamente restricciones sobre las mismas. Definir las cardinalidades de las relaciones mediante OCL no solo resultaría una tarea muy tediosa que requeriría escribir gran cantidad de código repetitivo, sino que además sería más difícil de comprender que su contrapartida gráfica, más ligera y clara.

Por otra parte, OCL es un lenguaje muy flexible y expresivo. La sintaxis de OCL permite expresar la misma restricción de múltiples formas, todas ellas lógicamente equivalentes. Esto da al diseñador la libertad necesaria para

expresar la restricción del modo que considera más claro para su posterior implementación.

La forma habitual de trabajar con UML y OCL permite al diseñador expresarse con libertad, pero a la hora de procesar y analizar las restricciones de una forma automatizada es necesario preparar previamente estas restricciones. Por un lado, se deben generar las restricciones explícitas equivalentes a las expresadas implícitamente en las relaciones del modelo de clases, para poder procesarlas del mismo modo que el resto de restricciones OCL. Una vez el conjunto de restricciones completo está en el lenguaje OCL, se pasarán por un proceso de simplificación y canonización que facilitará su análisis.

Fig 15: Proceso de simplificación de las restricciones.

8.1 Extracción de restricciones implícitas

En algunas de las relaciones de los diagramas de clases UML, como la asociación o la composición, es posible añadir en los extremos de la relación notación para indicar la cardinalidad de la relación.

Como se muestra en la Fig 16, la notación de UML permite al diseñador definir distintos tipos de cardinalidades:

- *: De 0 a muchos: Sin restricción respecto al número de relaciones. Puede tener cualquier número de relaciones, o ninguna.
- +: De 1 a muchos: La instancia debe tener al menos una relación.
- n..*: De n a muchos: La instancia debe tener al menos “n” relaciones.
- n..m: De n a m: La instancia debe tener entre “n” y “m” relaciones.

— n: Exactamente n: La instancia debe tener siempre exactamente "n" relaciones.

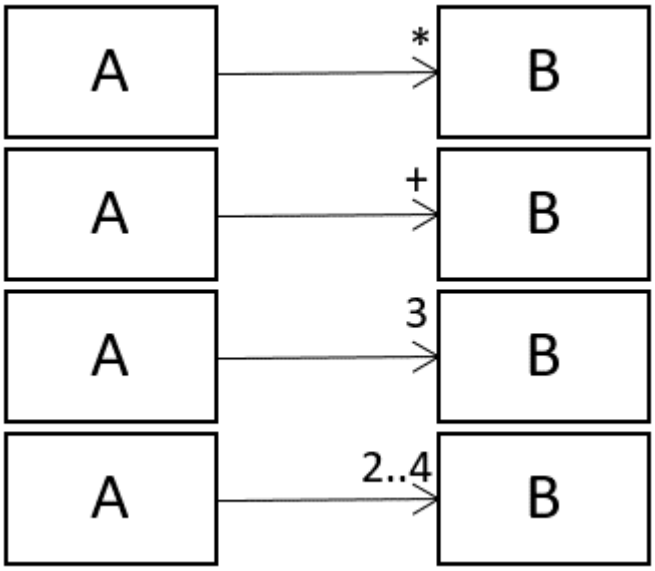


Fig 16: Distintas posibilidades a la hora de definir la cardinalidad de las relaciones sobre el modelo de clases.

Dado un modelo de clases, será necesario generar para cada cardinalidad implícita en una relación la correspondiente sentencia OCL equivalente. Para ello, el contexto será la clase origen de la relación, y el nombre de la invariante se extraerá utilizando los nombres de las clases de origen y destino, y el nombre de la relación:

```
context ClaseOrigen inv generatedImplicitReference _relName:
```

Ejemplo de código 17: Patrón de nombreado de las invariantes generadas.

La condición variará en función del tipo de cardinalidad.

*	No genera restricción
+	`self.relName->size() >= 1`
n..*	`self.relName->size() >= n`
n..m	`n <= self.relName->size() <= m`
n	`self.relName->size() = n`

Tabla 2: Cuerpo de las restricciones en función del tipo de cardinalidad.

Todas estas nuevas restricciones se añadirán a las ya existentes, siendo fácilmente distinguibles de las creadas a mano por su nombre generado de forma automática.

8.2 Simplificación de las restricciones

Una vez se dispone de todas las restricciones expresadas en el lenguaje OCL, el siguiente paso consiste en la simplificación de las restricciones, siguiendo las técnicas descritas por Cabot y Teniente [61]. Una restricción OCL puede expresarse de múltiples formas que son lógicamente equivalentes.

```
context Employee inv R1:
self.name.size() <= 45 and self.name.toUpperCase() = name
```

```
context Employee inv R1:
not (self.name.size() > 45 or self.name.toUpperCase() <>
name)
```

```
context Employee inv R1:
 not Employee.allInstances()
->exists( name.size() > 45 or name.toUpperCase() <>
name)
```

Fig 17: Ejemplos de restricciones OCL equivalentes.

Dado que OCL es un lenguaje que permite expresar predicados lógicos, es posible aplicar reglas de equivalencia lógica de predicados para simplificarlos. Esto proporciona varias ventajas.

La simplificación permite eliminar algunos de los operadores y funciones utilizados en OCL, sustituyéndolos por otras alternativas. Esto permite reducir el conjunto de elementos a tener en consideración durante el análisis automático, lo que facilitaría el desarrollo de los pasos posteriores y ayudaría a razonar acerca de ellos de un modo más simple.

Además, la simplificación permitiría en algunos casos descomponer expresiones complejas que hacen alusión a múltiples elementos del modelo en conjuntos de expresiones más sencillas y lógicamente equivalentes. Es decir, tendremos una mayor cantidad de restricciones, pero más breves y que afectan a menos elementos cada una. Este aumento de la granularidad puede aislar componentes de la restricción que limitan su verificación en el cliente, separándolos de aquellos otros que pueden ser verificados localmente. De esta forma se aumentan las probabilidades de encontrar restricciones verificables localmente de forma automatizada y sin la intervención del diseñador.

Finalmente, disponer de expresiones más simples y granulares reduce el coste computacional de la verificación en algunas circunstancias. Aunque la eficiencia en la ejecución de las restricciones no es uno de los objetivos de este trabajo, puede ser un efecto colateral ventajoso.

El proceso de simplificación se realiza en varias etapas, cada una focalizada en diferentes tipos de simplificación, tal y como se describe a continuación.

Eliminación de expresiones *let*

Las expresiones *let* en OCL permiten definir una variable local, que posteriormente se puede utilizar en la expresión. Este mecanismo es particularmente útil para evitar repetir expresiones varias veces a lo largo de la restricción.

```
context Person inv:
  let numberJobs: Integer = self.job->count() in
  if isUnemployed then
    numberJobs = 0
  else
    numberJobs > 0
```

Fig 18: Ejemplo de expresión let.

Aunque este mecanismo permite disponer de restricciones más limpias y legibles, puede complicar el proceso de análisis automático. Es por eso que el primer paso para la simplificación consiste en eliminar las expresiones *let*. Para ello, se inserta la expresión completa en todos aquellos lugares donde se está haciendo uso de la variable, al tiempo que se elimina su declaración inicial.

```
context Person inv:
  if isUnemployed then
    self.job->count() = 0
  else
    self.job->count() > 0
```

Fig 19: La expresión let mostrada en Fig 18 tras la transformación. Se elimina la declaración de la variable, y su expresión se inserta ahí donde ser referenciaba la variable.

Esta transformación puede resultar en algunos casos en restricciones más largas y repetitivas, pero simplifica el procesamiento automático al contemplar una menor variedad de expresiones.

Transformaciones de expresiones booleanas, operadores sobre colecciones e iteradores

Las restricciones OCL utilizan el álebra de de Boole en sus expresiones, y por tanto están sujetas a las reglas de equivalencia lógica.

X <> Y → not X = Y	X = true → X
X = false → not X	not false → true
not true → false	X and false → false
X and true → X	X or false → X
X or true → true	X>Y and X<=Y → false
X>Y or X<=Y → true	X>Y or X<Y → X<>Y
not X>=Y → X<Y	not X<Y → X>=Y
not X<=Y → X>Y	not X>Y → X<=Y
X=Y → (X and Y) or (not X and not Y)	not X=0 → X>0
-- when X and Y are boolean expressions	-- when X evaluates to a natural type
X->size()<=0 or X->forAll(Y) → X->forAll(Y)	

Fig 20: Reglas de equivalencia lógica.

Cuando en OCL se navega a través de una referencia con una cardinalidad mayor que uno, el resultado obtenido es una colección de objetos. El lenguaje proporciona operaciones para estas colecciones, sobre las que también pueden aplicarse reglas de equivalencia.

X->includes(o) → X->count(o)>0	X->excludes(o) → X->count(o)=0
X->includesAll(Y) → Y->forAll(y1|X->includes(y1))	X->excludesAll(Y) → Y->forAll(y1| X->excludes(y1))
X->isEmpty() → X->size()=0	X->notEmpty() → X->size()>0
not X->isEmpty() → X->notEmpty()	not X->notEmpty() → X->isEmpty()
X->excluding(o) → X->–(Collection{o})	X->including(o) → X->union(Collection{o})
X->union(Y).r_1…r_n->forAll(Z) → X.r_1…r_n->forAll(Z) and Y.r_1…r_n ->forAll(Z)	X=Y → X->includesAll(Y) and Y->includesAll(X) -- when X and Y are collections of objects
X->last()→ X->at(X->size())	X->first()→ X->at(1)

Fig 21: Simplificaciones sobre colecciones.

Además, las colecciones también admiten operaciones de iteración, que permiten recorrerlas y operar sobre sus elementos. Los iteradores también pueden verse sometidos a reglas de transformación.

X->exists(Y) → X->select(Y)->size()>0	not X->exists(Y) → X->forAll(not Y)
not X->forAll(Y) → X->exists(not Y)	X->reject(Y) → X->select(not Y)
X->select(Y)->size()=0 → X->forAll(not Y)	X->select(Y)->size()=X->size() → X->forAll(Y)
X->select(Y)->forAll(Z) → X->forAll(Y implies Z)	X->select(Y)->exists(Z) → X->exists(Y and Z)
X->one(Y) → X->select(Y)->size()=1	X->any(Y) → X->select(Y)->asSequence()->first()
$X.r_{1...}r_n.Y.attr.Z \rightarrow X.r_{1...}r_n.Y->collect(attr).Z$ -- where *attr* represents an arbitrary attribute and at least a r_i has a multiplicity > 1	X->isUnique(Y) → X->forAll(x_1,x_2 | $x_1 <> x_2$ implies $x_1.Y <> x_2.Y$)
X->forAll(Y) and X->forAll(Z) → X->forAll(Y and Z)	X->forAll(v| Y [and|or] X->forAll(v_2| Z)) → X->forAll(v,v_2| Y [and|or] Z)

Fig 22: Simplificaciones sobre iteradores.

Esta colección de reglas fueron propuestas por Cabot y Teniente [61], muchas de las cuales están a su vez definidas en la especificación de OCL [42].

A la hora de aplicar estas reglas no se producen bucles, por lo que la finalización del proceso está garantizada. Es posible que las reglas se encadenen cuando la salida de una regla corresponda al patrón de entrada de otra, pero ningún encadenamiento de reglas lleva nunca de vuelta al patrón inicial de la secuencia.

Sustitución de expresiones *allInstances* redundantes

La expresión *allInstances* devuelve una colección con todas las instancias de una determinada clase. Esto permite iterar sobre ella para realizar alguna operación sobre todos los elementos de esa clase.

Sin embargo, este mecanismo puede resultar redundante. Cuando se utiliza la variable *self* dentro de una restricción, nos referimos a una instancia de la clase del contexto de esa restricción. La semántica de las restricciones OCL indica que la restricción debe ser evaluada en todas las instancias existentes de ese contexto. Por tanto, las operaciones que incluyen la variable *self* se están aplicando a todas

las instancias de una forma implícita, y en determinados casos es posible eliminar el uso de *allInstances* de una restricción y utilizar *self* en su lugar. Así, por ejemplo, la restricción:

```
context Employee inv R1:
     Employee.allInstances()->forAll( name.size() <=  45)
```

Ejemplo de código 18: Restricción con allInstances.

es funcionalmente equivalente a:

```
context Employee inv R1:
     self.name.size() <= 45
```

Ejemplo de código 19: Restricción equivalente sin allInstances.

La principal diferencia entre ambas aproximaciones es que el uso de *allInstances* indica explícitamente que al ejecutar esa operación se deben recorrer todas las instancias de esa clase. OCL específica las restricciones que se deben respetar, pero no indica al desarrollador en qué momento se debe realizar la evaluación. Cuando se utiliza la variable *self*, el desarrollador tiene margen para poder implementar la restricción siguiendo estrategias de optimización, como por ejemplo evaluar solo las instancias tras ser modificadas. Sin embargo, una restricción que utiliza *allInstances* exige recorrer toda la colección de instancias siempre que se evalúe, limitando el control del desarrollador si decide seguir la semántica de la restricción de una forma literal. Por eso, es deseable evitar el uso de *allInstances* en aquellos casos en los que la misma restricción se puede expresar utilizando *self*.

El objetivo es, pues, eliminar durante el proceso de simplificación las ocurrencias de las expresiones *allInstances* cuando éstas sean redundantes. Esto sucede cuando el contexto de la restricción coincide con el tipo sobre el que se está aplicando la operación *allInstances.* No se aplica en el caso de que la restricción ya incluya alguna referencia explícita a la variable *self*. Las reglas son:

$$type.allInstances()->forAll(v \mid Y) \leftarrow\rightarrow Y'$$

Descripción Formal 1: Regla 1 para la eliminación de allInstances.

Donde Y' se obtiene reemplazando todas las ocurrencias de v en Y con *self*.

$$type.allInstances()->forAll(v_1, v_2, .. v_n \mid Y) \leftarrow\rightarrow type.allInstances()->forAll(v_2, .. v_n \mid Y')$$

Descripción Formal 2: Regla 2 para la eliminación de allInstances.

Donde Y' se obtiene reemplazando todas las ocurrencias de $v1$ en Y con *self*. En este caso no se elimina completamente la expresión, pero al menos se introduce la variable *self*, lo que simplifica el procesamiento posterior.

Aplicando estas reglas a la restricción de ejemplo podemos ver el resultado final. Se partiría por tanto de la restricción:

```
context Employee inv R1:
   not Employee.allInstances()
      ->exists( name.size() > 45 or name.toUpperCase() <> name)
```

Ejemplo de código 20: Ejemplo de restricción con allInstances.

Aplicando la regla *not X->exists(Y)* ←→ *X->forAll(not Y)* se obtendría:

```
context Employee inv R1:
   Employee.allInstances()
      ->forAll( not(name.size() > 45 or name.toUpperCase() <> name))
```

Ejemplo de código 21: Aplicación de simplificaciones.

Y por aplicación de *type.allInstances()->forAll(v | Y)* ←→ *Y'* tendríamos como resultado final:

```
context Employee inv R1:
   not( self.name.size() > 45 or self.name.toUpperCase() <> name)
```

Ejemplo de código 22: Aplicación de la regla 1 para la eliminación de allInstances.

Esta restricción es semánticamente equivalente a la de partida, pero evita el uso de la expresión *allInstances* y permite al desarrollador implementarla de forma más eficiente.

Transformación en forma normal conjuntiva y división de las mismas

Tras la aplicación de las simplificaciones descritas, se aplicarán los pasos necesarios para transformar la restricción a forma normal conjuntiva. Una expresión lógica está en forma normal conjuntiva si es una secuencia de operaciones AND que combinan los valores de secuencias de operaciones OR. La forma normal conjuntiva (FNC) es de interés al ser funcionalmente equivalente a varias restricciones aplicadas sobre el mismo contexto. Por ejemplo, la siguiente restricción en FNC:

```
context Employee inv R1:
  self.name.size() <= 45 and self.salary > 1000
```

Ejemplo de código 23: Ejemplo de restricción en FNC.

Sería equivalente a las siguientes dos restricciones:

```
context Employee inv R1a: self.name.size() <= 45
context Employee inv R1b: self.salary > 1000
```

Ejemplo de código 24: División en varias restricciones.

Al descomponerlas en varias expresiones más breves y granulares, tienen mayores posibilidades de ser aplicadas en el cliente.

Aplicando reglas de transformación bien conocidas es posible pasar cualquier expresión a su FNC.

X implies Y → not X or Y
if X then Y else Z → (X implies Y) and (not X implies Z) → (not X or Y) and (X or Z) *(Solo si IF es una expresión booleana)*
X xor Y → (X or Y) and not (X and Y) → (X or Y) and (not X or not Y)
not (not X) → X
not (X or Y) → not X and not Y (ley de Morgan)
not (X and Y) → not X or not Y (ley de Morgan)
X or (Y and Z) → (X or Y) and (X or Z)

Fig 23 Reglas de transformación a FNC

Aplicándolas sobre el siguiente ejemplo:

```
context Employee inv R1:
     not( self.name.size() > 45 or self.name.toUpperCase() <> name)
```

Ejemplo de código 25: Ejemplo de paso a FNC, paso1.

Por aplicación de *not* (X *or* Y) → *not* X *and not* Y

```
context Employee inv R1:
   not( self.name.size() > 45 ) and not( self.name.toUpperCase() <> name)
```

Ejemplo de código 26: Ejemplo de paso a FNC, paso 2.

Por aplicación de *not (X > Y)* → *X <= Y*

```
context Employee inv R1:
   self.name.size() <= 45 and not( self.name.toUpperCase() <> name )
```

Ejemplo de código 27: Ejemplo de paso a FNC, paso 3.

Por aplicación de *not (X <> Y)* → *X = Y*

```
context Employee inv R1:
   self.name.size() <= 45 and self.name.toUpperCase() = name
```

Ejemplo de código 28: Ejemplo de paso a FNC, paso 4.

Resultando en la expresión en forma normal conjuntiva, equivalente a la expresión de partida.

Una vez una expresión está en FNC se procederá a dividirla, creando una restricción separada para cada una de las expresiones conectadas por los conjuntores. Este proceso es trivial, dado que todas las restricciones resultantes tendrán el mismo contexto.

```
context C inv A: X and Y →
    context C inv Aa: X
    context C inv Ab: Y
```

Fig 24 Regla para la división de restricciones en FNC

Aplicando esta división sobre el ejemplo anterior el resultado final sería:

```
context Employee inv R1a: self.name.size() <= 45
context Employee inv R1b: self.name.toUpperCase() = name
```

Ejemplo de código 29: Resultado final tras la división.

Este es el último paso dedicado a simplificar las restricciones antes de comenzar su análisis.

8.3 Caso de ejemplo: Extracción de restricciones implícitas y simplificación de las restricciones

A continuación, se utilizará el caso de ejemplo para mostrar cómo se aplicaría sobre él la extracción de restricciones implícitas, y posterior simplificación.

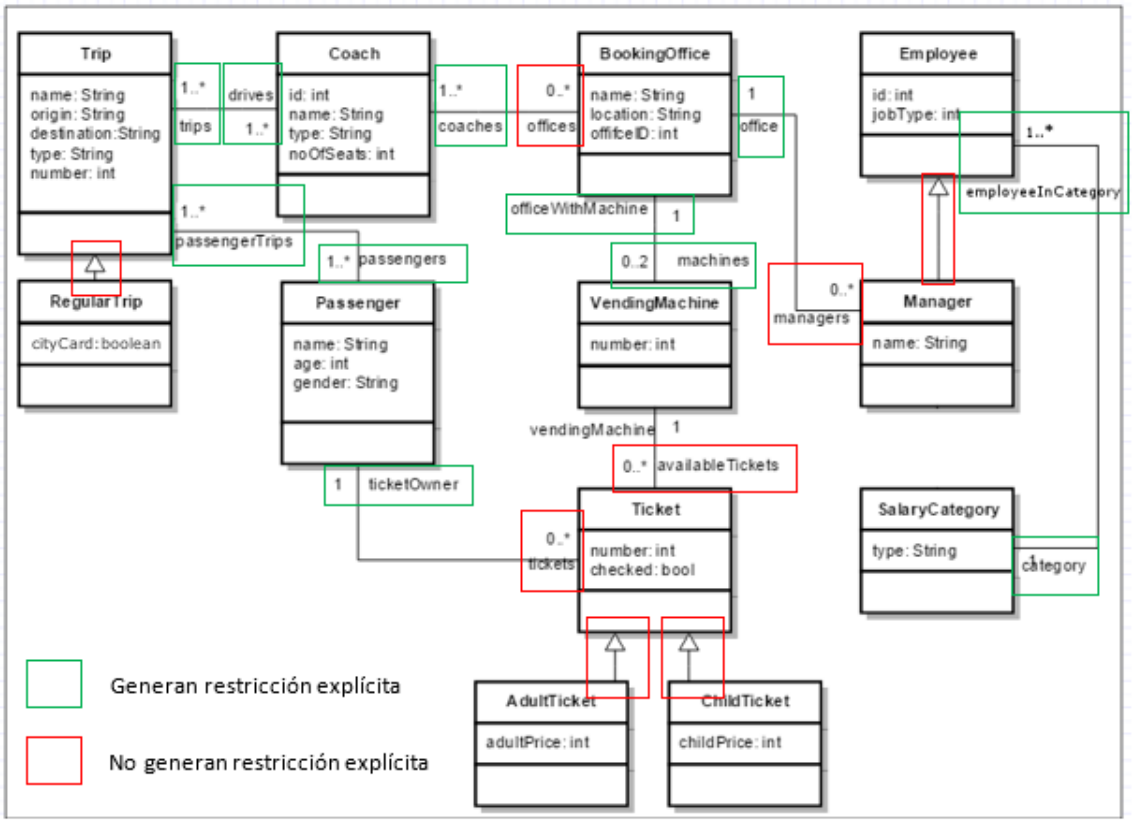


Fig 25: En verde, los extremos de relaciones de las que se extraerán restricciones implícitas. En rojo, las restricciones que no generan ninguna restricción.

De las relaciones que incluyen algún tipo de límite respecto a su cardinalidad se extraerán restricciones implícitas, mientras que el resto se verán ignoradas. En este ejemplo, las restricciones generadas serían:

```
context Trip inv generatedImplicitReference_drives:
  self.drives->size() >= 1

context Trip inv generatedImplicitReference_passengers:
  self.passengers->size() >= 1

context Coach inv generatedImplicitReference_trips:
  self.trips->size() >= 1

context Passenger inv generatedImplicitReference_passengerTrips:
  self.passengerTrips->size() >= 1

context BookingOffice inv generatedImplicitReference_machines:
  self.machines->size() <= 2

context BookingOffice inv generatedImplicitReference_coaches:
  self.coaches->size() >= 1

context VendingMachine inv generatedImplicitReference_officeWithMachine:
  self.officeWithMachine->size() = 1

context Ticket inv generatedImplicitReference_ticketOwner:
  self.ticketOwner->size() = 1

context Ticket inv generatedImplicitReference_vendingMachine:
  self.vendingMachine->size() = 1

context Employee inv generatedImplicitReference_category:
  self.category->size() = 1

context Manager inv generatedImplicitReference_office:
  self.office->size() = 1

context SalaryCategory inv
generatedImplicitReference_employeesInCategory:
  self.employeesInCategory->size() >= 1
```

Fig 26: Lista de restricciones explícitas generadas en base al modelo mostrado en la Fig 25.

Tras la generación de las restricciones explícitas, se pasará al proceso de simplificación de todas las restricciones. Aplicando las reglas descritas, el resultado final sería el mostrado en la Tabla 3.

Restricción Original	**Restricción simplificada en FNC**
context Passenger inv nonNegativeAge: self.age > 0	context Passenger inv nonNegativeAge: self.age > 0
context Coach inv typeOfCoach: self.type = 'big' **implies** self.noOfSeats>30	context Coach inv typeOfCoach: self.type **<>** 'big' **or** self.noOfSeats>30

context Ticket inv ticketNumberPositive_alternative: Ticket.**allInstances()->**forAll(a, b | a<>b **implies** a.number<>b.number)	context Ticket inv ticketNumberPositive_alternative: Ticket.allInstances()->forAll(b | **self**=b **or** (**self**.number<>b.number))
context ChildTicket inv correctAgeChild: self.ticketOwner.age < 12	context ChildTicket inv correctAgeChild: self.ticketOwner.age < 12
context Passenger inv correctAgeChild2: self.age<12 **implies** self.tickets->forAll(a|a.oclIsKindOf(ChildTicket))	context Passenger inv correctAgeChild2: self.age**>=12 or** self.tickets->forAll(a|a.oclIsKindOf(ChildTicket))
context Trip inv enoughSeats: self.passengers->size() <= self.drives.noOfSeats->sum()	context Trip inv enoughSeats: self.passengers->size() <= self.drives.noOfSeats->sum()
context BookingOffice inv enoughTickets: self.coaches.noOfSeats->sum() >= self.machines.availableTickets->size()	context BookingOffice inv enoughTickets: self.coaches.noOfSeats->sum() >= self.machines.availableTickets->size()

Tabla 3: En la columna izquierda se listan las restricciones originales. En la columna derecha se listan sus equivalentes simplificados.

9 GENERACIÓN AUTOMÁTICA DEL MODELO DE CLASES DEL CLIENTE

Tras la extracción de restricciones implícitas y la aplicación de todas las simplificaciones, eéstas pasan a ser las restricciones para el GDM (Global Domain Model, el modelo de dominio para el servidor).

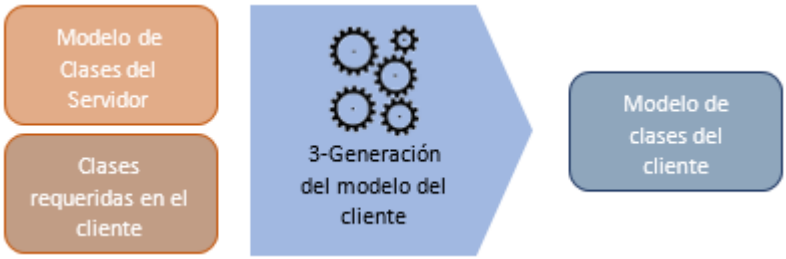


Fig 27: Proceso de generación automática del CDM.

El objetivo de esta nueva fase es simplificar la tarea del diseñador una vez ya ha creado el GDM, permitiéndole generar diferentes CDM (Client Domain Model, modelo de dominio del cliente) de forma automatizada a partir de la relación de clases provista por el diseñador que se usarán en el CDM. En este capítulo se describirá el proceso de la creación automática del modelo de clases para el CDM.

9.1 Selección inicial de clases

Si se desea crear un cliente que tenga el mayor nivel de independencia posible respecto al servidor. Éste debe ser capaz de manejar localmente la misma

información que hay disponible en el GDM. Por tanto, el CDM deberá contener al menos aquellas clases del GDM que sean relevantes para su funcionalidad.

Generalizando, el alcance de cualquier cliente será un subconjunto de las clases del GDM que, en el caso peor, incluirá todas ellas. Además de esto, es posible crear para el mismo servidor una variedad de clientes, cada uno especializado en funcionalidades diferentes, y todos ellos comunicándose con el mismo. Cada uno de estos clientes tendría como modelo de dominio diferentes subconjuntos del modelo del GDM.

La tarea de diseñar el CDM es un proceso manual que requiere de la intervención del diseñador. Éste debe observar las características del GDM, y crear un modelo de clases para el CDM que permita implementar las funcionalidades necesarias, respete los requisitos, y sea consistente con el GDM. A menudo esto implica replicar manualmente gran parte de la información ya existente en el GDM, como relaciones, cardinalidades o métodos, y por supuesto, la redefinición de las restricciones que sean aplicables localmente.

Como ya se ha explicado, nuestra propuesta requiere que el diseñador seleccione aquellas clases del GDM relevantes para el CDM. Esto implica necesariamente prescindir no solo del resto de clases, sino de las relaciones que en el GDM unían a las clases seleccionadas con las que no lo están. Sin embargo, en un modelo de clases no todas las relaciones tienen la misma semántica, y algunas tienen implicaciones que se deben tener en consideración. Por ejemplo, un diseñador puede identificar que necesitará para el cliente una clase que hereda de otra. Es posible que el diseñador ignore en ese momento la clase padre si no tiene el mismo nivel de relevancia para su cliente que la hija, pero al romper esa relación de herencia puede perderse información relevante y necesaria para el correcto funcionamiento del cliente.

Es por eso que esta propuesta no exige al diseñador identificar todos los detalles e implicaciones de las relaciones del GDM, sino identificar las clases que sabe que necesitará. Si existen relaciones de dependencia entre las clases, serán identificadas y añadidas automáticamente al modelo. Sigue siendo responsabilidad del diseñador hacer una selección que se ajuste a las funcionalidades requeridas en el cliente, pero de este modo no necesita preocuparse por respetar los detalles estructurales del modelo completo del GDM.

9.2 Criterios elegidos para el proceso de *Model Slicing*

Para generar el modelo de clases del cliente, es necesario definir el criterio que se seguirá para aplicar las técnicas de *Model Slicing*.

El primero de los criterios es el conjunto inicial de clases seleccionado por el diseñador, tal y como se describió en el apartado anterior. Todas las clases incluidas en el submodelo generado contendrán los mismos atributos y métodos

que tenían en el GDM, y las relaciones existentes entre las clases del conjunto inicial también se mantendrán.

Al tratarse de un subconjunto del GDM, las relaciones con clases externas al mismo se perderán. Esto no siempre es deseable para todos los tipos de relaciones. En algunos casos, en vez descartarse, se incluyen en el submodelo junto con la clase al otro extremo, aunque no fuese seleccionada inicialmente por el diseñador. A continuación, se describen los criterios para cada tipo de relación:

— **Relaciones de asociación, agregación y dependencia**: En este tipo de relaciones, si una de las clases en uno de los extremos de la relación está en el subconjunto y la otra no, esta relación se perderá. Esto sucederá siempre, independientemente de la navegabilidad o carnalidad definida para dicha relación.

— **Relaciones de herencia y de interfaz**: Una clase padre no necesita de sus clases hijas para tener sentido por sí misma. Sin embargo, una clase hija siempre está definida en parte por su clase padre y la necesita para estar completa. Cuando una clase hija está incluida en el subconjunto, pero su clase padre no, esta última será añadida también de forma automática. En los casos en los que la relación sea la opuesta y una clase padre esté incluida pero no su hija, esa relación no se añadirá, incluso si se trata de una clase abstracta. Una clase padre puede tener muchas clases hijas, y el diseñador puede no requerir ninguna de ellas en el CDM, o solo algunas de ellas. En esos casos no tendría sentido forzar la inclusión de clases que el diseñador no va a requerir.

— **Relaciones de composición**: La composición define una relación *parte*/*todo* entre dos clases, vinculando sus respectivos ciclos de vida. Una clase en el extremo *parte* puede instanciarse sin estar conectada a la clase *todo*. Una clase en el extremo *todo* necesita tener relación con su clase *parte* para tener instanciarse correctamente, ya que las instancias de la misma le deben acompañar durante todo su ciclo de vida. En los casos en los que semánticamente no existe ese tipo de vinculación, debería modelarse con otro tipo de relación distinto, como agregación o asociación. Es por esto que, cuando en el subconjunto hay un extremo de la relación *todo*, pero la clase del extremo *parte* no se encuentra en él, esta es incluida automáticamente. Cuando se trata del caso opuesto y es únicamente el extremo *parte* el incluido, no se incluirá el extremo *todo* de la relación si el diseñador no lo incluyó en la selección inicial.

— **Métodos**: Otro punto a considerar es que la signatura de un método contenido en una clase puede hacer referencia a otras clases. Por tanto, si dentro de una de las clases contenida en el subconjunto existe un método cuya signatura referencia clases que no están en dicho conjunto, el método se eliminará de la clase.

Todos estos criterios se deben aplicar iterativamente a medida que el algoritmo de *Model Slicing* completa el submodelo, como se mostrará en el siguiente apartado.

9.3 Algoritmo de *Model Slicing* para la generación del CDM

La selección inicial de clases realizada por el diseñador no es aún el submodelo completo. Como se ha visto, se deben evaluar las relaciones con las clases que quedan fuera del submodelo para comprobar si alguna debe añadirse al mismo. El algoritmo tiene por tanto tres fases, una vez el diseñador ha seleccionado del GDM el conjunto inicial de clases.

1. **Extracción inicial de elementos del modelo**: Se crea el subconjunto inicial. Contendrá todas las clases originalmente seleccionadas por el diseñador y mantendrá las relaciones existentes entre ellas.
2. **Inclusión iterativa de nuevas relaciones y clases**: Se recorre cada clase del subconjunto, y se comprueba qué relaciones tenía en el GDM con clases que no están en el subconjunto actual. En caso de que alguna de las relaciones cumpla los criterios definidos en el apartado anterior, estas serán incluidas en el subconjunto, junto con la clase al otro extremo de la relación. Cada vez que durante este proceso se añada alguna clase nueva, se vuelve a realizar el proceso para comprobar si las clases recién añadidas cumplen alguna de las reglas. Este proceso se repite iterativamente hasta que ninguna clase más es añadida.
3. **Eliminación de métodos**: Una vez se han terminado de añadir todas las clases y relaciones, se revisan todos los métodos de cada clase y se eliminan aquellos cuya signatura referencia clases no presentes en el subconjunto.

Una vez el proceso ha finalizado, se garantiza que el CDM generado será consistente con la información existente en el GDM. Este proceso se limita a garantizar la consistencia respecto a la selección realizada por el usuario, ya que es posible que el diseñador cometa errores a la hora de hacer una selección inicial de clases apropiada para sus propósitos.

La ventaja de este proceso es que simplifica la tarea del diseñador, limitándola a esa selección inicial. De este modo se evita la necesidad de realizar comprobaciones manuales de todos los detalles del modelo que se está creando respecto a la información ya existente en el GDM, y los posibles errores humanos que se pueden producir durante ese proceso.

9.4 Consideraciones adicionales

El proceso descrito en este capítulo crea un CDM a partir de una selección inicial de las clases del GDM. Cabe la posibilidad de que el cliente que se está diseñando

tenga funcionalidades adicionales que no tienen una relación directa con el GDM. El método descrito asegura que incluso en esos casos, la parte que sí está relacionada con el GDM será consistente con el mismo. La adición de clases y relaciones adicionales, y sus conexiones con la parte del CDM generada automáticamente quedarían bajo la responsabilidad del diseñador.

Este método también puede ser compatible con clientes que se conectan a varios GDM diferentes. El diseñador puede seleccionar las clases relevantes de los distintos GDM, y obtener las partes consistentes con cada uno de ellos por separado. La responsabilidad del diseñador en este caso consistiría en decidir cómo modelar la conexión entre los dos submodelos generados automáticamente.

9.5 Caso de ejemplo: Generación automática del modelo de clases del cliente

Se mostrará el proceso de generación automática del modelo continuando con el caso de ejemplo. En base al modelo para la gestión de billetes de autobús ubicada en el servidor, se creará un modelo para una aplicación móvil que permitirá a los conductores realizar la comprobación de los billetes de los pasajeros. El modelo para este cliente necesitará solo algunas de las clases del GDM.

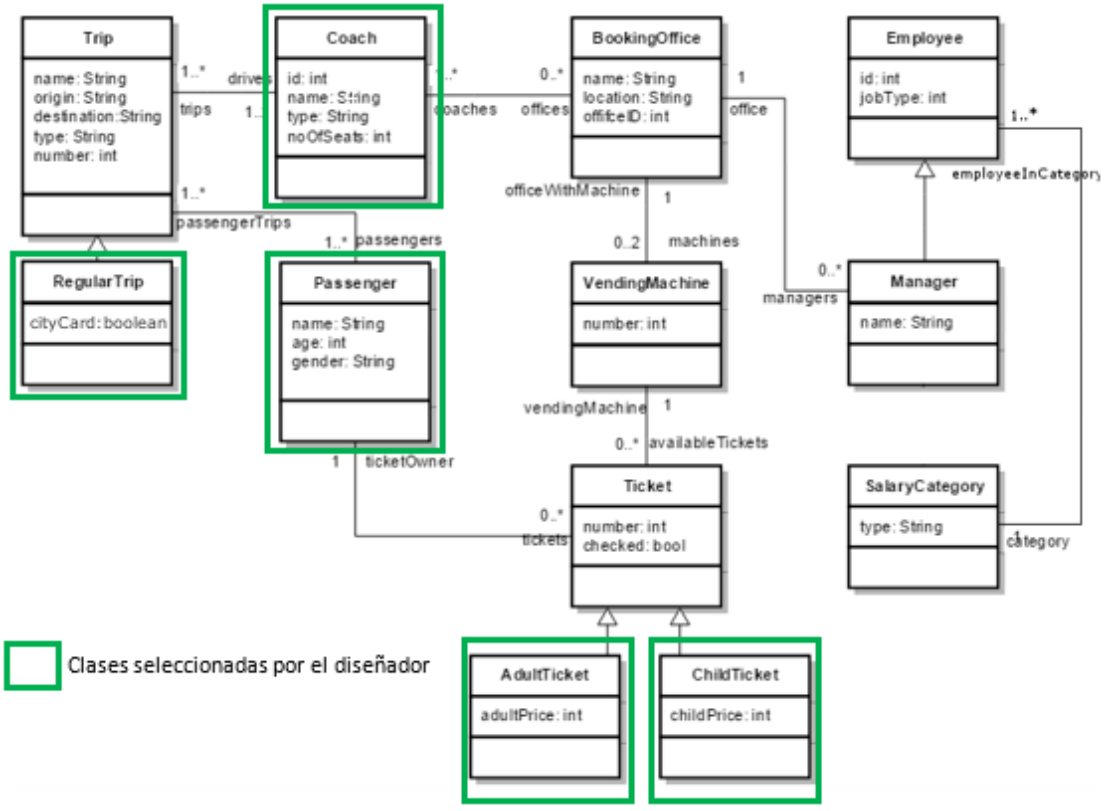


Fig 28: Selección manual sobre el GDM de clases para el cliente.

Se puede partir del supuesto de que el diseñador identificará como necesarias las clases *RegularTrip*, *Coach*, *Passenger*, *AdultTicket y ChildTicket*. Esta selección permitiría al conductor verificar en cada tipo de ticket, quién es el pasajero, qué viaje está realizando y qué autobús le corresponde. Esta selección inicial daría el siguiente submodelo de partida.

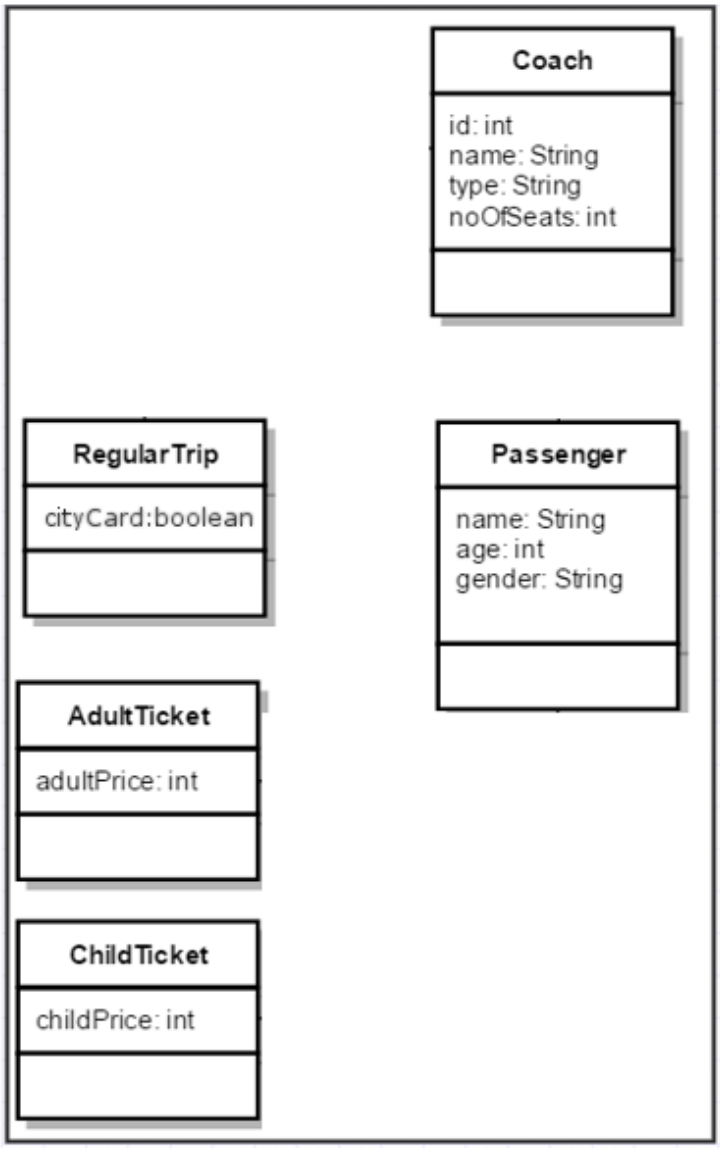


Fig 29: Subconjunto inicial para el CDM que contiene exclusivamente las clases seleccionadas manualmente por el diseñador.

Se puede apreciar que el submodelo del que se parte está incompleto ya que las clases seleccionadas, aunque necesarias para el cliente, carecen de relaciones directas entre ellas en el GDM. El siguiente paso es analizar las relaciones que se han visto cortadas respecto a las clases del GDM que no fueron seleccionadas. En este ejemplo, *RegularTrip* tiene una relación de herencia con *Trip* en la que esta última ejerce de padre, mientras que *AdultTicket* y *ChildTicket* tienen relaciones de herencia con Ticket, ejerciendo también como padre para ambas. Por tanto, ambas clases serán añadidas automáticamente, como se puede apreciar en la Fig

30. El diseñador podría haber detectado la necesidad de añadir esas clases manualmente en su selección inicial, pero el método asegura que incluso cuando se realizan omisiones el resultado final será consistente.

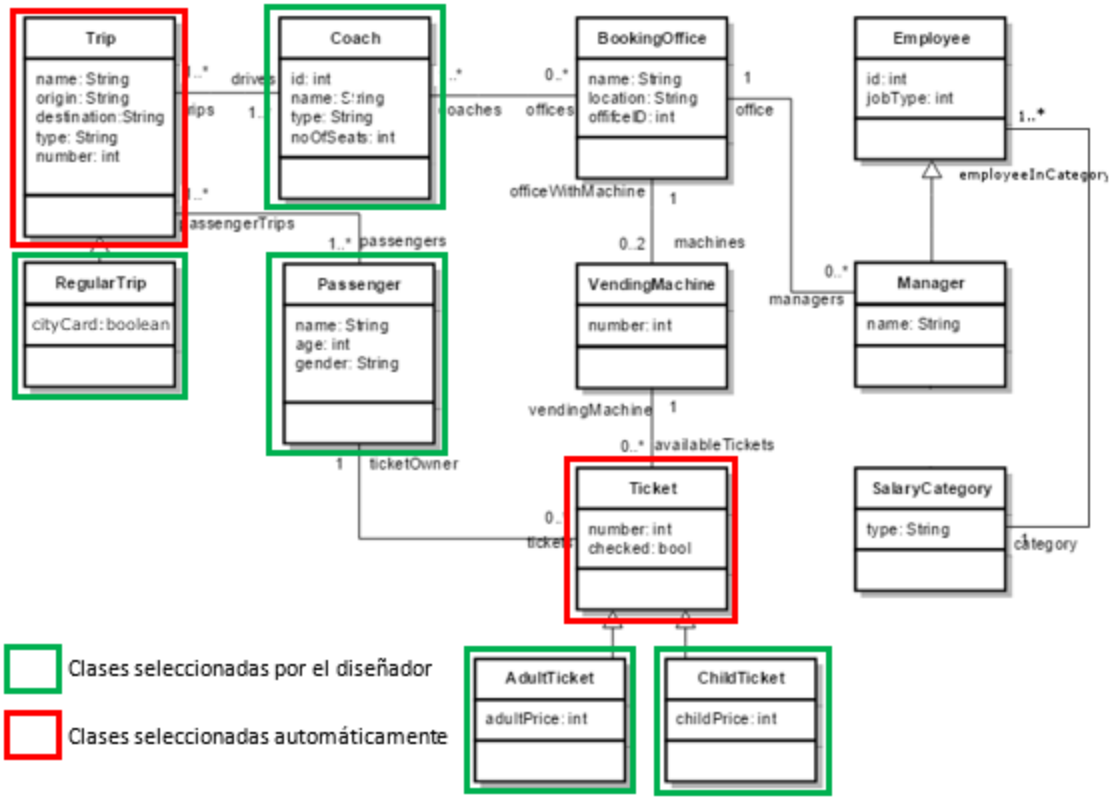


Fig 30: GDM con la selección manual del diseñador, en verde, y las clases detectadas automáticamente como necesarias, en rojo.

El resultado final, una vez finalice el proceso iterativo de incluir todas las clases y relaciones necesarias, sería el mostrado en la Fig 31.

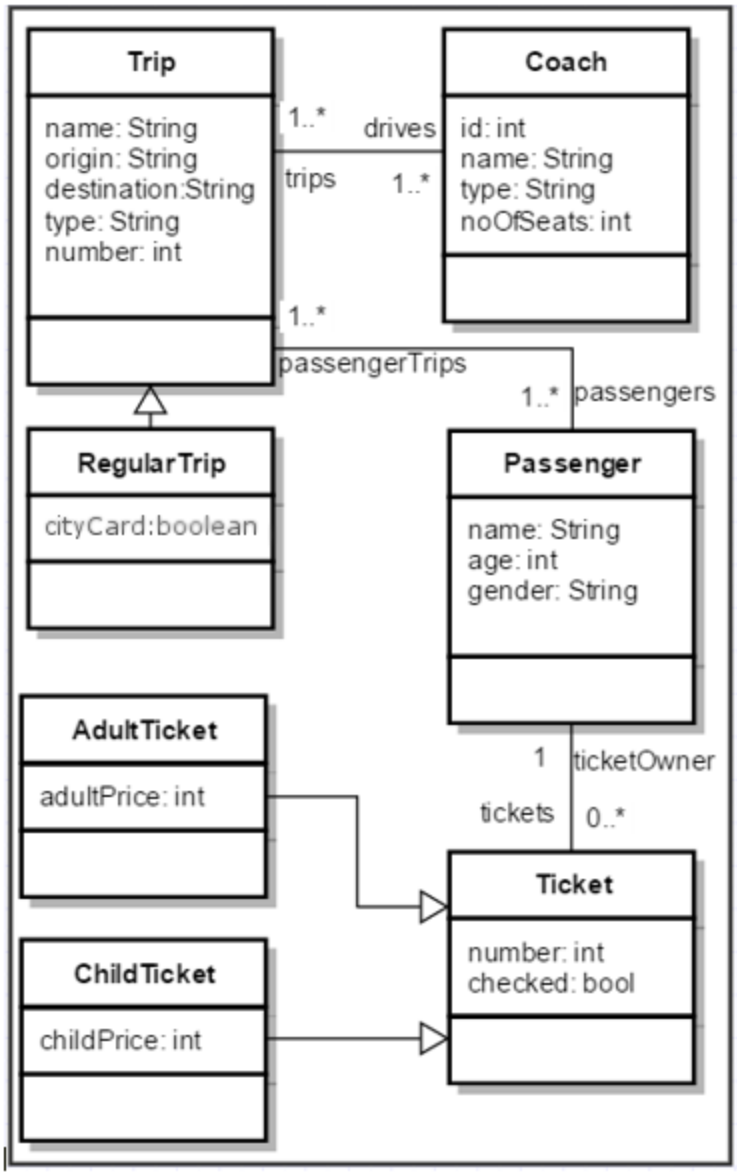


Fig 31: Modelo de clases final para el cliente, tras la inclusión de las clases y relaciones de forma automatizada.

Como se puede apreciar, ahora se dispone de un CDM consistente con el GDM a pesar de que el diseñador no incluyó algunas clases clave su selección inicial. Una vez se dispone de un modelo de clases para el cliente se puede proceder a realizar el análisis y extracción de métricas de las restricciones, como se verá en el siguiente capítulo.

10 ANÁLISIS DE LAS RESTRICCIONES Y CREACIÓN DE ÁRBOLES DE INSTANCIA

Tras la simplificación de todas las restricciones del GDM, y antes de decidir cuáles de las restricciones son aplicables al CDM que se ha generado, debemos analizarlas y extraer de ellas la información que nos permita clasificarlas. En este capítulo se mostrará cómo realizar este análisis y describirá los árboles de instancias, una abstracción que posteriormente ayudará a extraer las métricas relevantes para la clasificación de restricciones, además de servir como documentación para el diseñador.

10.1 Consideraciones relevantes para el análisis de las restricciones

Para identificar cuáles de las restricciones del GDM son aplicables al CDM se deben tener en cuenta los elementos que son necesarios para que éstas puedan ser evaluadas. Si todos los elementos necesarios para evaluar una restricción están disponibles en el CDM, entonces podrá evaluarse localmente. Si el CDM solo contiene algunos de los elementos requeridos, pero otros deben ser consultados al GDM, esa restricción no es completamente independiente para su evaluación en el cliente. No obstante, incluso cuando no hay una independencia total, puede haber distintos grados de dependencia respecto al GDM que hagan más o menos viable su inclusión en el CDM. Es necesario por tanto identificar qué elementos son necesarios para verificar cada restricción.

Las expresiones OCL siempre tienen un contexto y un cuerpo. El contexto es la clase del modelo para la cual se está definiendo la restricción, mientras que el cuerpo lo forma en la expresión booleana que debe ser satisfecha por todas las instancias de la clase definida en el contexto.

En el cuerpo de una expresión OCL, siempre se puede referenciar la variable *self*, que representa la instancia raíz sobre la que se evalúa la expresión, la cual tiene como tipo el definido en el contexto. A través de esa instancia se puede hacer referencia a sus atributos y métodos, y también navegar sus asociaciones hacia instancias de otras clases definidas en el mismo modelo. Esto se hace utilizando operadores similares a los existentes en los lenguajes de programación orientados a objetos populares, como el punto "." en el caso de acceder a atributos o navegar relaciones, o la flecha "->" cuando se trata de acceder a métodos. Aunque las expresiones OCL hacen referencia a los elementos definidos en el modelo de clases UML, son evaluadas contra el modelo de instancias (grafo de objetos) que existirá en tiempo de ejecución. El modelo de instancias estará derivado del modelo de clases, que ejerce como metamodelo del mismo.

De las expresiones OCL puede extraerse dos tipos de información. Por un lado, los elementos a los que la restricción hace referencia (instancias, clases, atributos, métodos, navegaciones…), y por otro, las operaciones lógicas que se aplican sobre dichos elementos para evaluar el predicado. En el método que se está presentando, las métricas que se utilizarán para evaluar y clasificar las restricciones obviarán qué expresiones lógicas están siendo aplicadas, y se centrarán únicamente en identificar y cuantificar qué elementos del modelo están siendo referenciados.

Comprendiendo cómo funciona una expresión OCL, se desprende que para que ésta pueda ser evaluada en tiempo de ejecución, el grafo de objetos debe contener todos los elementos necesarios para que las operaciones que se plantean puedan ser ejecutadas. Por ejemplo, para que la restricción:

```
context ChildTicket inv correctAgeChild:
     self.ticketOwned.age < 12
```

Ejemplo de código 30: Restricción correctAgeChild.

pueda ser evaluada en tiempo de ejecución, el grafo de objetos debe contener al menos una instancia de tipo *ChildTicket* desde la que se pueda navegar a una instancia del tipo *Passenger* a través de la relación *ticketOwned* para poder acceder al atributo *age*. Si el grafo de objetos no contiene instancias del tipo *ChildTicket* esta restricción no podrá ser evaluada, incluso si existen instancias del tipo *Passenger.* Tal y como está formulada, esta restricción solamente puede evaluarse navegando desde una instancia de *ChildTicket.* Del mismo modo, disponer de instancias de los tipos *ChildTicket* y *Passenger* no es garantía de que la restricción se pueda evaluar, a menos que exista una relación *ticketOwned* entre

ellas. Es necesario por tanto que existan instancias del tipo adecuado, y que la estructura de relaciones que las une sea compatible con la que se está describiendo en el cuerpo de la restricción.

La navegación entre instancias también proporciona información acerca de la cantidad de ellas que pueden verse involucradas en la evaluación. Las relaciones expresan sus cardinalidades con una cantidad mínima y máxima de instancias, por lo que, cuando ambos valores son distintos no es posible saber la cantidad exacta de instancias que van a ser accedidas durante la evaluación, pero sí el rango de instancias posible. Por ejemplo, navegar hacia una la relación con cardinalidad *1..1* devolverá una única instancia, mientras que navegar hacia una relación con cardinalidad *0..** devolverá una colección de ellas.

La variedad de grafos de objetos que una aplicación puede generar en tiempo de ejecución viene definida por su metamodelo, es decir, su modelo de clases. Dado que las restricciones se definen basándose en los elementos del modelo de clases, está asegurado que cualquier restricción definida sobre ese modelo tiene el potencial de ser evaluable en algún momento durante el tiempo de ejecución de la aplicación.

Del mismo modo, se puede realizar el razonamiento inverso. Partiendo de una restricción, podemos analizar la estructura de los grafos de objetos que es capaz de evaluar. Los elementos relevantes para este análisis serían:

— Clases referenciadas

— Número potencial de instancias

— Estructura de relaciones entre instancias

— Atributos accedidos

Con esta información, es posible comprobar para un determinado modelo de clases si es capaz de generar una estructura de objetos compatible con una restricción. Gracias a esto, es posible saber si una de las restricciones del GDM puede ser evaluable en un determinado CDM.

10.2 Análisis inicial de las restricciones: Creación del árbol AST

La información requerida para comprender qué elementos se ven afectados por una restricción OCL se puede extraer del cuerpo de su expresión. Podemos representar esta expresión mediante un árbol de sintaxis abstracta (AST, *abstract syntaxt tree*) basado en el metamodelo de OCL. Un árbol AST representa la estructura sintáctica de una expresión mediante nodos y aristas en forma de árbol, y puede ser procesado posteriormente con múltiples propósitos, siendo su uso en compiladores el ejemplo más habitual.

Existen herramientas como *Eclipse Modeling Framework* [44] o *Dresden Tools* [63] que facilitan la tarea de obtener un árbol AST a partir de una expresión OCL y son ampliamente utilizadas para propósitos similares. El árbol AST resultante incluirá información tanto de los elementos que son referenciados como de las operaciones lógicas que se ejecutan sobre ellos. En la Fig 32 se puede observar un ejemplo de árbol AST.

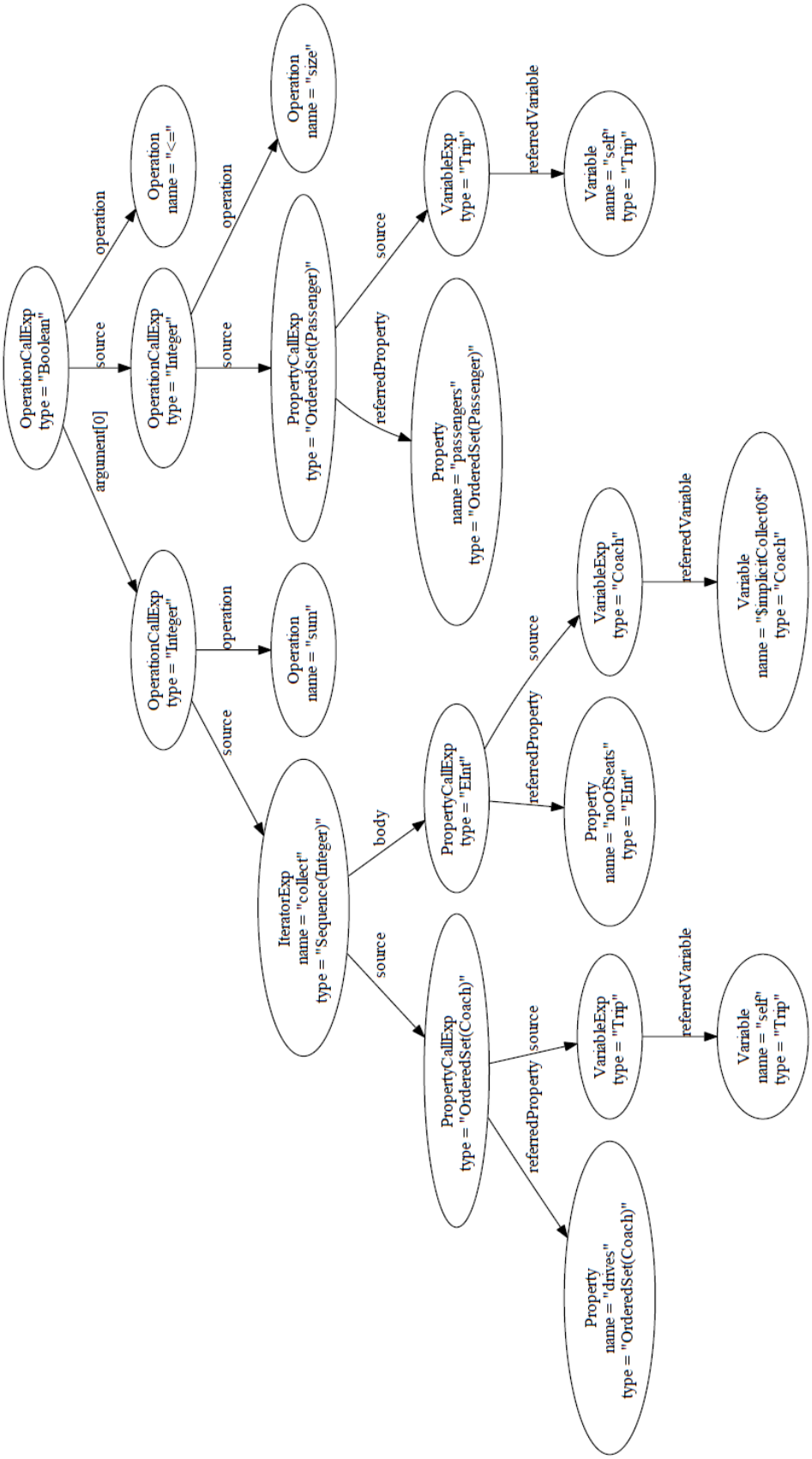


Fig 32: Árbol AST para la Restricción 6: enoughTickets.

Dado que las expresiones lógicas empleadas no son relevantes para detectar los elementos afectados por la restricción, utilizar un árbol AST completo para realizar el análisis formal complicaría la tarea innecesariamente. En cambio, se propone crear una estructura de árbol más simple que represente únicamente la información relevante para este problema. Esta estructura se generará partiendo del AST, y será denominada *árbol de instancias o IT (Instance Tree)*. Esta representación encapsulará la información relacionada con la estructura de instancias, atributos y relaciones necesarias para evaluar la expresión, ignorando las expresiones lógicas que se aplican sobre dichos elementos. En el siguiente apartado se detallarán las características de los árboles de instancia, y cómo crearlos a partir de un AST.

10.3 Identificación de las instancias involucradas en una restricción: Árboles de Instancias

Los *árboles de instancias* pretenden encapsular en una estructura sencilla toda la información relevante para el análisis de las restricciones. Por un lado, esto hace más fácil el análisis formal y la extracción de métricas relevantes acerca de la restricción. Por otro, como herramienta de documentación ayuda al diseñador a comprender de una forma visual y simple qué instancias son necesarias para evaluar una restricción y qué relaciones tienen entre ellas.

Dado que una invariante partirá siempre de una única instancia raíz *self* del mismo tipo que el contexto de la restricción, el resto de instancias serán accesibles navegando las asociaciones que parten de ésta. Es posible por tanto representar los elementos requeridos para evaluar una restricción como un árbol con nodos y aristas etiquetados, donde el nodo raíz representará a la instancia *self*. Cada nodo del árbol se denominará *nodo de instancias* y representará a un conjunto de instancias de una clase determinada, que son navegables a través de un recorrido que parte de la raíz. Las aristas que conectan los nodos de instancia tendrán etiquetada la misma información que las asociaciones definidas en el modelo de clases, incluyendo el nombre de la asociación y las cardinalidades. La Fig 33 muestra un ejemplo de *árbol de instancias* o IT. A continuación, se especificarán las etiquetas que puede incluir cada elemento del árbol.

— Nodos de instancia: Representa a una instancia o conjunto de instancias de un determinado tipo. La instancia raíz *self* siempre representa una única instancia. En los demás casos, la cantidad de instancias representadas por un nodo puede ser inferido en base a la cardinalidad definida por la asociación que navega hacia él. Dentro de cada nodo, se muestran diversas etiquetas:
    - *Variable*: Nombre de variable utilizada en la expresión OCL. Dado que a menudo es posible omitir el nombre en las expresiones

OCL, cuando se da este caso se añade automáticamente un nombre de variable (a, b, c, …).

- *Type*: Clase de la instancia.
- *Attribute*: Se incluyen únicamente los nombres y tipos de los atributos de esta instancia a los que se accede en la expresión OCL. Otros atributos existentes en el modelo de clases que no son referenciados quedan omitidos.

— **Aristas**: Se trata de las relaciones que se han utilizado en la expresión OCL para navegar de una instancia a otra. Se representan como una flecha indicando la dirección de la navegación, y junto a ella se añaden etiquetas con el nombre de la relación, y las cardinalidades mínimas y máximas entre paréntesis.

- *Nombre*: El nombre de la relación. Debería ser idéntico al descrito en el modelo de clases.
- *Cardinalidad mínima*: La cantidad mínima de instancias permitida. Este valor viene definido en el modelo de clases.
- *Cardinalidad máxima*: La cantidad máxima de instancias permitida. Este valor viene definido en el modelo de clases.

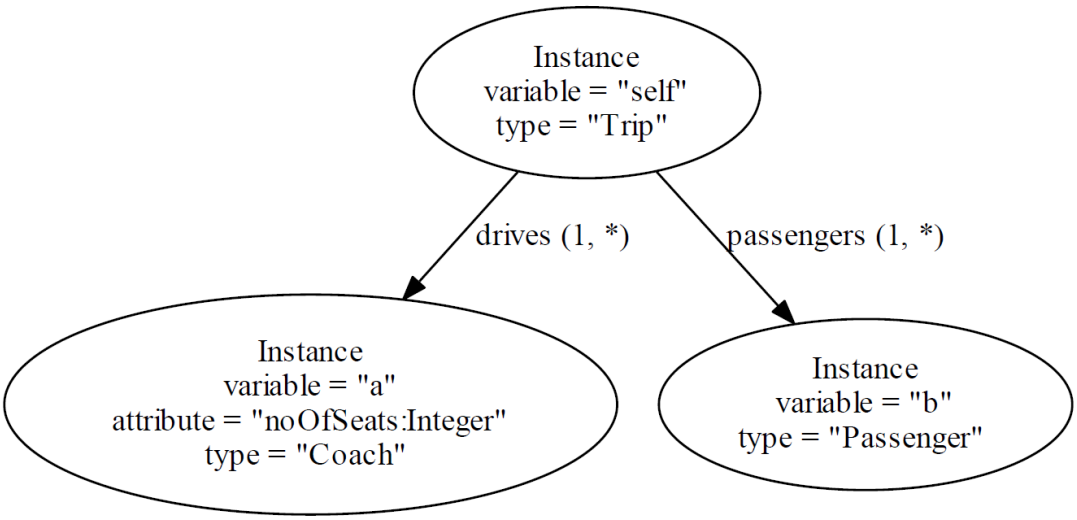


Fig 33: Ejemplo de un IT para la Restricción 6: enoughTickets AST para la misma mostrada en la Fig 32.

Es importante aclarar que los *árboles de instancias* representan las navegaciones que parten desde la raíz *self* para evaluar la restricción, y no se tratan de grafos de objetos convencionales. Cada nodo puede estar representando un conjunto de objetos del mismo tipo. OCL permite además realizar expresiones en las que la navegación genere ciclos y se vuelva a pasar de nuevo por una instancia que ya ha sido navegada. Dado que los árboles de instancia representan las navegaciones y no el grafo de objetos, no se generaría una arista que vuelva a la instancia original (rompiendo por tanto la estructura

de árbol), sino que se crearía un nuevo nodo que deje claro el recorrido de la navegación. Por ejemplo, se podría realizar una restricción con contexto *Passenger* que realice la navegación *self.passengerTrips.passengers*. En dicho caso, el árbol partiría de *self*, continuaría a un nodo *Trip*, y crearía un nuevo nodo de tipo *Passenger*.

Los árboles de instancias permiten comprender de forma visual y rápida qué elementos del modelo es necesario recorrer para evaluar la restricción. Además, pueden ser procesados fácilmente para extraer las métricas necesarias para la evaluación y clasificación de restricciones. A continuación, se mostrará el proceso de creación de estas estructuras a partir de un árbol AST.

10.4 Proceso de creación de los Árboles de Instancias

Para crear un *árbol de instancias*, es necesario partir del árbol AST de la expresión original. Éste estará compuesto por varios tipos de nodos representando las diferentes partes de la expresión, como por ejemplo operaciones, accesos a propiedades, argumentos, o declaraciones de variables.

El método presentado recorre el AST en post-orden. Para cada nodo del árbol, sus hijos son procesados primero, y ningún nodo es procesado hasta que todos sus hijos ya hayan sido recorridos.

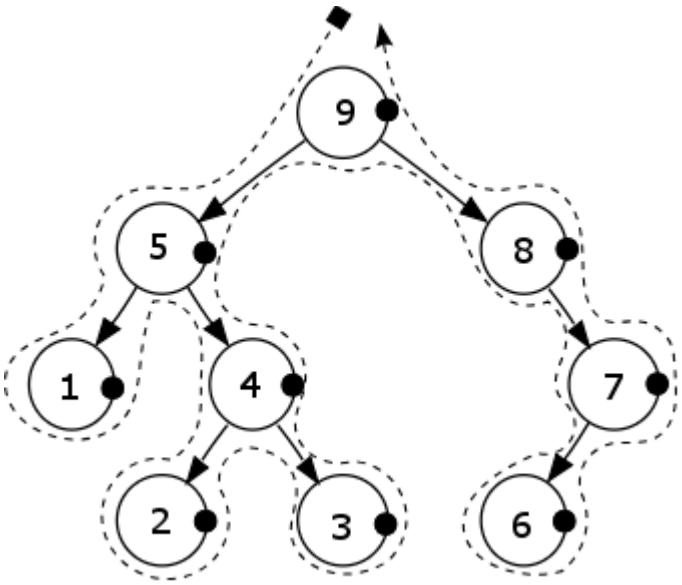


Fig 34: Recorrido Post-Orden de un árbol. Siempre se recorren todos los hijos de cada nodo primero.

A medida que los nodos AST son recorridos, se comprueba de qué tipo son. En función de dichos tipos, se generarán *árboles de instancia provisionales*. Cuando ya se han recorrido todos los hijos de un nodo AST y se procede a procesar su padre, se evalúan los nodos de instancia que han generado cada uno de sus hijos, y esas estructuras se unirán siguiendo una serie de reglas. De este modo, los hijos irán generando estructuras provisionales que se van fusionando cuando se

evalúan sus padres, hasta llegar al nodo AST raíz, donde todas las estructuras provisionales se unen para dar como resultado final un único IT. Estos *árboles de instancias provisionales* se denominarán PIT (*Provisional Instance Tree*).

Proceso de unión de *nodos de instancias*

Al recorrer el árbol AST, cada nodo puede generar diversas estructuras de *nodos de instancias*. Al ser un recorrido en post-orden, siempre se procesan los hijos primero, generando los *nodos de instancias* necesarios, y al recorrer su padre, estos son unidos. Antes de describir cómo se crean los *nodos de instancias* a partir de los diversos tipos de nodo del AST, vamos a definir el algoritmo de unión de nodos de instancia.

Dos *nodos de instancias* se pueden unir siempre y cuando sean de la misma clase. Al unir dos nodos, se creará un nuevo nodo que tendrá características compartidas de ambos. El resultado final del proceso será este nuevo nodo, mientras que los nodos originales son descartados tras finalizar la generación del nuevo nodo. El algoritmo para unir dos nodos es el siguiente:

1. Comprobar que ambas instancias son de la misma clase.
2. Se crea un nuevo nodo de instancias vacío.
3. Las variables del nuevo nodo consistirán en la unión de las variables definidas en ambos nodos.
4. Los atributos del nuevo nodo consistirán en la unión de los atributos definidos en ambos nodos.
5. Los hijos inmediatos del nuevo nodo consistirán en la unión de los hijos inmediatos de ambos nodos.
    a. Si ambos nodos a unir tienen un hijo de la misma clase y unidos por una referencia del mismo nombre, ambos nodos se unirán (siguiendo recursivamente este mismo algoritmo).

En este algoritmo solo es necesario describir qué sucede con los hijos inmediatos de los nodos que se van a unir, independientemente si estos a su vez tienen más hijos. Dado que es un proceso recursivo, todos los hijos por debajo de la primera generación pasarán por el proceso de unión de forma recursiva. La Fig 35 muestra el resultado de la unión de dos nodos sin hijos, mientras que la Fig 36 muestra el resultado de la unión de dos nodos con hijos.

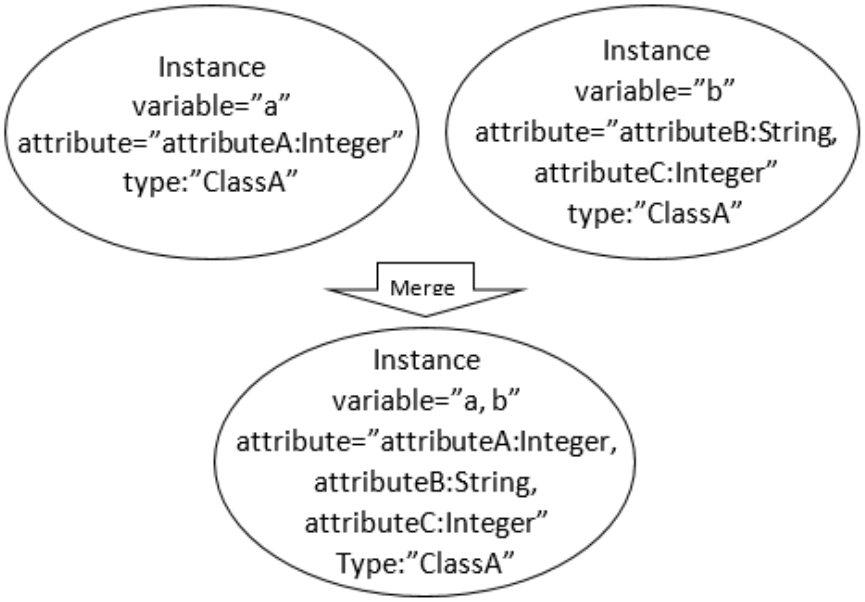


Fig 35: Ejemplo de unión de dos nodos de instancias sin hijos. Tanto las variables como los atributos quedan unidos en un único nodo.

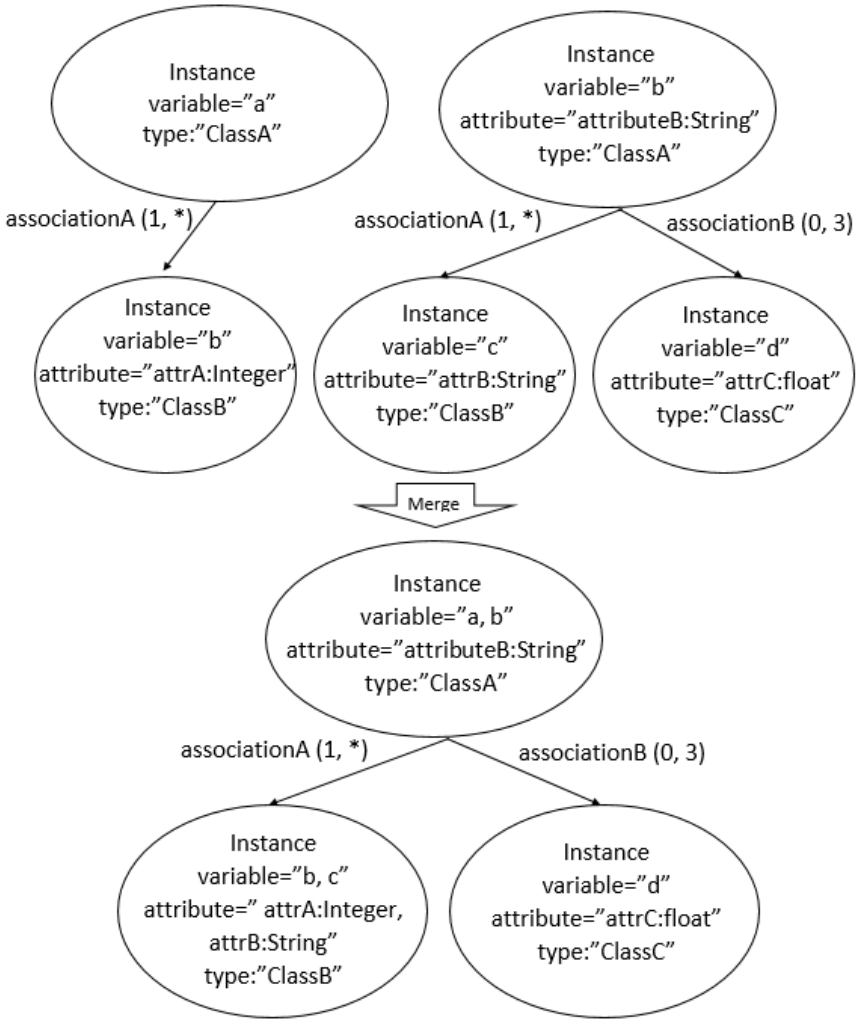


Fig 36: Proceso de unión de dos nodos de instancias con hijos.

Generación de nodos de instancias durante el recorrido del árbol AST

Antes de describir el proceso de generación de los nodos de instancias durante el recorrido del árbol AST, se realizarán algunas aclaraciones previas necesarias para comprender el proceso.

Cuando un nodo AST se procesa, el resultado es un PIT (*Provisional Instance Tree*, árbol de instancias provisional) asociado a él. Dado que el recorrido es en post-orden, cuando un nodo AST está siendo recorrido, todos sus hijos ya han sido procesados previamente, y por tanto todos ellos tienen ya sus PIT generados. Los PIT que han generado los hijos de un nodo AST son importantes, ya que pueden ser accedidos y utilizados para generar el PIT del mismo. Cada vez que un nodo AST termina de ser procesado, los PIT de sus hijos son utilizados para la creación del mismo, dando como resultado un único PIT. La Fig 37 representa este proceso de forma simplificada.

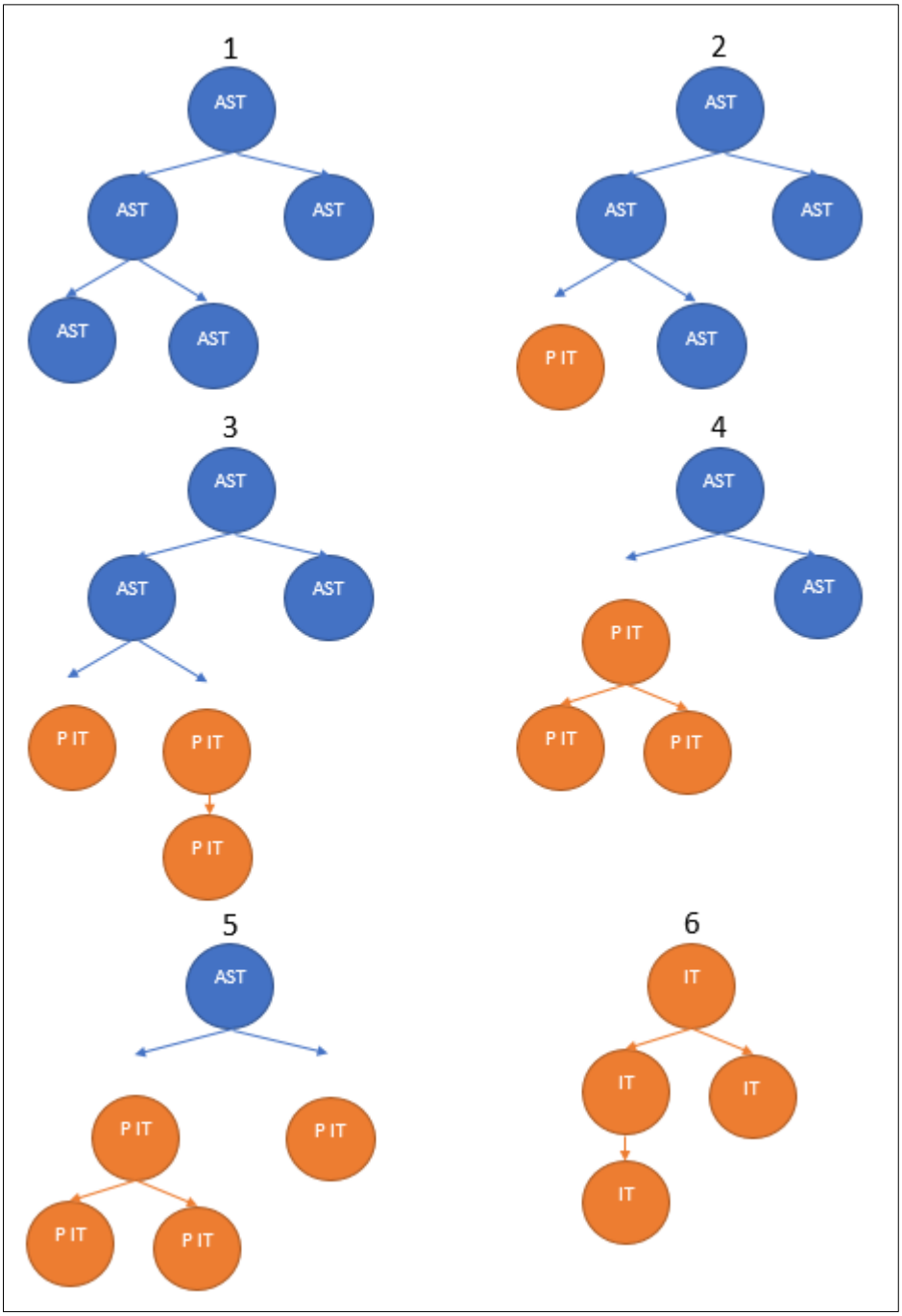


Fig 37: Ejemplo simplificado representando el proceso de creación de un árbol de instancias (IT, Instance Tree).

Durante este proceso, es habitual que un PIT tenga una estructura en la que el nodo raíz tiene un único hijo, y que cada hijo sucesivo tenga también un único hijo, resultando en un árbol con un único hijo en cada nivel, tal y como se muestra en la Fig 38. Llamaremos a este tipo de estructura *cadena de instancias o IC (Instance Chain)*, y será relevante a la hora de procesar ciertos nodos AST, ya que se necesitará acceder bien a la raíz o bien al último hijo de la cadena.

Todos los nodos de instancias tienen siempre al menos un nombre de variable. Las expresiones OCL permiten referirse a variables de forma anónima sin asignarles una variable, por lo que en esos casos al crear el nodo de instancias se le asigna un nombre de variable único (a, b, c, d, ...).

La especificación de OCL [42] define una variedad de tipos de nodos AST posibles para el árbol AST de una expresión, pero solamente los que se describen a continuación son relevantes para el proceso, otros tipos de nodos se recorrerán pero no tendrán efecto alguno.

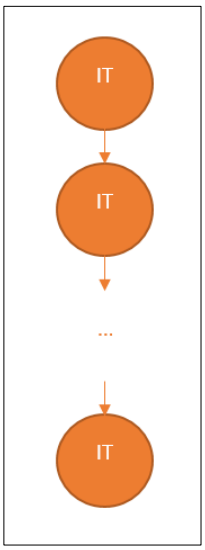


Fig 38: Cadena de Instancias o IC (instance chain).

Variable

Un nodo AST de tipo *Variable* generará un único nodo de instancias, cuyo tipo será la misma clase que la variable, y tendrá el mismo nombre que la variable.

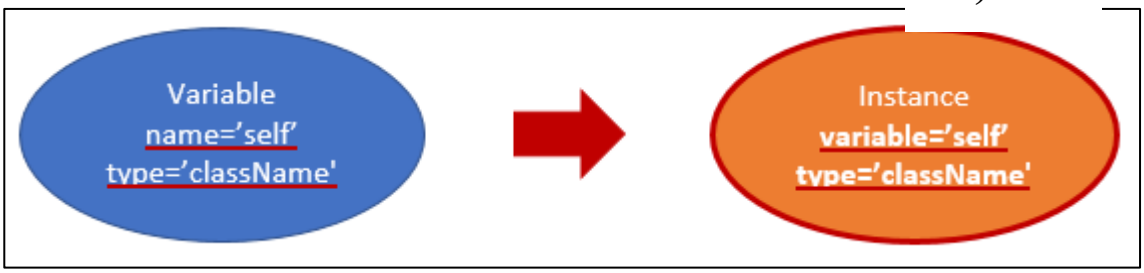


Fig 39: Ejemplo simplificado del procesamiento del nodo AST de tipo Variable.

TypeLiteralExp

Un nodo AST de tipo *TypeLiteralExp* generará un único nodo de instancias, cuyo tipo será de la misma clase que el del nodo AST. Los nodos *TypeLiteralExp* no incluyen un nombre de variable, así que el nodo de instancias añadirá uno único automáticamente.

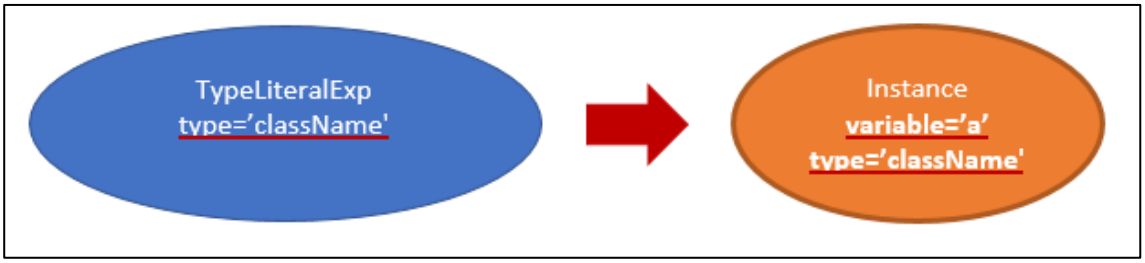


Fig 40: Ejemplo simplificado del procesamiento del nodo AST de tipo TypeLiteralExp.

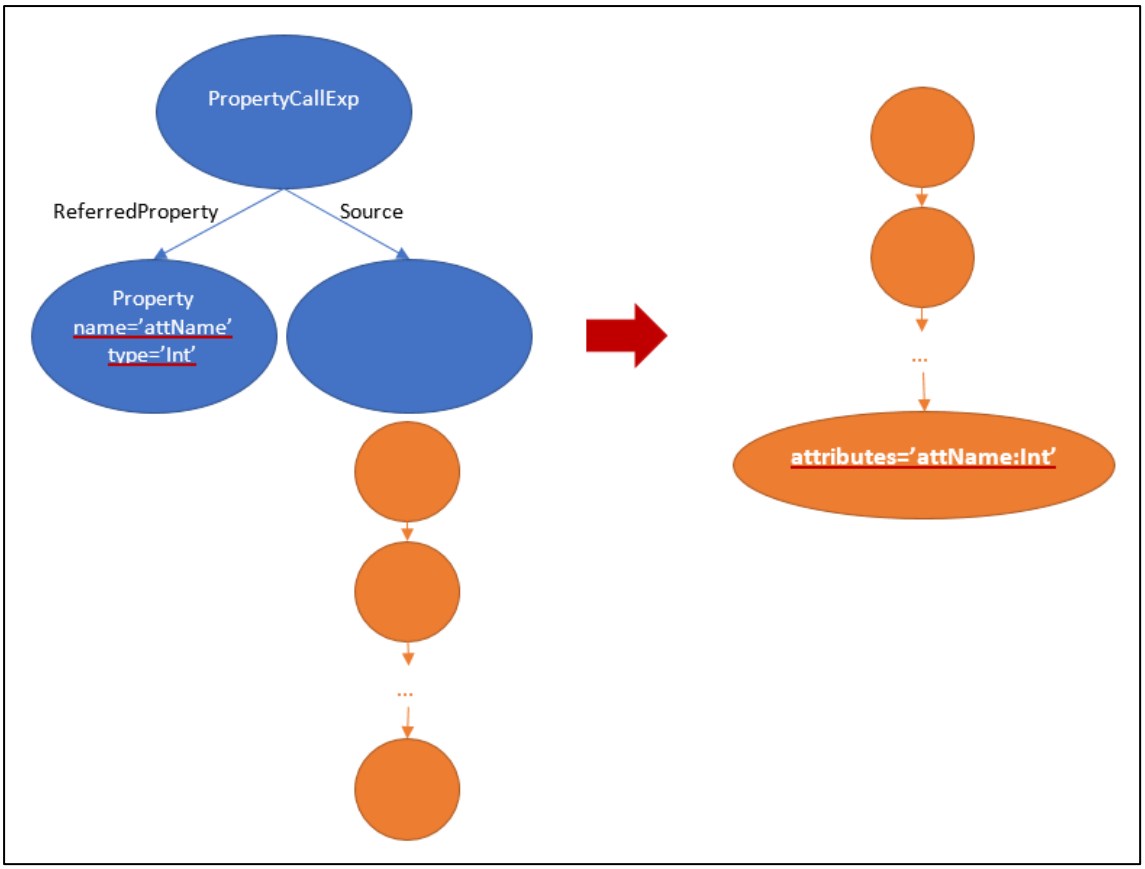


Fig 41: Ejemplo simplificado del procesamiento del nodo AST de tipo PropertyCall Exp cuando su hijo Property es de un tipo Primitivo.

PropertyCallExp

Un nodo AST de tipo *PropertyCallExp* siempre tendrá dos nodos hijos, uno referido como *source* y otro como *property*. El nodo AST que ejerce como *property* no habrá generado previamente ningún PIT tras haber sido recorrido. El nodo AST que ejerce como *source* habrá generado previamente un PIT que siempre tendrá la estructura de una cadena de instancias o IC. En función del tipo del nodo AST referido como *property* se llevan a cabo acciones diferentes para construir el PIT.

— Si el tipo del nodo AST *property* es un tipo primitivo, al último hijo de la IC del nodo *source* se le añadirá un atributo con el nombre de la propiedad

y su tipo primitivo. Esta IC modificada pasará a ser el PIT generado para el nodo AST *PropertyCallExp*.

— Si tipo del nodo AST *property* es una clase, un nuevo nodo de instancias con un nombre de variable implícito será creada y añadida como la última instancia de la IC del nodo *source*. La arista hacia este nuevo nodo tendrá el mismo nombre que la propiedad, y tendrá como cardinalidades las mismas que vienen indicadas en el modelo de clases para esa relación. Esta IC ampliada pasará a ser el PIT generado para el nodo AST *PropertyCallExp*.

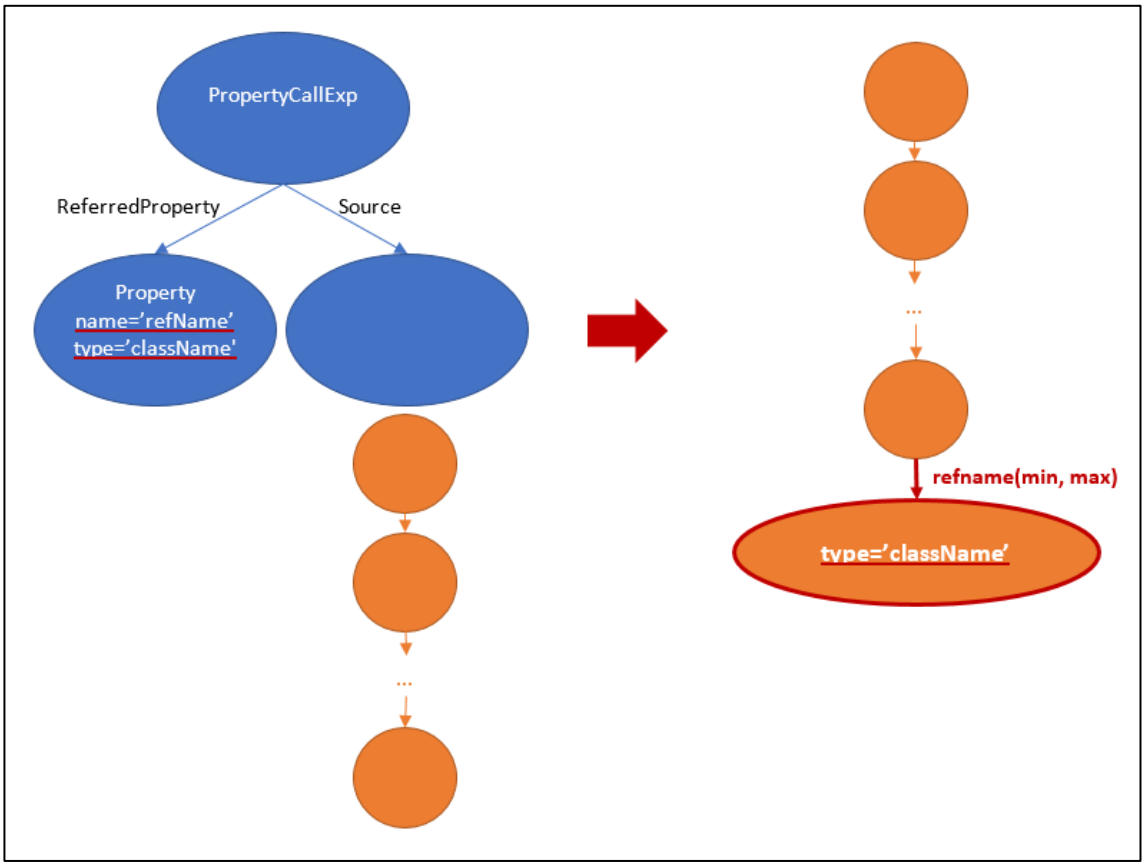


Fig 42: Ejemplo simplificado del procesamiento del nodo AST de tipo PropertyCallExp cuando su hijo Property es una clase.

IteratorExp

Un nodo AST de tipo *IteratorExp* siempre tendrá dos nodos hijos, uno referido como *source* y otro como *body*. El PITdel nodo *source* será siempre una IC. Por su parte, los nodos hijos del PIT del nodo *body* serán siempre de las mismas clases que los presentes en la IC de *source*. Al procesar el nodo *IteratorExp*, el resultado es la unión de la IC previamente obtenida del hijo *source*, con el árbol de instancias del hijo *body*, aunque la forma en la que esto sucederá dependerá del tipo de nodo AST encontrado en *body*.

— Si el nodo AST *body* es del tipo *OperationCallExp*, su PIT partirá siempre de una raíz vacía temporal, y una serie de hijos. El nodo raíz de la IC de *source*

es comparado con cada uno de los hijos del nodo raíz del PIT de *body*. Cada vez que uno los nodos de *body* es de su mismo tipo, este es unido al nodo de la IC de *source*. Este proceso es repetido para cada uno de los nodos de la IC, hasta que todos los nodos hijos del PIT de *body* hayan sido unidos. Este proceso siempre finalizará, ya que todos los nodos presentes en *body* tendrán los mismos tipos que los presentes en la IC de *source*.

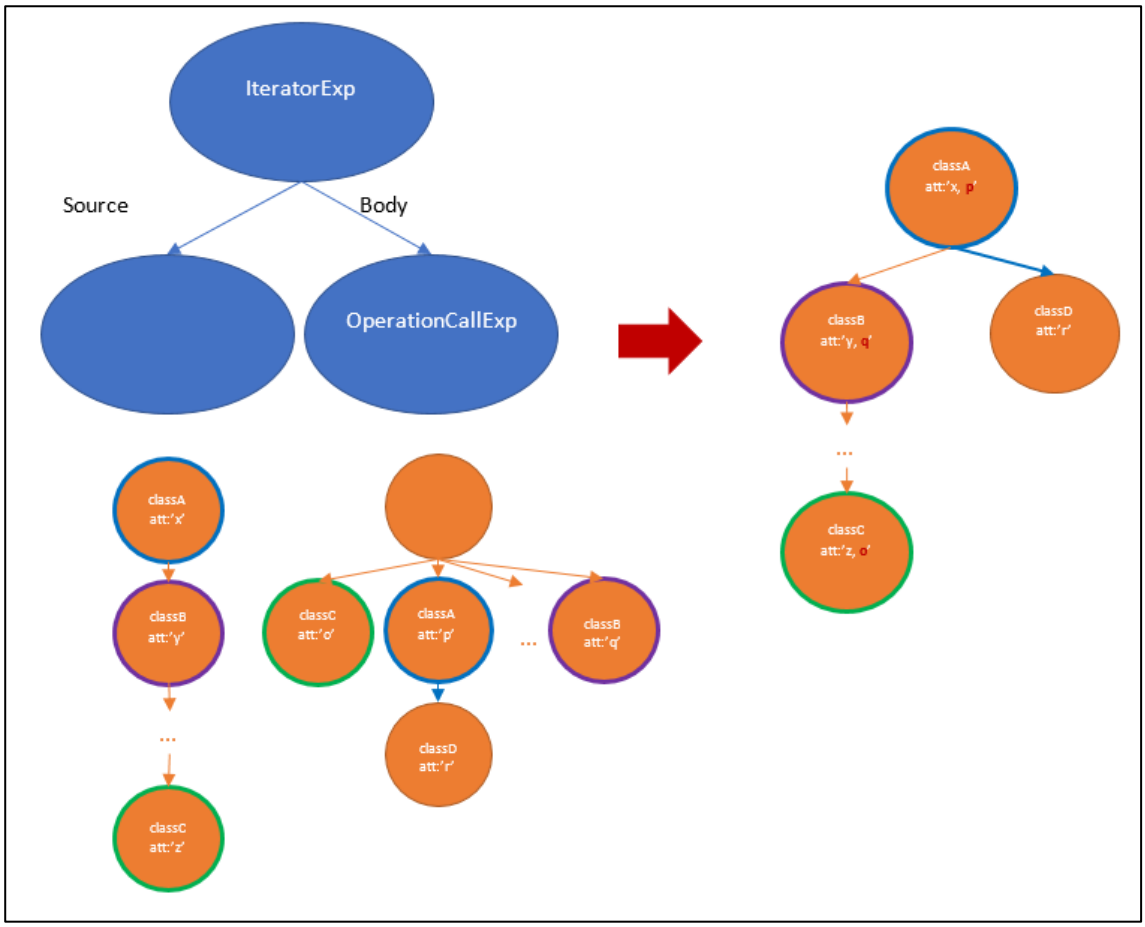


Fig 43: Ejemplo simplificado del procesamiento del nodo AST de tipo IteratorExp cuando su hijo body es de tipo OperationCallExp.

— Si el nodo AST *body* es de cualquier otro tipo, el *body* siempre generará como PIT una IC. En ese caso, la raíz de la IC de *body* será unido al último hijo de la IC de *source*.

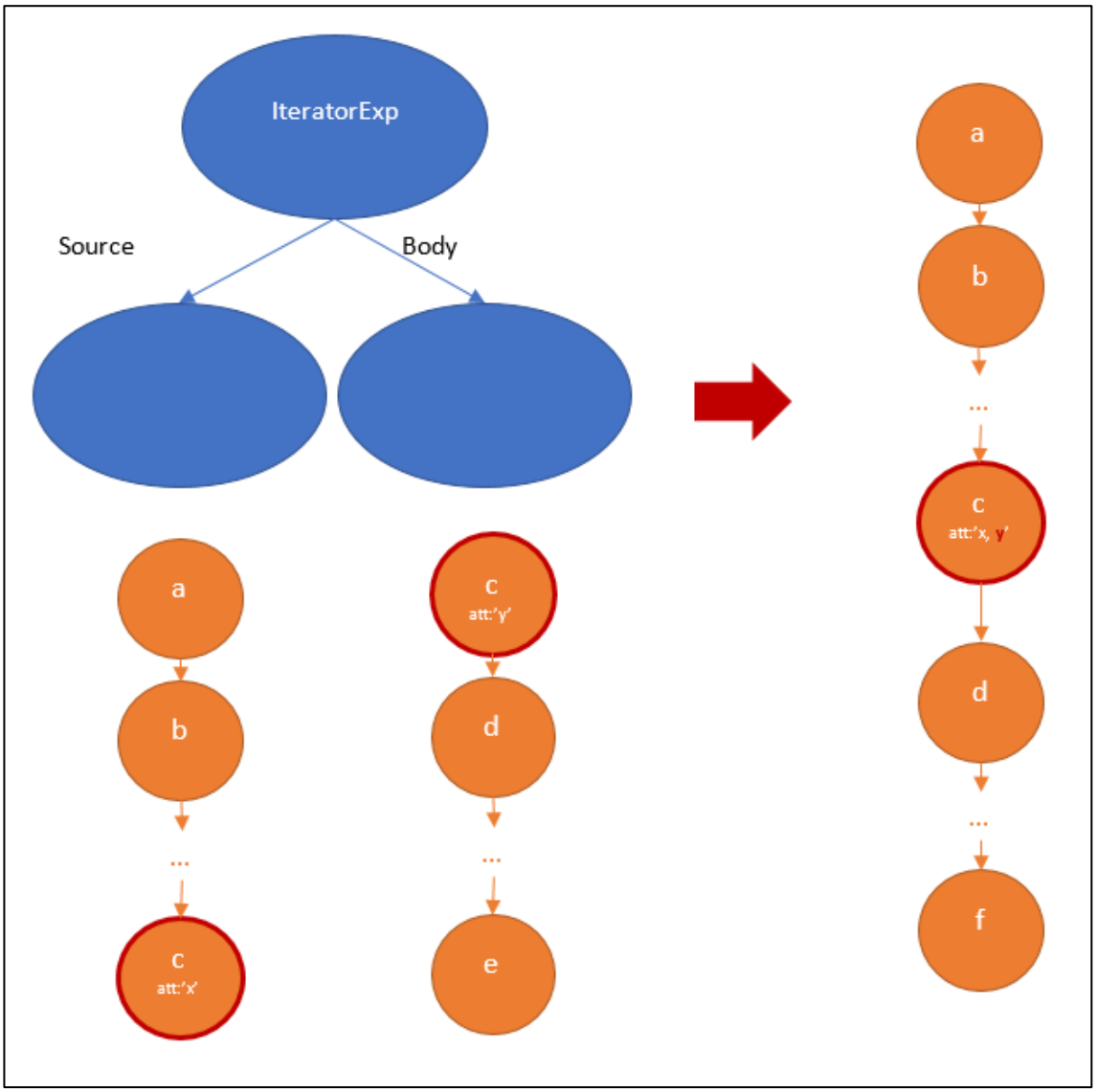


Fig 44: Ejemplo simplificado del procesamiento del nodo AST de tipo IteratorExp cuando su hijo body es de un tipo distinto a PropertyCallExp.

OperationCallExp

El nodo AST *OperationCallExp* siempre tendrá un hijo referenciado como *source*, un hijo referenciado como *operation* y, opcionalmente, un número indeterminado de nodos que ejercerán como argumentos. El hijo *source* siempre generará una IC. Los hijos que ejercen como argumentos pueden generar diferentes tipos de PIT. El hijo referenciado como *operation* nunca genera un PIT, pero es necesario comprobar si es del tipo allInstances u otro tipo de operación diferente.

— Si el nodo AST *operation* no es del tipo *allInstances*, se crea para el PIT un nodo raíz temporal vacío. Todos los PIT de todos los hijos del AST se comparan unos con otros son añadidos como hijos de esta raíz, y si son del mismo tipo se unen entre ellos. Este nodo vacío temporal se verá eliminado más adelante. Esto puede suceder de dos formas, bien cuando el PIT es procesado por un AST *Iterator*, que verá unidos sus hijos a la IC

de *source*, o bien en el momento en el que este nodo vacío tenga un único hijo, en cuyo paso pasará a sustituir a la raíz. A lo largo del recorrido del AST completo, alguno de estos dos supuestos se dará siempre, por lo que el nodo temporal siempre será eliminado.

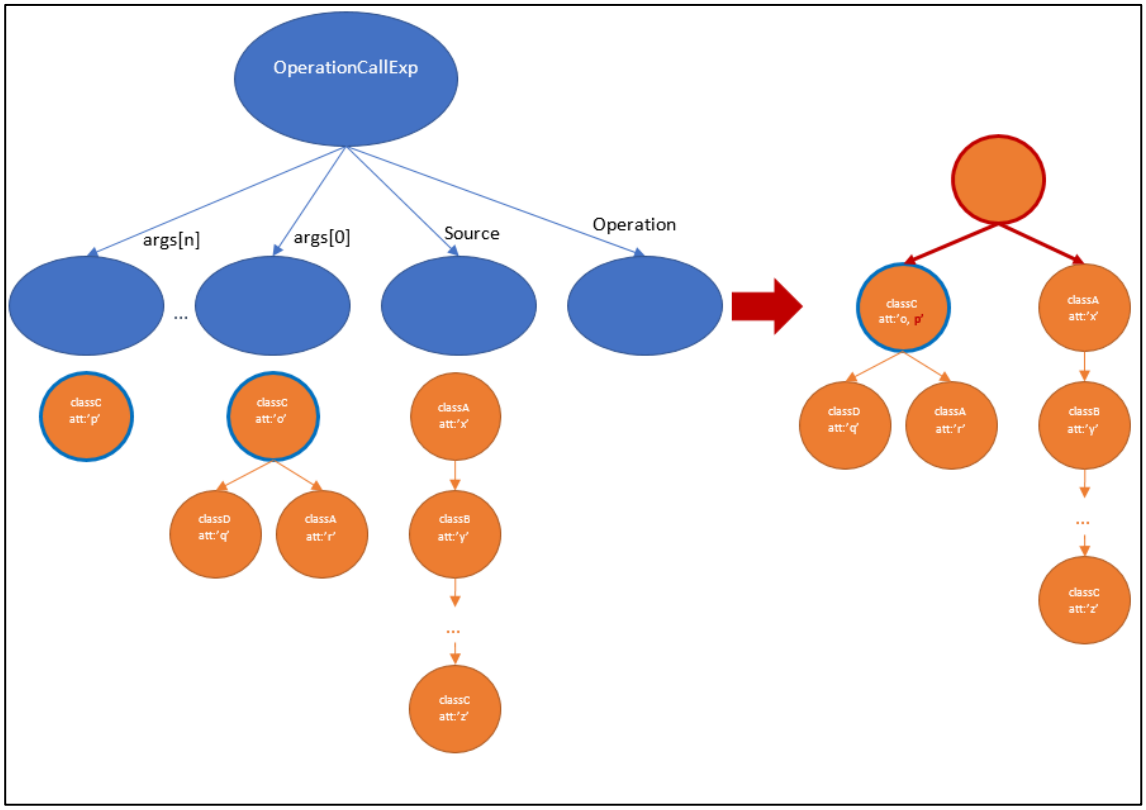


Fig 45: Ejemplo simplificado del procesamiento del nodo AST de tipo OperationCallExp cuando su hijo Operation es de un tipo distinto a allInstances.

— Si el nodo AST *operation* es del tipo *allInstances*, no existirán nodos AST de argumentos, solo *source* y *operation*. El nodo *source* habrá generado un único nodo de instancias, y el *operation* ninguno. Las operaciones *allInstances* en OCL devuelven todas las instancias de una determinada clase. Dado que se trata de una operación especial que permite ignorar el contexto inicial de la expresión, en los IT se representará como una referencia que sale desde *self*, y que representa todas las instancias de una determinada clase. Por tanto, cuando se da este caso, se crea un nodo de instancias con el nombre de variable *self* y clase igual a la del contexto, y se unirá como hijo el nodo de instancias de *source*. Ambas instancias estarán unidas por una arista etiquetada como "*className: allInstances*", con cardinalidad mínima 0 y máxima *.

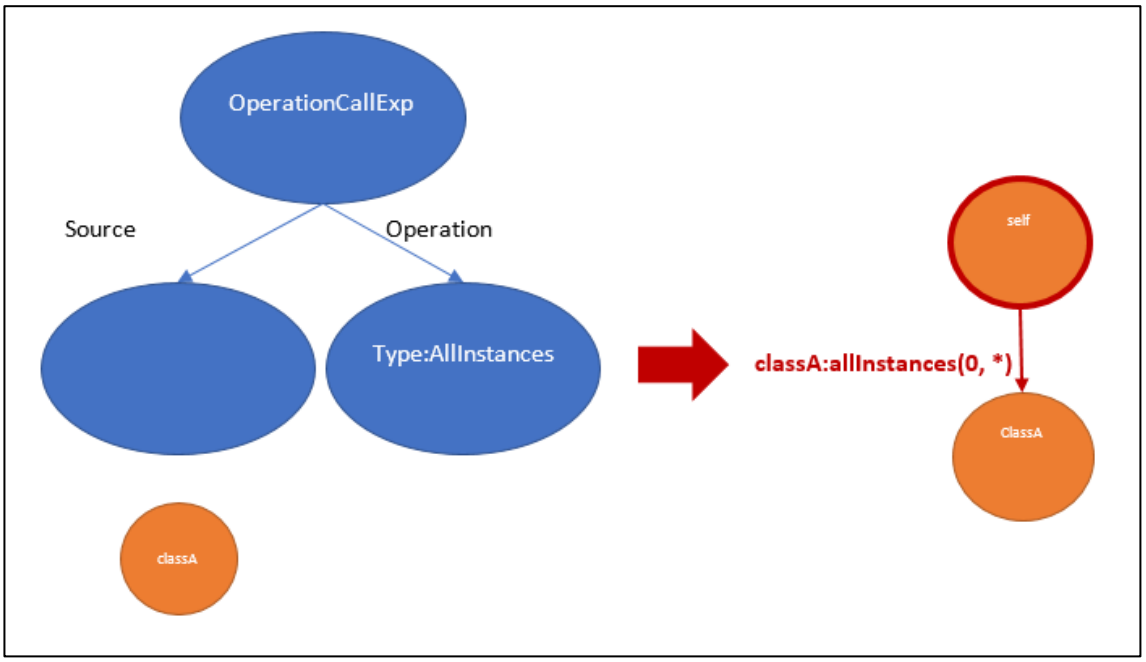


Fig 46: Ejemplo simplificado del procesamiento del nodo AST de tipo OperationCallExp cuando su hijo Operation es del tipo allInstances.

El proceso de generación del IT completo siempre finalizará, puesto que todos los nodos del AST se recorrerán en post-orden, y generarán estructuras que se irán uniendo. A pesar de que durante el proceso se generan varios árboles provisionales, el proceso de unión de nodos asegura que tras procesar el nodo raíz del AST el resultado será un grafo con una única raíz y estructura de árbol. Este tipo de grafos permitirán definir de forma clara y sencilla los criterios para la evaluación y clasificación de las restricciones.

10.5 Caso de ejemplo: Generación de los árboles de instancias

A continuación, se mostrará el proceso de generación de árboles de instancias utilizando el caso de ejemplo propuesto.

Se partirá de la siguiente restricción:

```
context Trip inv enoughSeats:
      self.passengers->size() <= self.drives.noOfSeats->sum()
```

Ejemplo de código 31: Restricción enoughSeats.

Se mostrará su AST y el proceso por el cual se genera su IT. Posteriormente, se mostrarán todos los IT generados para todas las restricciones del modelo.

El árbol AST para esta restricción puede observarse en la Fig 47. Cuenta con 19 nodos, que serán recorridos en post-orden.

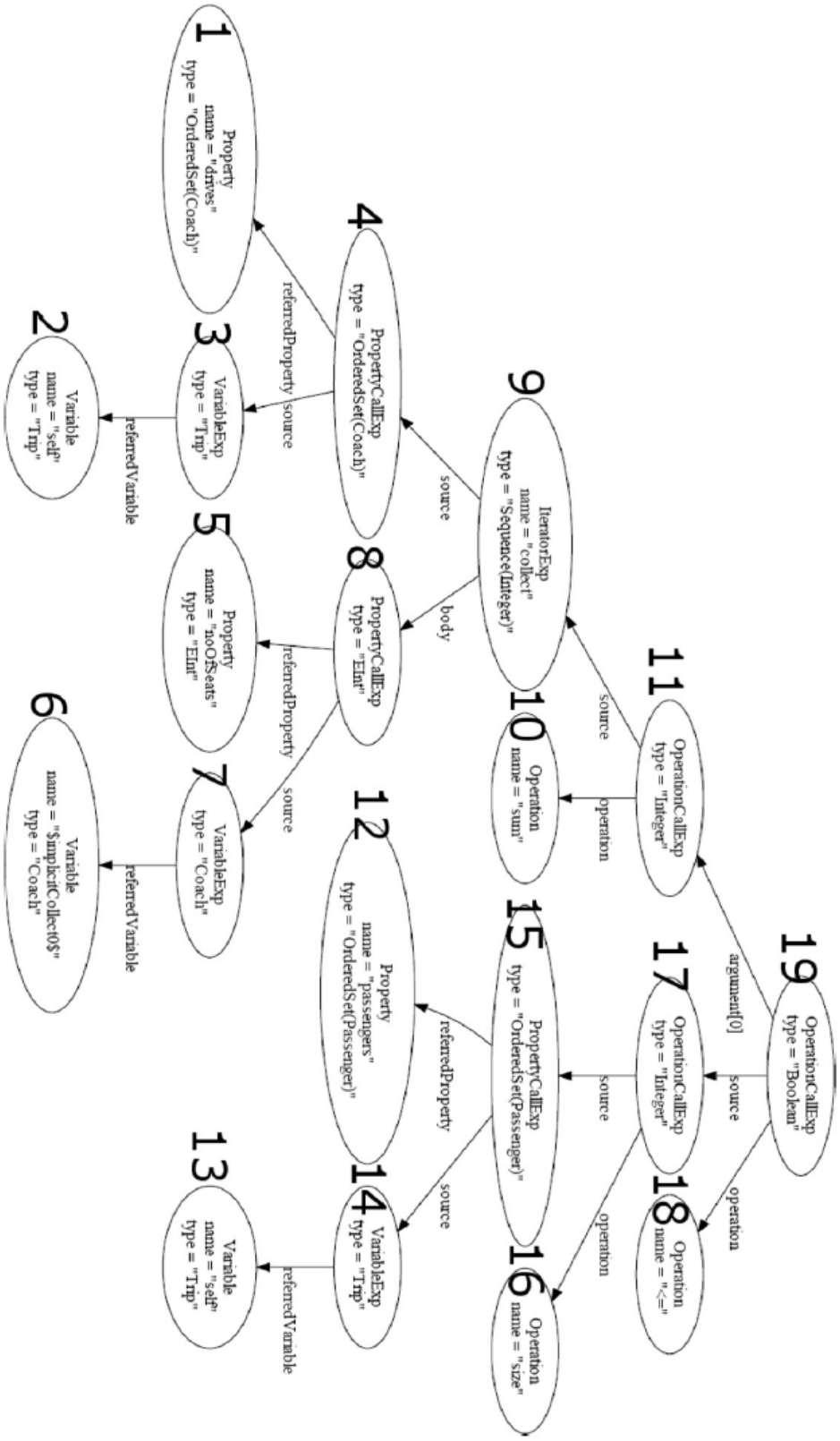


Fig 47: Árbol AST de la restricción EnoughSeats, indicando el orden en el que serán recorridos los nodos para la generación del IT.

Los nodos serán recorridos uno a uno, y aquellos de alguno de los tipos indicados en el capítulo anterior generarán árboles de instancias temporales. Entre la Fig 48 y la Fig 57 se puede observar paso a paso cómo a medida que el árbol continua recorriéndose, los PIT se van creando, modificando y uniendo, hasta recorrer el último nodo, la raíz, que resultará en el IT final.

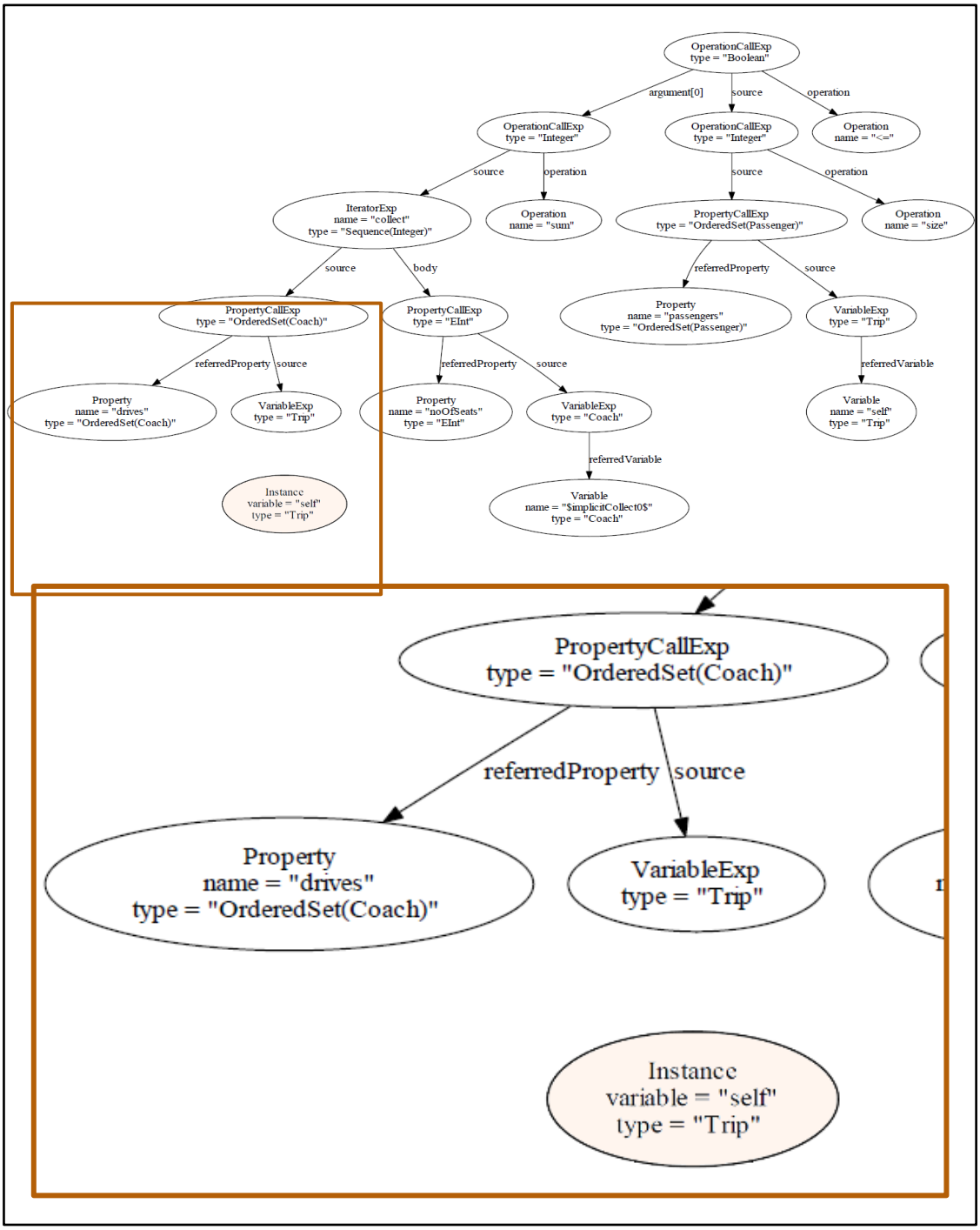


Fig 48: Proceso de creación de un IT. Paso 1, procesado de nodo Variable.

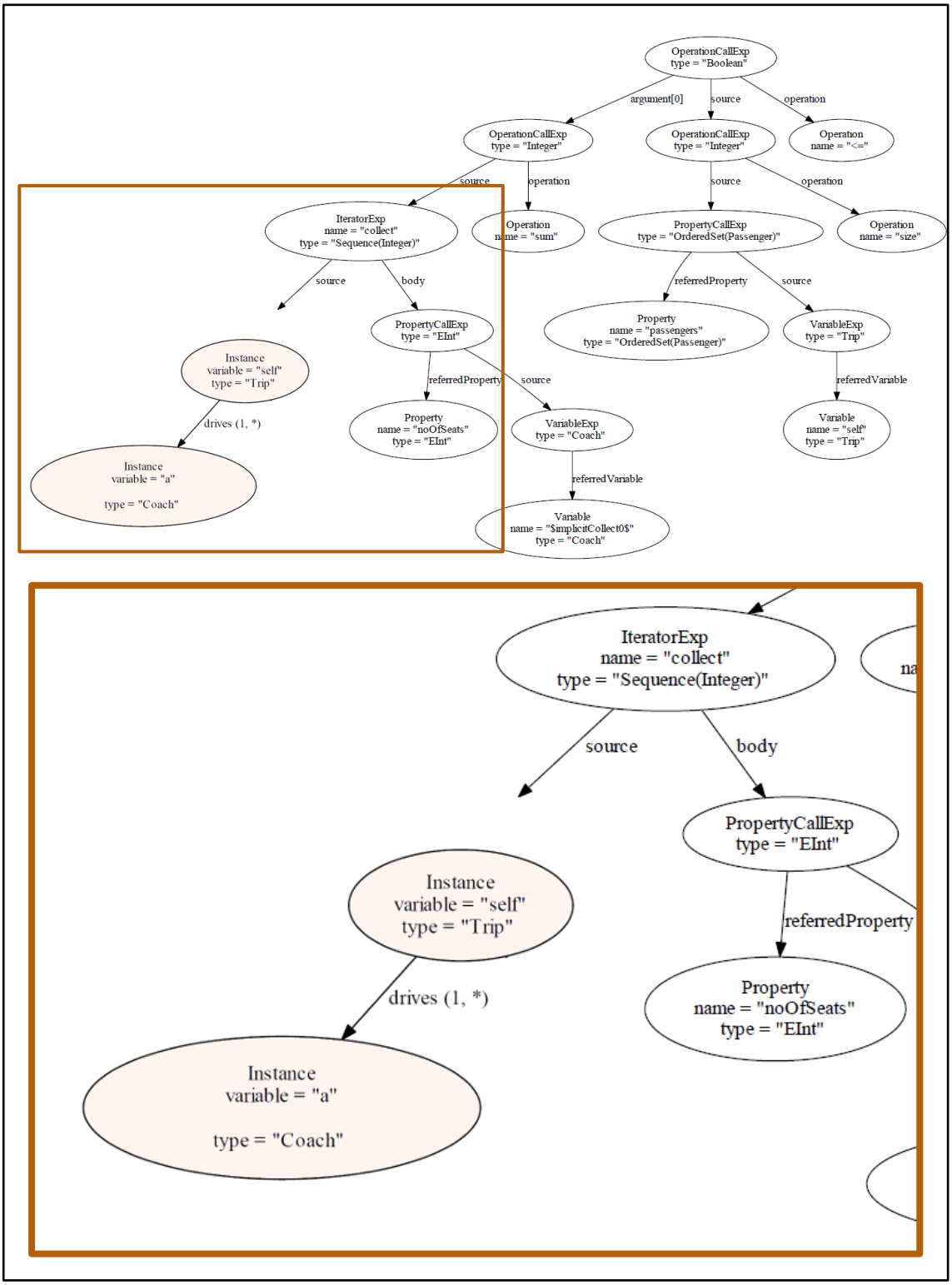


Fig 49: Proceso de creación de un IT. Paso 2, procesado de nodo PropertyCallExp.

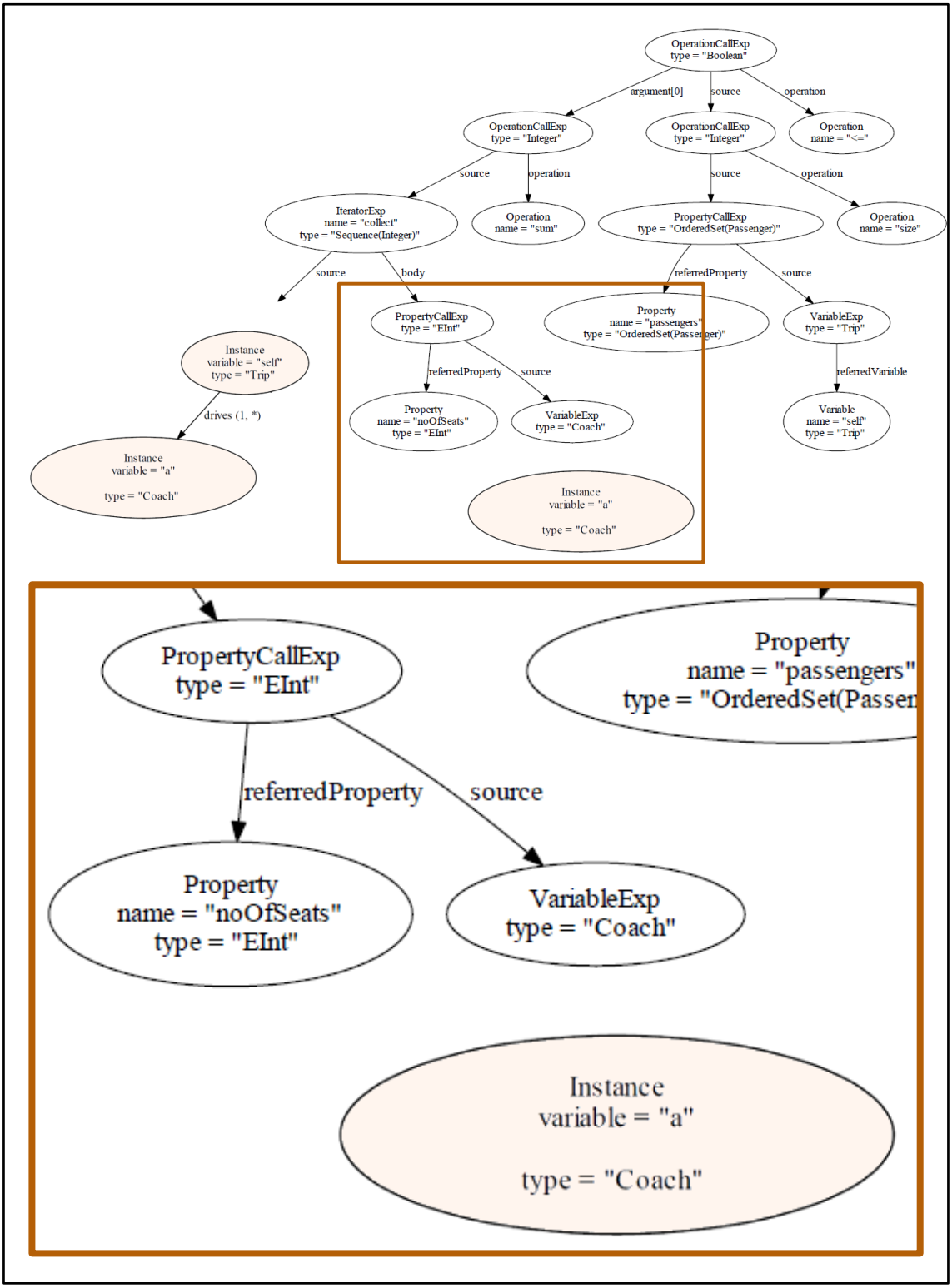


Fig 50: Proceso de creación de un IT. Paso 3, procesado de nodo Variable.

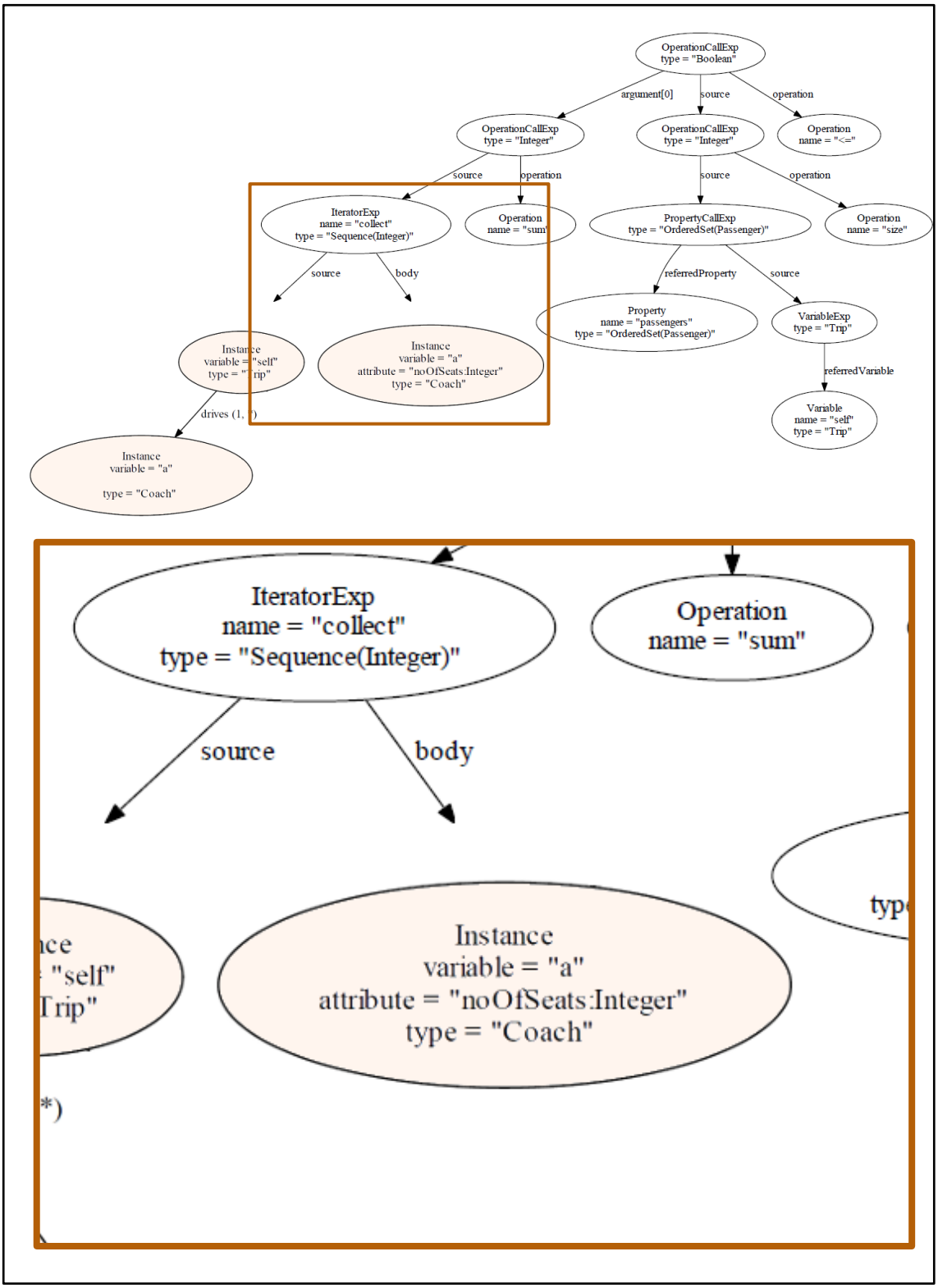


Fig 51: Proceso de creación de un IT. Paso 4, procesado de nodo PropertyCallExp.

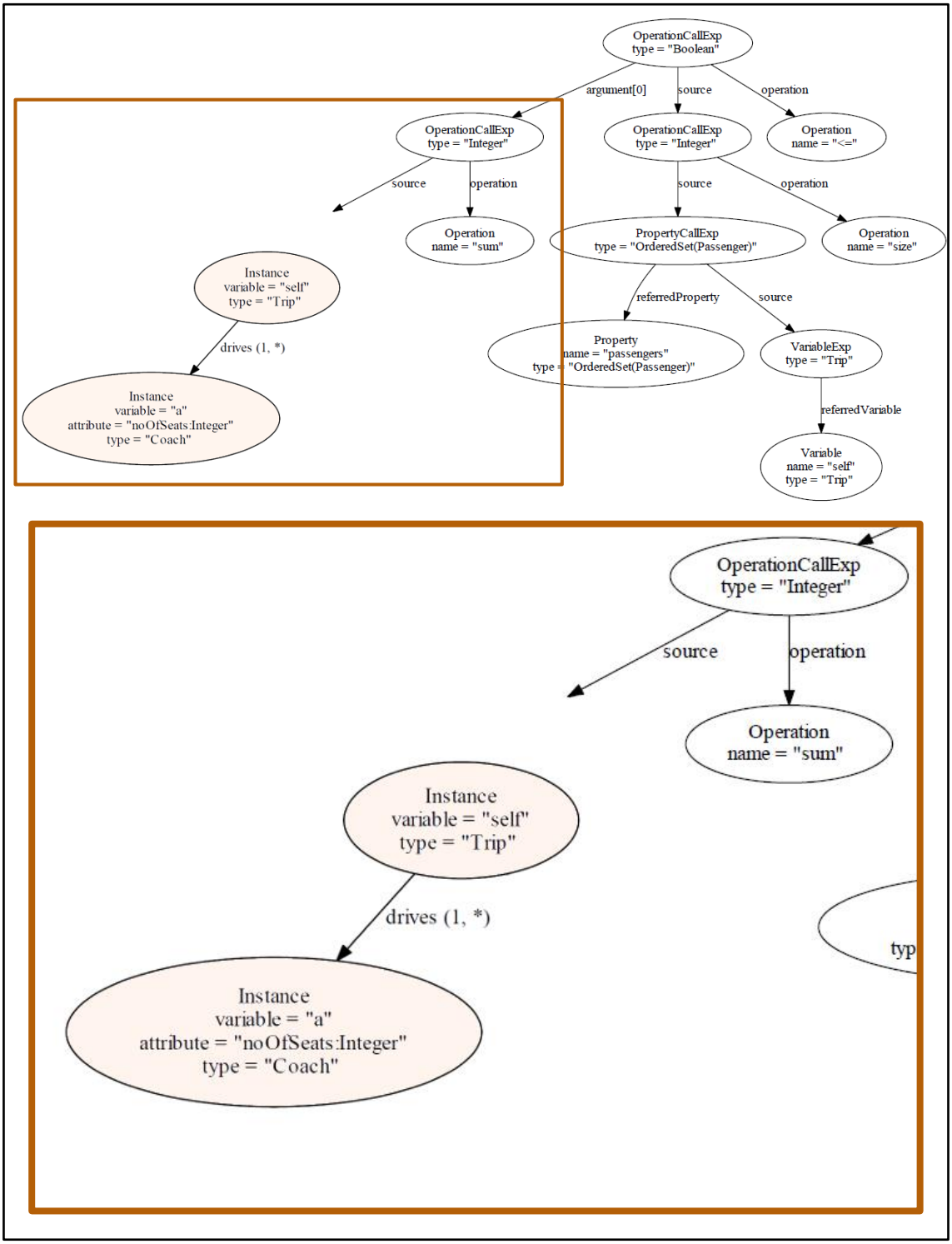


Fig 52: Proceso de creación de un IT. Paso 5, procesado de nodo IteratorExp.

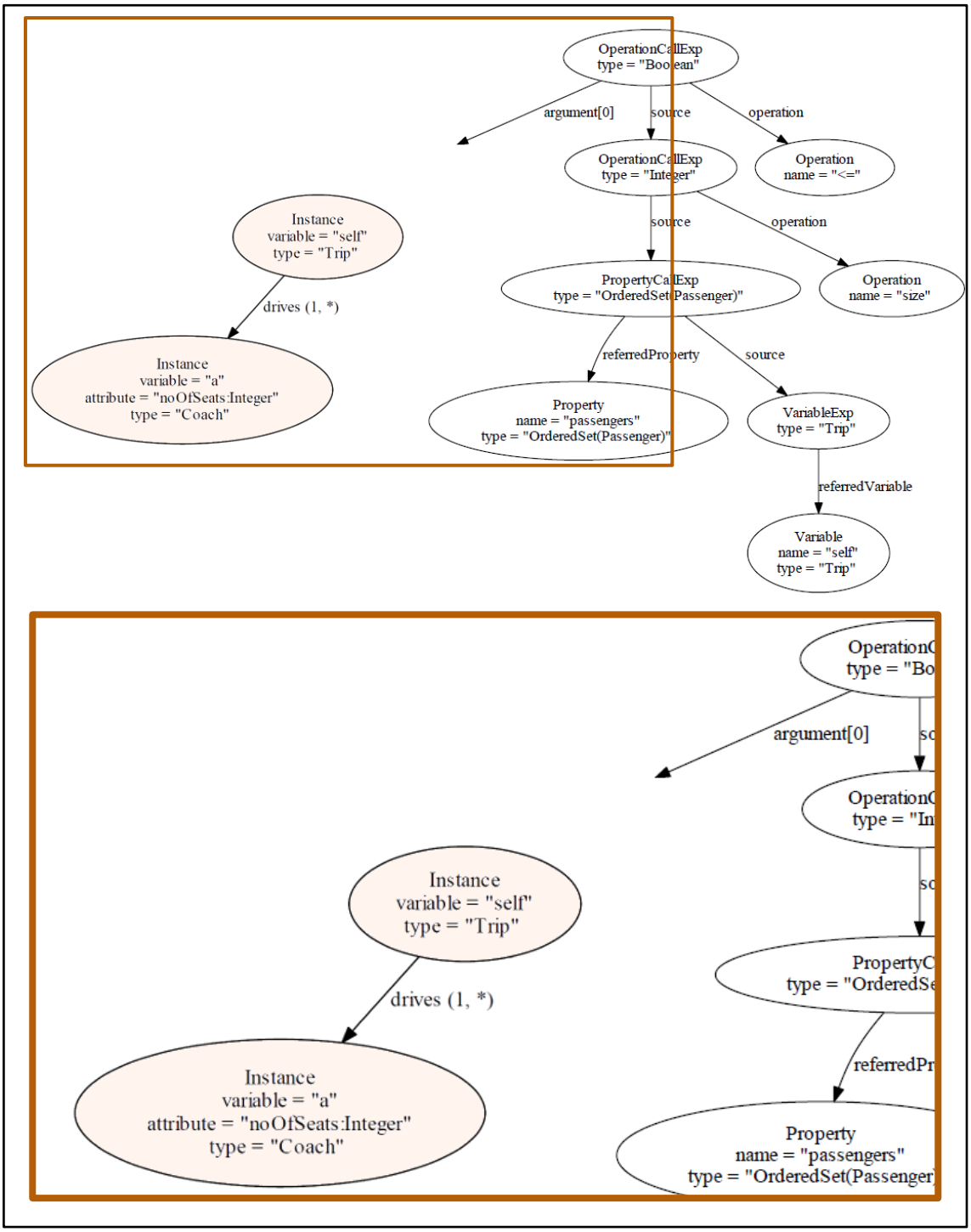


Fig 53: Proceso de creación de un IT. Paso 6, procesado de nodo OperationCallExp.

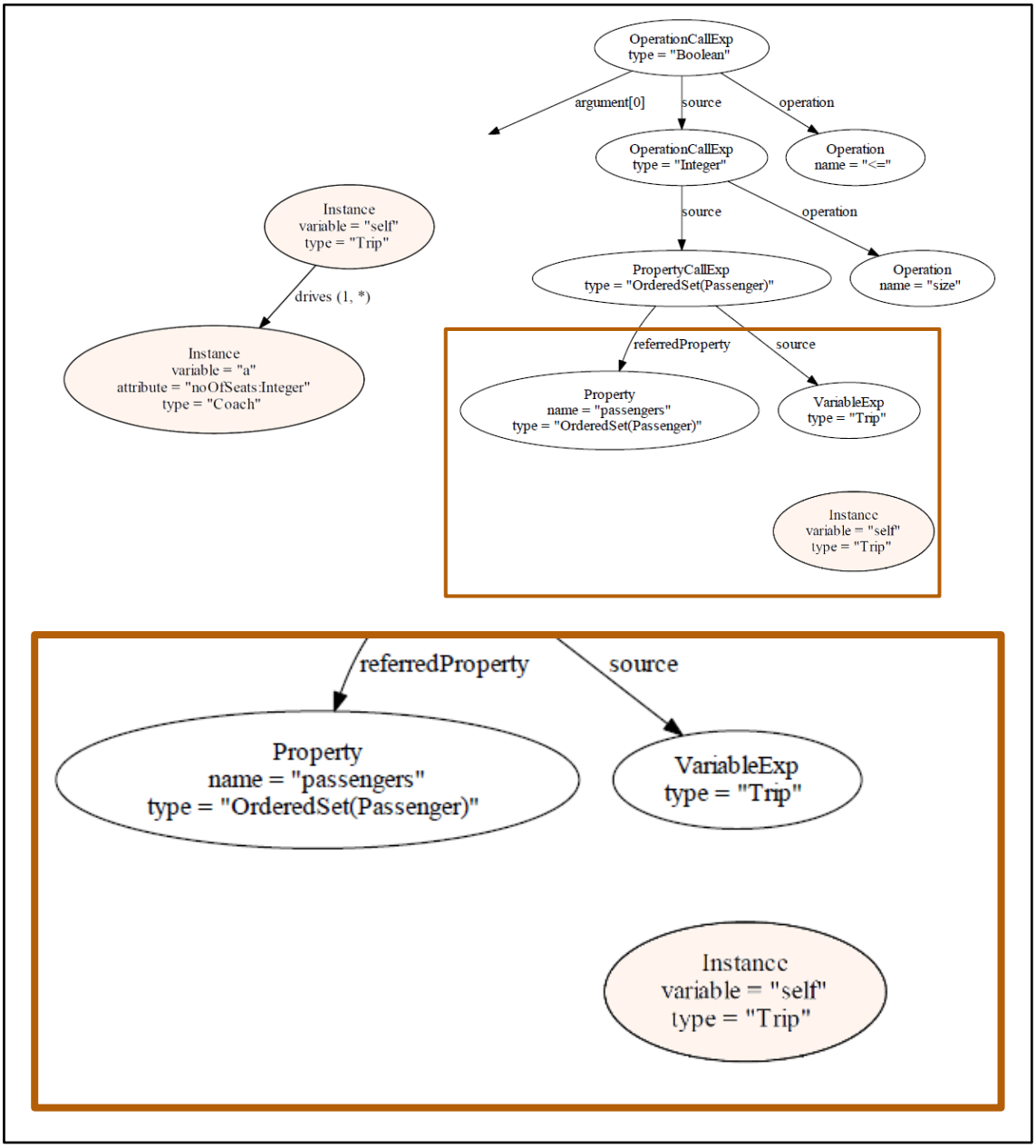


Fig 54: Proceso de creación de un IT. Paso 7, procesado de nodo Variable.

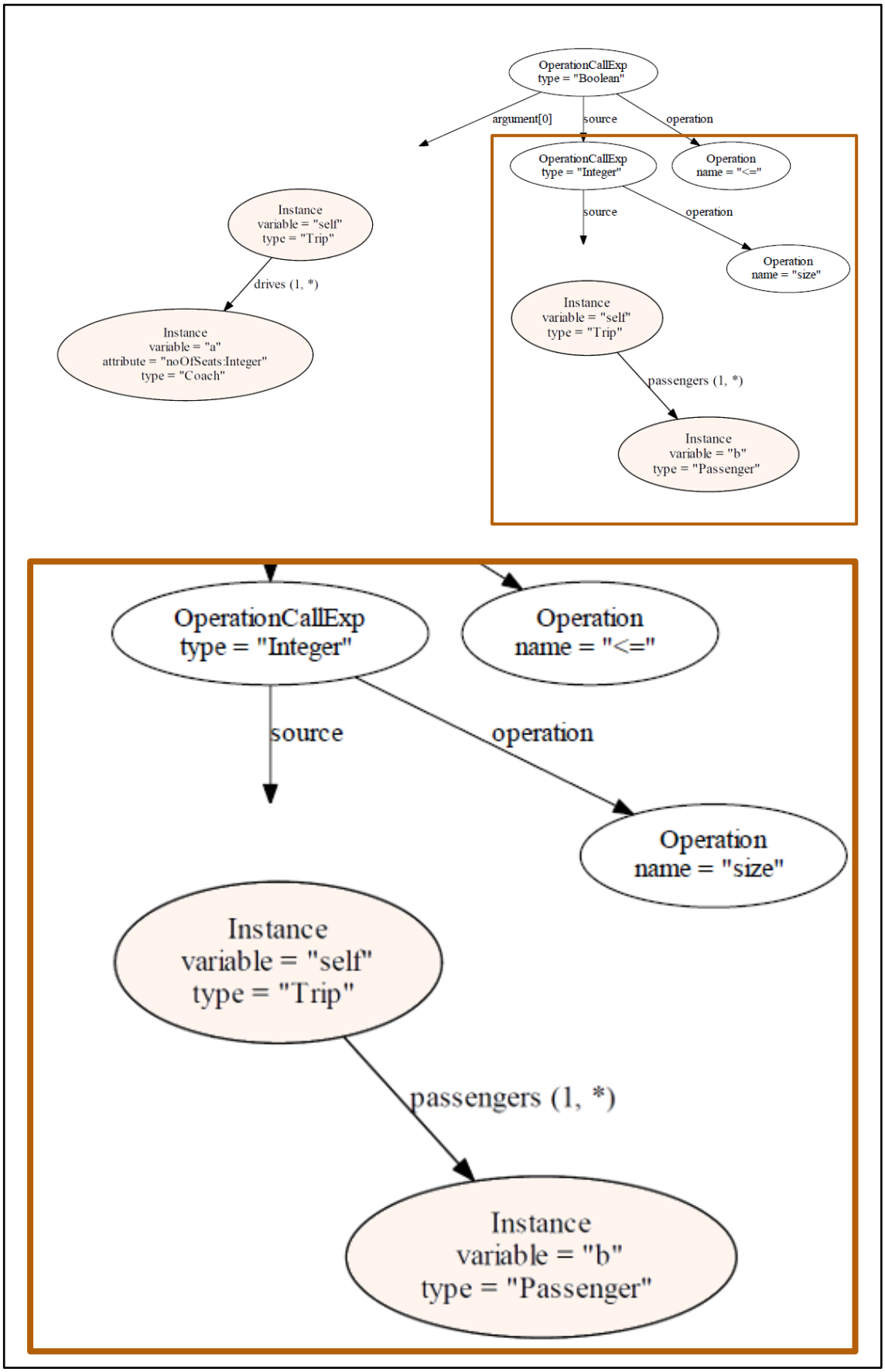


Fig 55: Proceso de creación de un IT. Paso 8, procesado de nodo PropertyCallExp.

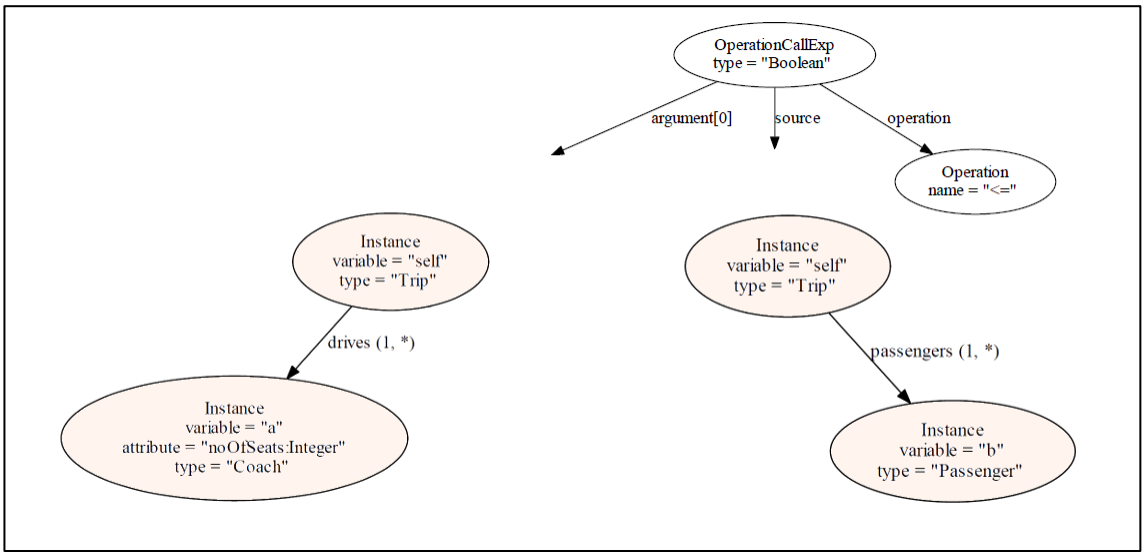


Fig 56: Proceso de creación de un IT. Paso 9, procesado de nodo OperationCallExp.

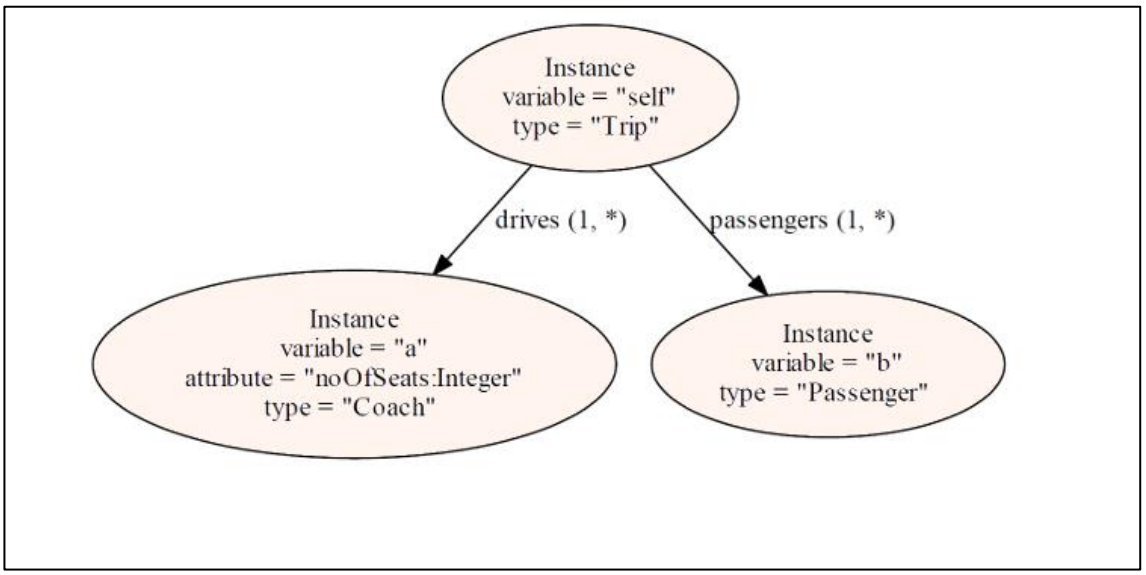


Fig 57: Proceso de creación de un IT. Paso 10, procesado de nodo OperationCallExp.

Como se puede apreciar, el IT final posee únicamente una raíz y dos nodos hijos, lo que hace esta representación mucho más sencilla y legible que el AST original. En él se puede apreciar claramente que hay 3 clases distintas implicadas en la evaluación de esta restricción, *Trip*, *Coach* y *Passenger*. Tanto *Coach* como *Passenger* son accedidas desde el contexto *Trip* a través de relaciones que indica que puede haber entre una y muchas instancias de ambas.

Este mismo proceso puede ser aplicado a todas las restricciones del modelo. En la Fig 58 se puede observar que muchas restricciones producen IT extremadamente simples. Dos de ellos consisten en un único nodo raíz, tres tienen un único hijo, y solamente hay uno con tres nodos, y otro con cuatro respectivamente. Es interesante resaltar que la complejidad de los IT producidos no siempre está relacionada con la longitud de las expresiones OCL o la complejidad de las operaciones implicadas. Restricciones aparentemente

complejas como *ticketNumberPositive* o *correctAgeChild2* producen árboles relativamente simples. La variedad de instancias a las que es necesario acceder en esas restricciones es por ejemplo menor que en restricciones como *enoughTickets*, que a pesar de utilizar operaciones más simples, requieren acceder a una mayor cantidad de instancias.

Comprender las clases implicadas en la evaluación de una restricción no siempre es intuitivo y requiere comprobar el modelo de clases de forma frecuente. En análisis de las navegaciones realizadas en una restricción se ve condicionada por los nombres elegidos para las relaciones, ya que son el único indicador presente en la expresión. Esta representación gráfica no solo es útil para el procesamiento automático de las restricciones, sino que también puede servir para documentar las restricciones, comprenderlas mejor y analizar cómo modificarlas en caso de necesitarlo.

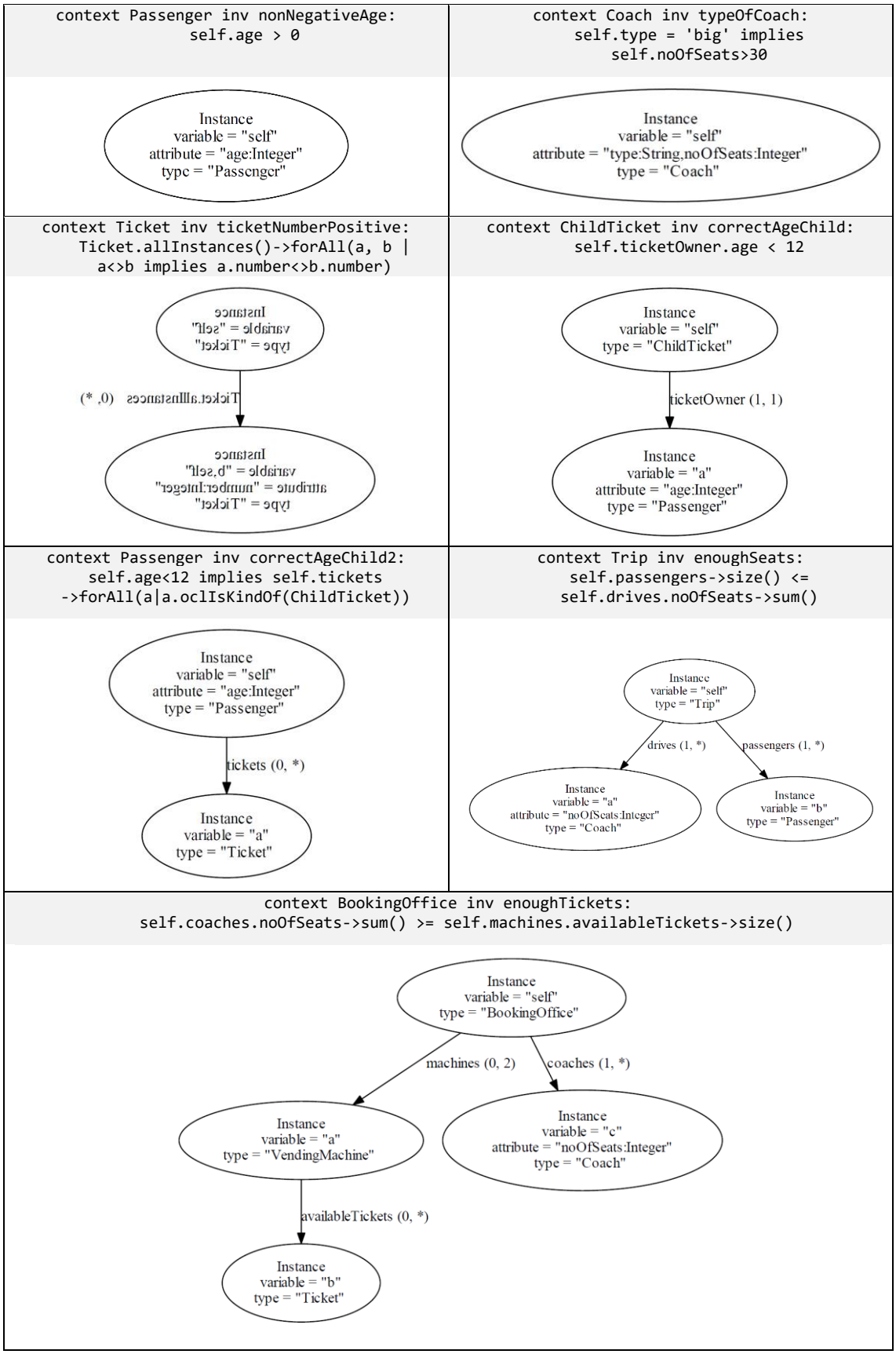


Fig 58: IT de cada una de las restricciones propuestas para el caso de ejemplo.

11 EXTRACCIÓN DE MÉTRICAS Y CLASIFICACIÓN AUTOMÁTICA DE LAS RESTRICCIONES

En este capítulo se describirá la extracción de métricas en base a los árboles de instancias de una restricción, y los criterios utilizados para clasificarla. Esta clasificación tiene dos objetivos, por un lado, identificar el tipo de restricción en función de los elementos a los que afecta, y por otro evaluar su nivel de dependencia respecto al servidor. Con esta información es posible detectar automáticamente qué restricciones no son aplicables al cliente, cuales lo son de forma totalmente independiente del servidor, y cuáles son aplicables, pero pueden requerir comunicación con el servidor. Este último tipo de restricción requiere de intervención humana para identificar su nivel de dependencia y tomar decisiones respecto a cómo manejarlas. Para ello la combinación del árbol de instancias, las métricas de la restricción, y su clasificación puede ayudar al diseñador en la toma de decisiones.

11.1 Extracción de métricas de los árboles de instancia

Los árboles de instancias no solo permiten al diseñador visualizar fácilmente cómo accede una restricción a los elementos del modelo de dominio, sino que recorriendo sus nodos es posible realizar todo tipo de cálculos.

Para la clasificación de las restricciones, se considerará la aplicación de las siguientes métricas:

— Número mínimo de instancias involucradas.

— Número máximo de instancias involucradas.

— Clases accedidas.

— Atributos accedidos.

En muchos casos es imposible saber cuántas instancias pueden verse involucradas en la evaluación de una restricción, ya que depende del estado que el grafo de objetos en el momento en que se realiza la evaluación en tiempo de ejecución. Sin embargo, conociendo el modelo de dominio y las navegaciones realizadas por la restricción, es posible conocer el rango entre la cantidad mínima y máxima de instancias involucradas. Esta información es importante a la hora de determinar la potencial dependencia que la restricción tiene respecto al servidor, y también puede dar al diseñador una indicación del posible coste computacional de evaluar la restricción. También es importante conocer qué clases y atributos se ven afectados por la restricción a la hora de realizar la clasificación.

El cálculo del número mínimo y máximo de instancias involucradas se puede describir como una función que recibe un nodo de instancias, y calcula su cardinalidad mínima o máxima según proceda. Este cálculo se realiza de forma recursiva, calculando el valor para ese nodo, y aplicando la función a sus hijos inmediatos. Para obtener el número mínimo de instancias de un nodo, se obtiene la cantidad de nombres de variables contenidas en dicho nodo. A este valor se le suman los resultados de aplicar esta misma función a cada uno de sus hijos y multiplicando dichos valores obtenidos por la cardinalidad mínima de la relación que les une a ellos. Descrita de modo informal, la función sería

$$min(Nodo) = Nº\ Variables\ en\ nodo + \sum hijos\ (min(hijo) \cdot cardinalidad\ mínima(relaciónHaciaHijo))$$

Descripción Formal 3: Cálculo del número mínimo de instancias.

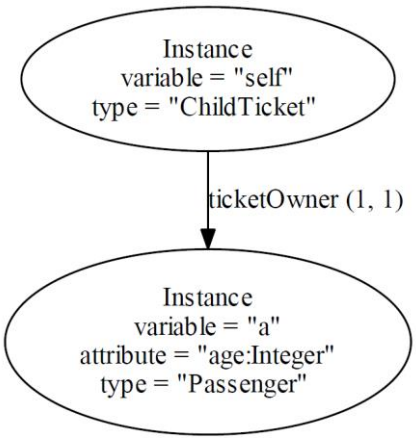
El cálculo del número máximo de instancias se obtiene del mismo modo, pero multiplicando el valor máximo de los nodos hijos por la cardinalidad máxima.

$$max(Nodo) = Nº\ Variables\ en\ nodo + \sum hijos\ (max(hijo) \cdot cardinalidad\ máxima(relaciónHaciaHijo))$$

Descripción Formal 4: Cálculo del número máximo de instancias.

Las clases y atributos accedidos son fáciles de detectar, basta recorrer el árbol de instancia y observar la clase de cada nodo, y qué atributos se acceden en cada caso.

Es posible aplicar estos cálculos sobre alguna de las restricciones del caso de ejemplo, como se puede observar en las Fig 59 y Fig 60.



min(raíz)

= 1 + (min(ticketOwner) · cardinalidadMínima(ticketOwner))

= 1 + (1 · 1) = **2**

max(raíz)

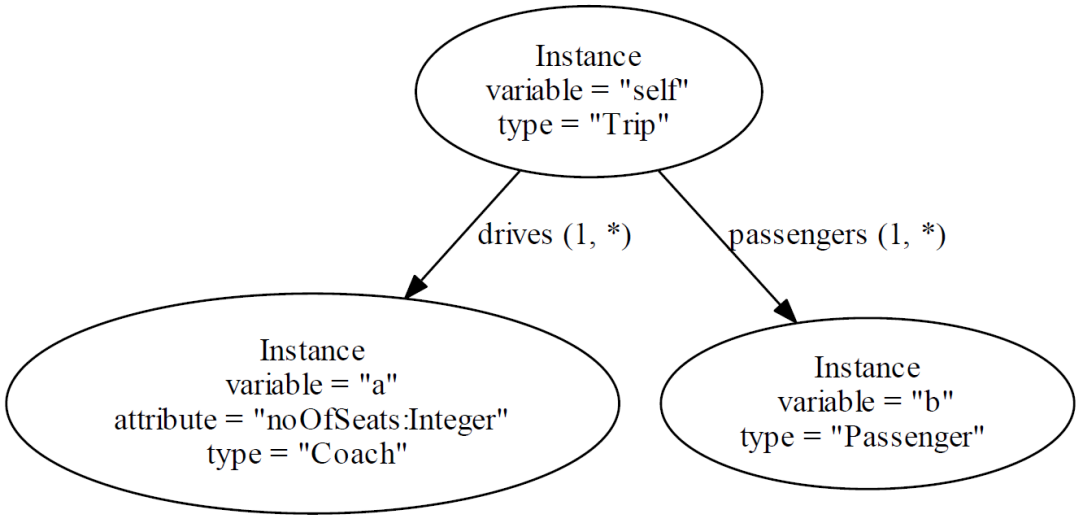
= 1 + (max(ticketOwner) · cardinalidadMáxima(ticketOwner))

= 1 + (1 · 1) = **2**

Clases = **ChildTicket, Passenger**

Atributos = **Passenger.age:Integer**

Fig 59: Ejemplo de cálculo de métricas para el árbol de instancia de la restricción context ChildTicket inv correctAgeChild: self.ticketOwner.age < 12 .



min(raíz)

= 1 + (min(drives) · cardinalidadMínima(drives) + min(passengers) · cardinalidadMínima(passengers))

= 1 + (1 · 1 + 1 · 1) = **3**

max(raíz)

= 1 + (max(drives) · cardinalidadMáxima(drives) + max(passengers) · cardinalidadMáxima(passengers))

= 1 + (1 · * + 1 · *) = *****

Clases = **Trip, Coach, Passenger**

Atributos = **Coach.noOfSeats:Integer**

Fig 60: Ejemplo de cálculo de métricas para el árbol de instancia de la restricción context Trip inv enoughSeats:self.passengers->size() <= self.drives.noOfSeats->sum() .

11.2 Clasificación de las restricciones respecto a su tipo

No todas las restricciones tienen el mismo alcance. Algunas evalúan solo un atributo de un objeto, mientras que otras afectan a diversos elementos del modelo de dominio. Tal como se mencionó en el Capítulo 2, autores como Haan y Koppelaars [23], Louwsma et al. [6], Liang y Jianling [7], Teniente y Cabot [24], o Miliauskaite y Nemuraite [25], han propuesto clasificaciones basadas en el número de elementos del modelo afectados. Aunque existen diferencias entre ellas y su nomenclatura y alcance pueden variar, todas comparten criterios similares y son compatibles entre ellas. Siguiendo este criterio, es posible clasificarlas de la siguiente forma:

- **Restricción de atributo:** Una restricción que accede únicamente a un atributo de un objeto.
- **Restricción de objeto:** Una restricción que accede a varios atributos, pero todos ellos contenidos en el mismo objeto.
- **Restricción de clase:** Una restricción que accede a más de un objeto, pero todos ellos de la misma clase (expresiones con *allInstances* o asociaciones reflexivas).
- **Restricción de dominio:** Una restricción que accede a elementos de más de un objeto, de los cuales no todos ellos son de la misma clase.

Para clasificar las restricciones en uno de estos cuatro tipos, se utilizarán las métricas descritas en el apartado anterior.

Tipo de restricción	**Nº Máximo de instancias**	**Nº de atributos**	**Nº de clases**
Atributo	=1	=1	=1
Objeto	=1	>1	=1
Clase	>1	-	=1
Dominio	>1	-	>1

Tabla 4: Criterios para la clasificación de restricciones por tipo, basados en las métricas que se extraen de los árboles de instancias.

Aplicando esta clasificación sobre algunas de las restricciones del caso de ejemplo se puede observar cómo se utilizan las métricas para determinar el tipo.

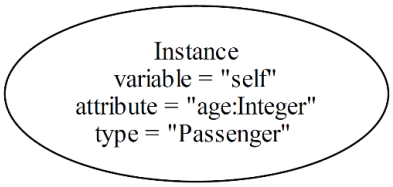
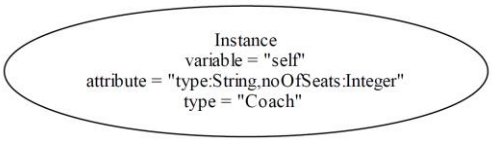
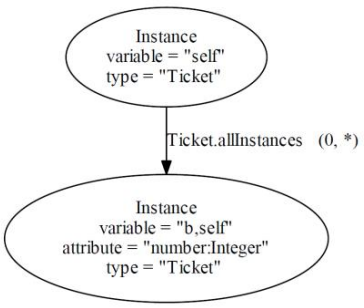
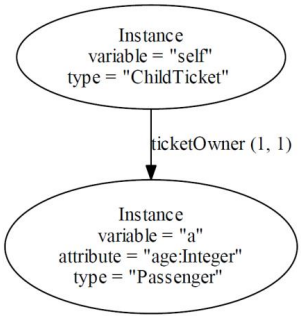
Tipo de restricción	Árbol de instancia de ejemplo	Nº Máximo de instancias	Nº de atributos	Nº de clases
Atributo	`context Passenger inv nonNegativeAge: self.age > 0` 	= 1	= 1	= 1
Objeto	`context Coach inv typeOfCoach: self.type = 'big' implies self.noOfSeats>30` 	= 1	= 2	= 1
Clase	`context Ticket inv ticketNumberPositive: Ticket.allInstances()->forAll(a, b | a<>b implies a.number<>b.number)` 	= *	= 1	= 1
Dominio	`context ChildTicket inv correctAgeChild: self.ticketOwner.age < 12` 	= 2	= 1	= 2

Tabla 5: Ejemplos de restricciones de atributo, objeto, clase y dominio, junto con sus métricas.

Esta clasificación proporciona una idea inmediata de lo extenso que es el rango de acción de una restricción respecto al modelo. Las restricciones de atributo siempre serán más granulares que las de dominio, por ejemplo. Dado que cuantos menos elementos se ven afectados por una restricción es más probable que esta puede ser aplicable en el cliente. Esta clasificación puede, por tanto, ayudar al diseñador a identificar más rápidamente qué restricciones del servidor pueden ser problemáticas a la hora de tratar de aplicarlas en el cliente.

La clasificación por tipo es independiente del tipo de CDM que se haya decidido diseñar. En el siguiente apartado se detallará una clasificación que tiene en cuenta el CDM al que tratan de aplicarse.

11.3 Clasificación de las restricciones respecto a su dependencia con el servidor

Las restricciones del GDM pueden ser clasificadas por tipo, pero solo esa información no es suficiente para decidir cuáles de ellas pueden ser aplicadas en el cliente. Una vez se ha generado el CDM y se dispone el modelo de dominio para el cliente, es posible clasificar las restricciones en función de su nivel de dependencia con el servidor. Esto ayudará a detectar automáticamente qué restricciones son evaluables en el cliente sin necesidad de comunicarse con el servidor, así como detectar aquellas que no pueden aplicarse en el cliente. Al mismo tiempo, habrá restricciones que tienen el potencial de ser aplicadas en el cliente, pero pueden requerir comunicación con el servidor.

En este caso la clasificación se hace no sólo en base a las métricas extraídas de las restricciones, sino que también evalúa esta información con el CDM para determinar su nivel de independencia cuando son aplicadas en ese cliente. La clasificación comprende las siguientes tres categorías:

— **Independiente del servidor**: Todos los elementos necesarios para evaluar la restricción en tiempo de ejecución se encuentran siempre en el cliente. Nunca es necesaria ningún tipo de comunicación con el servidor.

— **Potencialmente dependiente del servidor**: El modelo de dominio del cliente dispone de todas las clases necesarias para evaluar la restricción en tiempo de ejecución. Sin embargo, es posible que algunas de las instancias requeridas no se encuentren en el cliente y sea necesario realizar peticiones al servidor.

— **Dependiente del servidor**: Algunas de las clases necesarias para evaluar esta restricción en tiempo de ejecución nunca se encontrarán en el cliente, por lo que la restricción debe evaluarse en el servidor siempre.

Para que una restricción no se considere dependiente del servidor debe cumplir un requisito: todas las clases mencionadas en la restricción deben

encontrarse en el modelo de dominio del cliente. Si en el CDM no existe una de las clases involucradas en la restricción, ésta no podrá ser evaluada en el cliente.

Si todos los elementos de la restricción se encuentran en el CDM, entonces hay que discernir si se trata de una restricción independiente del servidor, o potencialmente dependiente. Las restricciones de atributo u objeto siempre serán independientes del servidor. Estas únicamente acceden a un objeto para realizar la evaluación, por lo que actuarán sobre los objetos presentes en el cliente sin necesidad de comunicación con el servidor.

Sin embargo, las restricciones de clase o de dominio, que acceden a la información de varios objetos, se clasificarán como potencialmente dependientes. En esos casos es posible realizar la evaluación en el cliente, pero si éste no dispone del grafo de instancias completo que permite realizar la verificación, es necesario hacer una petición al servidor para recibir o actualizar datos. La información proporcionada por la restricción y los modelos de dominio de cliente y servidor no son suficientes en estos casos para identificar de forma automatizada el tipo de comunicación que va a existir entre CDM y GDM, ni si todas las instancias necesarias estarán disponibles localmente, por lo que no es posible anticipar de forma automatizada si esta restricción podrá ser evaluada independientemente del servidor. Sin embargo, detectar las restricciones en esta situación ayudará al diseñador en el análisis del caso y la toma de decisiones, apoyándose en la información proporcionada por el árbol de instancias y sus métricas.

A continuación, se muestra cómo clasificar las restricciones en uno de estos tres tipos basándose en las métricas y los elementos existentes en el CDM

Clasificación de dependencia respecto al servidor	**Nº Máximo de instancias**	**Todas las clases existen en el CDM**	**Tipos de restricción posible**
Independiente	=1	Sí	Atributo, Objeto
Potencialmente dependiente	>1	Sí	Clase, Dominio
Dependiente	-	No	Atributo, Objeto, Clase, Dominio

Tabla 6: Criterios para la clasificación de restricciones por nivel de dependencia, basados en las métricas extraídas.

Aplicando esta clasificación sobre algunas de las restricciones del caso de ejemplo se puede observar cómo se utilizan las métricas para determinar el tipo.

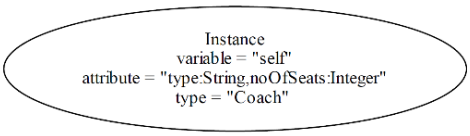
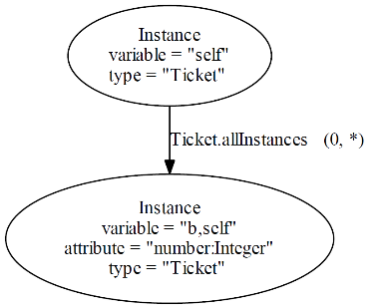
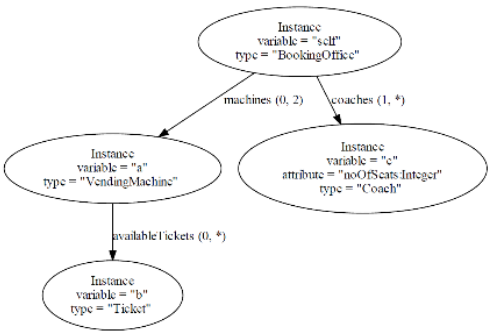
Tipo de restricción	Árbol de instancia de ejemplo	Nº Máximo de instancias	Todas las clases existen en el CDM
Independiente	`context Coach inv typeOfCoach: self.type = 'big' implies self.noOfSeats>30` 	**= 1**	**Sí**
Potencialmente dependiente	`context Ticket inv ticketNumberPositive: Ticket.allInstances()->forAll(a, b | a<>b implies a.number<>b.number)` 	**= 2**	**Sí**
Dependiente	`context BookingOffice inv enoughTickets: self.coaches.noOfSeats->sum() >= self.machines.availableTickets->size()` 	**= ***	**No**

Tabla 7: Ejemplos de restricciones para cada nivel de dependencia, junto con las métricas relevantes para esta clasificación.

11.4 Caso de ejemplo: Extracción de métricas y clasificación de las restricciones

Estos criterios se pueden aplicar una vez se ha creado el CDM y los árboles de instancia. Dado que los criterios de clasificación están claramente definidos, es posible obtener esta clasificación de forma totalmente automatizada. A continuación, se enumeran las restricciones del caso de ejemplo, junto con sus métricas y su clasificación.

En este caso, el diseñador puede saber que las dos primeras restricciones pueden aplicarse al CDM sin ninguna modificación, y que la última no. Para las restricciones entre la 3 a la 6, pueden aplicarse en el cliente, pero el diseñador deberá discernir en cada caso si requerirán hacer peticiones al servidor, y en caso de que así sea, estudiar si aun así es conveniente realizar la evaluación en el cliente, o delegarla completamente al servidor.

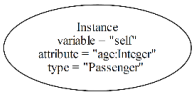
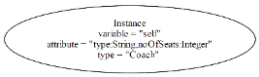
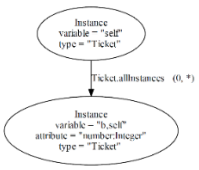
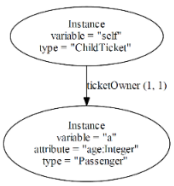
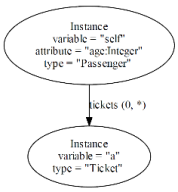
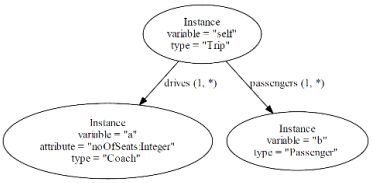
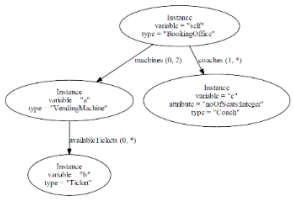
Restricción	mín inst	máx inst	Atributos	Clases	Tipo	Dependencia
Restricción 1: nonNegativeAge 	**1**	**1**	**Passenger:age**	**Passenger**	**Atributo**	**Independiente**
Restricción 2: typeOfCoach 	**1**	**1**	**Coach:type** **Coach:noOfSeats**	**Coach**	**Objeto**	**Independiente**
Restricción 3: ticketNumberPositive 	**1**	*****	**Ticket:number**	**Ticket**	**Clase**	**Potencialmente dependiente**
Restricción 4: correctAgeChild 	**2**	**2**	**Passenger:age**	**ChildTicket** **Passenger**	**Dominio**	**Potencialmente dependiente**
Restricción 5: correctAgeChild2 	**1**	*****	**Passenger:age**	**Passenger** **Ticket**	**Dominio**	**Potencialmente dependiente**
Restricción 6: enoughSeats 	**3**	*****	**Coach:noOfSeats**	**Trip** **Coach** **Passenger**	**Dominio**	**Potencialmente dependiente**
Restricción 7: enoughTickets 	**2**	*****	**Coach:noOfSeats**	**BookingOffice** **Coach** **VendingMachine** **Ticket**	**Dominio**	**Dependiente**

Tabla 8: Tabla completa con los árboles de instancias del caso de ejemplo, sus métricas, el tipo de restricción y su nivel de dependencia.

12 FORMALIZACIÓN

Los capítulos previos se han centrado en describir el método al completo de un modo informal que facilite comprender la idea general. Aunque los modelos de clases UML y las restricciones OCL han demostrado ser herramientas potentes para la definición de modelos, no están diseñadas para representar con claridad varios de los aspectos del problema que tratamos de resolver. Los modelos UML no son lenguajes formales y su ambigüedad semántica, si bien puede ser útil para su uso práctico, presenta dificultades a la hora de razonar acerca del problema o presentar bases lógicas sobre las que clasificar las restricciones.

Para dar una respuesta al problema de una forma clara y no ambigua es necesario disponer de un modelo formal. Este debería ser además fácil de traducir a los estándares y herramientas utilizados en la industria, y viceversa. También debería permitir tratar de forma homogénea los modelos de clases UML y las restricciones OCL.

Dado que los modelos UML son descritos gráficamente mediante nodos y aristas anotados, es viable traducirlo a teoría de grafos. Respecto a las restricciones, ya se han introducido los árboles de instancias, que son también un tipo de grafo. El modelo formal debería respetar y ser fácilmente traducible a los principios de los estándares UML y OCL, pero al mismo tiempo ser suficientemente general para poder ser adaptado a otros modelos basados en representaciones gráficas y predicados lógicos, como podría ser el caso de diagramas entidad-relación y SQL. Otras aproximaciones como las propuestas por el domain-driven design definen elementos como agregados, value-objects y entidades que pueden ser expresados de forma gráfica, y adaptables también a esta formalización.

12.1 Notación

La notación utilizada está basada en teoría de conjuntos y grafos. Se definirán los elementos de un conjunto utilizando la siguiente notación:

$$SET = \{a, b, c, ...\}$$

Descripción Formal 5: Notación para elementos de conjuntos.

Gran parte de los conjuntos que se definirán estarán formados por tuplas, y estas se definirán utilizando la siguiente notación:

$$\forall \text{ element} \in SET \text{ elemento} = (a_{element}, b_{element})$$

Descripción Formal 6: Notación para definir conjuntos.

En esta notación los atributos de las tuplas en minúsculas representarán valores, mientras que aquellos en mayúsculas representarán conjuntos. Para referirse al valor contenido en el atributo de una tupla se utilizará la siguiente notación:

$$elemento[a_{element}]$$

Descripción Formal 7: Notación para el acceso a atributos de tuplas.

Para referirse a la cantidad de elementos en un conjunto se utilizará la siguiente notación:

$$|CONJUNTO|$$

Descripción Formal 8: Notación para el número de elementos de un conjunto.

12.2 Modelos de clases UML

Un modelo de clases puede ser formalizado como un grafo dirigido anotado, en el que cada clase es un nodo y cada propiedad de esa clase es un atributo de ese nodo. Las asociaciones, composiciones y relaciones de herencia pueden ser representadas como aristas con atributos que especifiquen el tipo, nombre y cardinalidad de la relación.

La formalización utilizada en este método está centrada en los atributos de las clases, e ignorará clases de asociación y otras relaciones como "uso". Añadir a la descripción formal otros elementos como métodos no haría la descripción más clara y resultaría en una descripción y formalización más compleja. Una vez el método está descrito, los aspectos de UML que se han dejado fuera de la formalización pueden ser aplicados siguiendo los mismos principios descritos. Otros autores han desarrollado formalizaciones que describen diferentes aspectos de UML [19][22][34][14] , y sus propuestas pueden ser fácilmente incorporadas a este método en caso de que una formalización más compleja fuera necesaria.

Los nodos del grafo representarán clases, los cuales tendrán atributos que representarán los atributos de la clase. Estos atributos tendrán un nombre y un

tipo. Un tipo es un conjunto que representa un conjunto de valores válidos. Por ejemplo INTEGER es el conjunto de números enteros, STRING el conjunto de todas las posibles cadenas de caracteres válidas, y BOOLEAN un conjunto con los valores TRUE y FALSE. TYPE es el conjunto de conjuntos que contienen todos los posibles tipos existentes, TYPE = {INTEGER, STRING, BOOLEAN, DOUBLE, ...}. Se definirá además un conjunto ATTRNAME para referirse al conjunto de todos los nombres válidos, para evitar la confusión con el conjunto STRING cuando se utiliza como tipo.

CLASS será el conjunto que contiene todas las clases posibles. Cada elemento c del conjunto CLASS es una tupla $\forall c \in CLASS$, $c = (n_c, A_c)$; siendo n_c el nombre de la clase $n_c \in ATTRNAME$, y A_c un conjunto de pares representando los atributos de la clase. Cada par perteneciente a A_c contiene un nombre que lo identifica dentro de la clase y un tipo, $\forall a \in A_c$, $a = (n_a, t_a)$, donde $n_a \in ATTRNAME$ y $t_a \in TYPE$.

Un modelo de clases puede ser representado como un grafo:

$$G_{cd} = (V_{cd}, E_{cd})$$

Descripción Formal 9: Grafo para representar modelos de clases.

En el grafo G_{cd}, V_{cd} es un conjunto de nodos que representan a las clases, y E_{cd} es el conjunto de aristas que representan a las relaciones entre clases. Cada elemento c del conjunto V_{cd} es un elemento perteneciente al conjunto CLASS, $V_{cd} \subseteq CLASS$, que representa cualquier clase posible. E_{cd} es un conjunto de aristas que representa las relaciones entre clases y está formado por la unión de dos conjuntos, uno por cada tipo de arista posible, $E_{cd} = A_{cd} \cup I_{cd}$. A_{cd} es el conjunto que representa a las asociaciones y I_{cd} es el conjunto que representa a las relaciones de herencia.

A_{cd} es el conjunto que representa las relaciones de asociación entre clases. Cada elemento de este conjunto es una tupla $\forall a \in A_{cd}$, $a = (n_a, o_a, d_a, l_a, u_a, c_a)$ donde $n_a \in ATTRNAME$ contiene el nombre de la relación, $o_a \in V_{cd}$ y $d_a \in V_{cd}$ son las clases de origen y destino respectivamente, $l_a \in INTEGER$ y $u_a \in INTEGER \cup *$ son las cardinalidades mínima y máxima de la relación (donde el símbolo * representa una cardinalidad “muchos”), y $c_a \in BOOLEAN$ indica si la asociación se trata de una relación de composición (TRUE) o no (FALSE).

I_{cd} es un conjunto que representa las relaciones de herencia entre clases. Cada elemento del conjunto es un par $\forall i \in I_{cd}$, $i = (c_i, p_i)$ que contiene a la clase hija y padre, $c_i \in V_{cd}$ and $p_i \in V_{cd}$ respectivamente.

La Fig 61 muestra un resumen de la formalización del modelo de clases al completo, mientras que la Descripción Formal 10 y la Descripción Formal 11 muestran parte de la formalización aplicada sobre el caso de ejemplo.

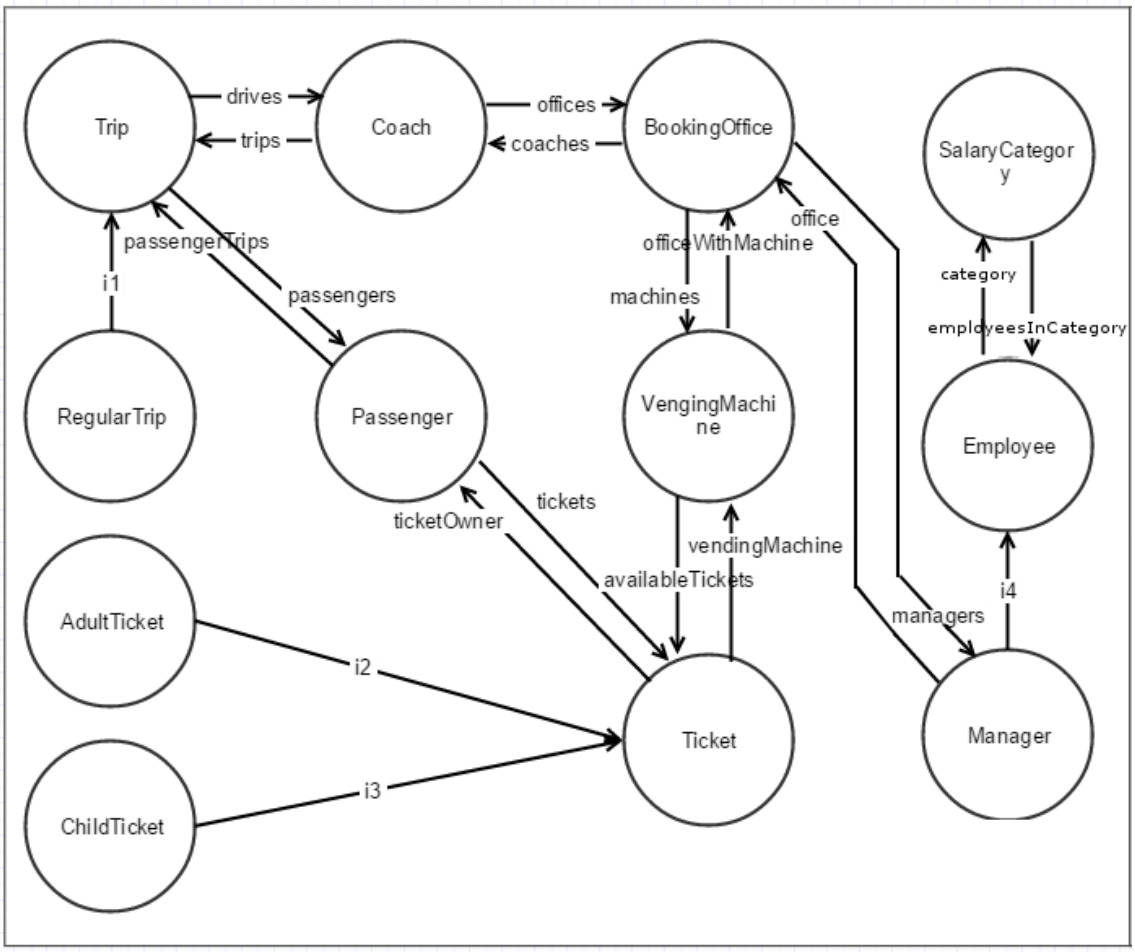


Fig 61 Formalización del modelo de clases en forma de grafo.

$G_{cd} = (V_{cd}, E_{cd})$	Grafo
$V_{cd} \subseteq CLASS$	Nodos: clases del modelo
$\forall c \in CLASS, c = (n_c, A_c)$	Clase
$n_c \in ATTRNAME$	Nombre de la clase
$\forall a \in A_c, a = (n_a, t_a)$	Atributo
$n_a \in ATTRNAME$	Nombre del atributo
$t_a \in TYPE$	Tipo del atributo
$E_{cd} = A_{cd} \cup I_{cd}$	Aristas: relaciones entre clases
$\forall a \in A_{cd}, a = (n_a, o_a, d_a, l_a, u_a, c_a)$	Relaciones de asociación
$n_a \in ATTRNAME$	Nombre de la asociación
$o_a \in V_{cd}$	Clase de origen
$d_a \in V_{cd}$	Clase de destino
$l_a \in INTEGER$	Cardinalidad mínima

$u_a \in INTEGER \cup *$	Cardinalidad máxima
$c_a \in BOOLEAN$	Es composición
$\forall i \in I_{cd}, i = (c_i, p_i)$	Relaciones de herencia
$c_i \in V_{cd}$	Clase hija
$p_i \in V_{cd}$	Clase padre

Descripción Formal 10: Formalización completa del modelo de clases.

G_{cd} = (V_{cd}, E_{cd}),
V_{cd} = {
(Trip, {(name, String), (origin, String), (destiny, String), ...)}),
(Coach, {(id, int), (name, String), (type, String), noOfSeats, int)}),
(SalaryCategory, {(type, String)})
},
E_{cd} = A_{cd} U I_{cd},
A_{cd} = {
(trips, Coach, Trip, 1, *, false),
(drives, Trip, Coach, 1, *, false),
...,
...,
(employeesInCategory, SalaryCategory, Employee, 1, *, false),
(category, SalaryCategory, Employee, 1, 1, false)
},
I_{cd} = {
i_1: (RegularTrip, Trip),
i_2: (AdultTicket, Ticket),
i_3: (ChildTicket, Ticket),
i_4: (Manager, Employee)
}

Descripción Formal 11: Formalización del grafo de la Fig 61.

12.3 Árboles de instancias

Para describir los elementos que se referencian desde una restricción OCL se utilizan árboles de instancias. Dado que el grafo sigue una estructura de árbol, la definición formal se basará en describir los nodos, los cuales, además de diversos atributos con la información necesaria, contendrán un conjunto que describirá sus hijos inmediatos, que serán también a su vez nodos.

Se definirá INSTANCENODE como el conjunto de todos los posibles nodos de instancias en un árbol de instancias. Cada elemento v_i perteneciente a INSTANCENODE es una tupla $\forall v_i \in$ INSTANCENODE, $v_i = (c_{vi}, V_{vi}, A_{vi}, CH_{vi})$ donde $c_{vi} \in$ CLASS es la clase de la instancia, V_{vi} es el conjunto de elementos ATTRNAME que representan las distintas variables que se usan para referirse a

las instancias, $\forall v \in V_{vi}$, $v \in$ ATTRNAME. A_{vi} es el conjunto de pares que representan los atributos que son accedidos en ese nodo. Cada uno de esos atributos contenidos en A_{vi} es un par $\forall a \in A_{in}$, $a = (n_a, t_a)$ donde $n_a \in$ ATTRNAME es el nombre del atributo y $t_a \in$ TYPE es su tipo. Los nodos de instancias solo contendrán aquellos atributos de la clase que son referenciados en la expresión OCL.

Cada elemento del conjunto CH_{vt} es una tupla representando cada nodo hijo del nodo de instancias y las aristas que los conectan, $\forall ch_{vi} \in CH_{vi}$, $ch_{vi} = (d_{ch}, n_{ch}, l_{ch}, u_{ch})$ donde $d_{ch} \in$ INSTANCENODE es el nodo hijo, $n_{ch} \in$ ATTRNAME es el nombre de la arista conectando ambos nodos, $l_{ch} \in$ INTEGER es la cardinalidad mínima de la relación entre ambas y $u_{ch} \in$ INTEGER $\cup$ * es la cardinalidad máxima.

Como se mencionó en capítulos anteriores, esta representación solo incluye los elementos afectados por la restricción, no el predicado lógico que se aplica sobre ellos. Aunque sería posible tratar de buscar una representación formal para esto, es algo fuera del alcance de este trabajo.

La Descripción Formal 12 muestra un resumen de la formalización del modelo de clases al completo.

$\forall v_i \in$ INSTANCENODE, $v_i = (c_{vi}, V_{vi}, A_{vi}, CH_{vi})$
- $c_{vi} \in$ CLASS
- $\forall v \in V_{vi}$, $v \in$ ATTRNAME
- $\forall a \in A_{in}$, $a = (n_a, t_a)$
  - $n_a \in$ ATTRNAME
  - $t_a \in$ TYPE
- $\forall ch_{vi} \in CH_{vi}$, $ch_{vi} = (d_{ch}, n_{ch}, l_{ch}, u_{ch})$
  - $d_{ch} \in$ INSTANCENODE
  - $n_{ch} \in$ ATTRNAME
  - $l_{ch} \in$ INTEGER
  - $u_{ch} \in$ INTEGER $\cup$ *

Descripción Formal 12: Formalización completa de los árboles de instancias.

12.4 Generación automática del CDM

La generación automática del cliente únicamente requiere como entrada una selección inicial de clases del GDM. Para describir formalmente el proceso de generación del CDM nos basaremos en la formalización ya realizada para los modelos de clases.

El criterio de corte es una selección de clases que deben estar presentes previamente en el GDM. Esta selección será definida en un conjunto ClassSelection $\subseteq V_{cd}$. El resultado final será otro grafo $G_{cd}' = (V_{cd}', E_{cd}')$ en el que las clases V_{cd}' y relaciones E_{cd}' serán generadas de forma recursiva, con una

clausula básica, cláusulas inductivas y una cláusula externa. En términos de *Model Slicing*, G_{cd}' sería el *slice* derivado del modelo original.

La cláusula básica es que la selección inicial de clases será un subconjunto del resultado final del CDM, que a su vez será también un subconjunto de las clases del GDM, ClassSelection $\subseteq V_{cd}' \subseteq V_{cd}$. Los atributos de las clases en V_{cd}' serán los mismos que en V_{cd}. Definido formalmente, si dos clases tienen el mismo nombre, implica que tienen también el mismo conjunto de atributos:

$$\forall c \in V_{cd}, \forall c' \in V_{cd}', c[n_c] = c'[n_c] \rightarrow A_c \in c = A_c' \in c'$$

Descripción Formal 13: Cláusula básica respecto al conjunto de atributos.

E′ es el conjunto de relaciones resultante de la unión de los conjuntos de relaciones de asociación y herencia, en los cuales tanto las clases de origen y de destino están contenidos en el conjunto V′:

$$\forall a \in A_{cd} / a[o_e] \in V_{cd}' \wedge \in V_{cd}' \rightarrow$$
$$\exists a' \in A_{cd}' / a' = a$$
$$\forall i \in I_{cd} / i[c_i] \in V_{cd}' \wedge i[p_i] \in V_{cd}' \rightarrow$$
$$\exists i' \in I_{cd}' / i' = i$$

Descripción Formal 14: Cláusula básica respecto al conjunto de relaciones.

Una vez se dispone de los elementos básicos del *slice*, las cláusulas inductivas cómo las relaciones definidas en E_{cd} añaden nuevos elementos a G_{cd}'. Por cada elemento presente en V_{cd}' que es hijo en una relación de herencia en E_{cd}, la clase padre de esa relación será también incluida en V_{cd}':

$$\forall i \in I_{cd}, i[c_i] \in V_{cd}' \rightarrow i[p_i] \in V_{cd}' \wedge i \in I_{cd}'$$

Descripción Formal 15: Cláusula inductiva respecto a relaciones de herencia.

Del mismo modo, por cada clase presente en V_{cd}' que forma parte del origen de una relación de composición en E_{cd}, esa relación y su clase destino también formarán parte de V′:

$$\forall c \in V_{cd}, \forall a \in A_{cd} / a[o_a] = c \wedge a[c_a] = \text{true},$$
$$a[o_a] \in V_{cd}' \rightarrow a[d_a] \in V_{cd}' \wedge c \in A_{cd}'$$

Descripción Formal 16: Cláusula inductiva respecto a relaciones de composición.

Finalmente, la cláusula externa es que los elementos de G_{cd}' se obtienen única y exclusivamente a partir de las cláusulas básicas e inductivas descritas anteriormente.

Si queremos aplicar estas cláusulas al caso de ejemplo con una selección inicial de clases ClassSelection = {*RegularTrip, Coach, Passenger, AdultTicket, ChildTicket*} el resultado será el mostrado en la Fig 62. Las clases padre *Trip* y *Ticket* serán añadidas iteractivamente mediante las cláusulas inductivas, así como sus relaciones con el resto de clases.

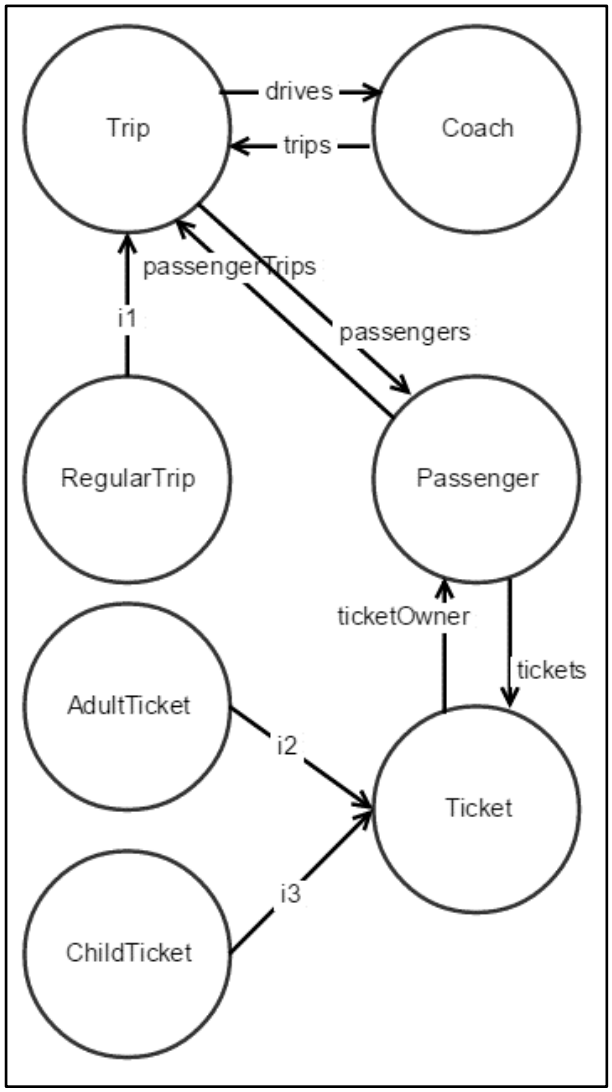


Fig 62: Grafo resultante tras la aplicación de las cláusulas para la generación del CDM sobre el Grafo de la Fig 61.

12.5 Extracción de restricciones implícitas

Los modelos de clases contienen numerosas restricciones implícitas relacionadas con la cardinalidad de las relaciones entre clases. Una vez definida la formalización de los modelos de clases y los árboles de instancias, es posible describir formalmente cómo generar árboles de instancias para las restricciones implícitas descritas en el grafo de clases.

Entre la definición de aristas de un grafo de clases está el conjunto A_{cd} que representa cada relación de asociación o composición. Por cada uno de los

elementos de ese conjunto un árbol de instancias puede ser creado describiendo la restricción de cardinalidad:

$$\forall a \in A_{cd}, \exists it \in IT$$

Descripción Formal 17: Relación entre relaciones del modelo de clases y árboles de instancias.

Cada árbol de instancias creado para la relación *a* se generará siguiendo las siguientes reglas. Como se ha descrito previamente, una relación *a* es una tupla $a = (n_a, o_a, d_a, l_a, u_a, c_a)$. A su vez, cada árbol de instancia se define como $it=(n_{it}, v_{it})$. Cada elemento *it* generado por cada elemento a_c se creará de la siguiente forma. La restricción tendrá el mismo nombre que la relación: $n_{it} = n_a$. La raíz del nodo de instancias será de la misma clase que la clase de origen de la relación, y únicamente tendrá un nodo hijo, de la misma clase que la clase destino de la relación:

$$\forall a \in A_{cd} \rightarrow \exists v_{it} \in it, ch_{vi} \in v_{it}[CH_{vi}] / v_{it}[c_{vi}] = a[o_a] \wedge d_{ch}[c_{vi}] = a[d_a]$$

Descripción Formal 18: Reglas para la creación de árboles de instancias.

La única arista entre la raíz y su hijo $ch_{vi}=(d_{ch}, n_{ch}, l_{ch}, u_{ch})$ tendrá el mismo nombre y cardinalidades que la relación original, donde: $n_{ch}=n_a$ $l_{ch}=l_a$ $u_{ch}=u_a$.

Disponer de todas las relaciones en la misma notación, facilitará la descripción formal de las métricas y clasificaciones.

12.6 Generación de árboles de instancias

El árbol AST de una expresión OCL tendrá diferentes tipos de nodos representando diferentes elementos de la expresión, tales como operaciones, llamadas a propiedades, declaraciones de variables, etc.

El método propuesto recorre este árbol en post-orden, generando árboles de instancias provisionales (PIT, *Provisional Instance Trees*). Cada vez que un nodo es recorrido, los PIT previamente generados para sus hijos son utilizados y unidos para la generación del nuevo PIT, tal y como se describió en el capítulo 0.

Dado que el proceso de unir PIT es importante, vamos a definir formalmente la función *merge* que a partir de dos nodos de instancias generará uno nuevo resultante de la unión de ambos. Con esta función definida será posible describir como se generan los diferentes PIT en función del tipo de nodo AST que se recorre.

Se define una función *merge* que recibe dos nodos de instancias v_{it} como parámetros, y devuelve un único nodo de instancias como resultado.

$$merge(v_a, v_b) \rightarrow v_m$$

Descripción Formal 19: Función merge.

Para que dos nodos de instancias puedan unirse, deben ser de la misma clase, y compartir al menos un nombre de variable.

$$v_a, v_b \in \text{INSTANCENODE}, v_a[c_{vi}] = v_b[c_{vi}] \land \exists a_{va} \in v_a[V_{vi}], \exists a_{vb} \in v_b[V_{vi}] \,/\, a_{va} = a_{vb}$$

Descripción Formal 20: Condiciones para la unión de nodos de instancias.

Si ambos comparten estas condiciones de partida, el conjunto de atributos y el conjunto de variables del nodo resultante será el resultado de la unión de esos mismos conjuntos en los nodos recibidos como parámetros.

$$\begin{aligned} &v_a, v_b, v_m \in \text{INSTANCENODE},\\ &v_m[A_{vi}] = v_a[A_{vi}] \cup v_b[A_{vi}],\\ &v_m[V_{vi}] = v_a[V_{vi}] \cup v_b[V_{vi}] \end{aligned}$$

Descripción Formal 21: Descripción de la unión de los atributos de los nodos de instancias.

El conjunto de nodos hijos del nodo resultante será igual a la unión de los conjuntos de hijos de los nodos recibidos como parámetros.

$$v_a, v_b, v_m \in \text{INSTANCENODE}, v_m[CH_{vi}] = v_a[CH_{vi}] \cup v_b[CH_{vi}]$$

Descripción Formal 22: Descripción de la inclusión de nodos hijos.

Finalmente, si ambos nodos comparten un hijo con el mismo tipo y nombre de arista, la función *merge* será aplicada sobre ambos recursivamente. La unión de los hijos repetidos se hará solo sobre los hijos inmediatos, ya que al aplicarse recursivamente este proceso continuará hacia los hijos consiguientes si fuese necesario.

$$\exists ch_a \in v_m[CH_{vi}], v_{cha} \in ch_a[d_{ch}], ch_b \in v_m[CH_{vi}], v_{chb} \in ch_b[d_{ch}] \,/\, ch_a[n_{ch}] = ch_b[n_{ch}] \land v_{cha}[c_{vi}] = v_{chb}[c_{vi}] \rightarrow \text{merge}\,(v_{cha}, v_{chb})$$

Descripción Formal 23: Descripción de la unión recursiva de nodos hijos.

Una vez descrita la función *merge*, se describirá formalmente el proceso de recorrer el árbol AST y generar los PIT. Solo se describirán aquellos tipos de nodo AST que tienen relevancia para este método. El resto son recorridos sin tener ningún efecto en el proceso, salvo el proceso de unir los PIT de los nodos hijos cuando se recorre su nodo padre.

Se definirá el conjunto ASTNODE como el conjunto de todos los posibles nodos AST, donde un nodo AST astn es una tupla $astn = (t_{astn}, A_{astn}, CH_{astn})$, donde $t_{astn} \in$ ATTRNAMES es el tipo de nodo AST según el metamodelo de OCL, cada elemento del conjunto A_{astn} es un atributo describiendo ciertas propiedades del nodo $\forall a \in A_{astn}$, $a = (n_a, t_a)$ donde $n_a \in$ ATTRNAME es el nombre del atributo, y $t_a \in$ TYPE es el tipo. Finalmente el conjunto CH_{astn} contiene todos los hijos de ese nodo AST, que son definidos mediante un par $\forall ch \in CH_{astm}$, $ch = (n, astc)$ donde n es el nombre de la arista uniendo padre e hijo $n \in$ ATTRNAMES, y astc ∈ ASTNODE es el nodo hijo.

La Descripción Formal 24 describe cada nodo AST relevante para este método. Es importante considerar que la lista no incluye todos los posibles atributos o hijos de los nodos, simplemente aquellos que son relevantes para el método.

-Variable = (A_{astn}{type, name}, CH_{astn}{initExpr(optional)})
-Property=(A_{astn}{type, name)})
-Operation=(A_{astn}{type, name)})
-TypeLiteralExp=(A_{astn} {type})
-PropertyCallExp = (A_{astn}{type}, CH_{astn}{source, property})
-IteratorExp = (A_{astn}{type}, CH_{astn}{source, body})
-OperationCallExp = (A_{astn}{type}, CH_{astn}{source, operation, args(optional)...})

Descripción Formal 24: Nodos AST relevantes para la creación de árboles de instancias.

Se define la función *processNode*(astn), donde astn es el nodo AST a procesar. Esta función devuelve el nodo raíz del árbol de instancia provisional $p_v \in$ INSTANCENODE generado tras procesar el nodo AST.

processNode(astn) $\rightarrow p_v$

Descripción Formal 25: Función processNode.

La función *processNode* devolverá resultados diferentes en función del tipo de nodo AST que reciba como parámetro. Dado que hay múltiples posibilidades a la hora de procesar cada tipo de nodo, se establecerán algunas consideraciones previas para simplificar la notación.

Cuando se procesa un nodo AST, puede ser necesario comprobar los PIT generados previamente por sus hijos. Definiremos la función *getChildInstanceNodes()* $\rightarrow P_v$, que devuelve como resultado el conjunto Pv, que contiene los nodos raíz de los PIT que fueron creados tras procesar cada uno de los nodos AST de los hijos del nodo que está siendo procesado actualmente. En algunos casos, puede ser necesario consultar el PIT generado por alguno de los hijos específicos. Para esas situaciones se define la función *getChildInstanceNode(edgeName)* $\rightarrow p_v$, donde *edgeName* es el nombre de la arista que conecta el nodo AST con su hijo, y p_v es el PIT resultante de ese nodo hijo. Cuando se aplica la función *processNode* sobre un nodo AST, esta se aplicará también en post-orden a todos sus hijos.

A la hora de definir el comportamiento de la función *processNode* ante los diferentes tipos de nodo AST, se utilizará la variable v para denotar el nuevo nodo de instancias PIT generado.

Durante este proceso, puede requerirse información proveniente del grafo de clases asociado a la restricción (por ejemplo, para consultar las cardinalidades de una relación). El grafo de clases será referenciado mediante la notación G_{cd}.

Cada nodo de instancias siempre tendrá al menos una variable refiriéndose a él. Dado que las expresiones OCL no siempre asignan variables explícitas a las instancias, en esos casos se generará una automáticamente para ese nodo. Para

esta tarea se define la función *generateImplicitVar()* → ATTRNAME, la cual generará un nombre de variable único para dicha instancia.

Además, cuando se procesan PIT, habrá situaciones en las que se dispone de un nodo raíz con un único hijo, que a su vez tiene un único hijo, etc. Este tipo de estructura se define como cadena de instancias, y en ocasiones será necesario consultar cuál es el último nodo de esta cadena. Para ello, se define la función *getLastChild*(v_i) → v_l, que recorrerá desde el nodo recibido v_i como parámetro hasta llegar al último hijo, y lo devolverá como v_l. En otras ocasiones, todos los nodos pertenecientes a una cadena de instancias deberán ser recolectados en un único conjunto V_i, para lo cual se define la función *collectAllChainInstanceChildren*(v_i) → V_i que devolverá todos los hijos sucesivos desde v_i. Se definirá también una función que devuelve un conjunto V_i. conteniendo todos los hijos inmediatos de un nodo v_i, pero no los sucesivos más abajo en la jerarquía del árbol, *collectAllDirectChildren*(v_i) → V_i. Estas tres funciones presentan las siguientes restricciones: las dos primeras funciones solo son válidas para cadenas de instancias, mientras que la última solo es válida para árboles con un solo nivel de hijos.

Con estas definiciones previas hechas, es posible comenzar a definir el comportamiento de la función *processNode* para cada tipo de nodo AST.

Variable

Cuando el nodo AST es de tipo *Variable*, generará un único PIT v. Tendrá la misma clase que el tipo mencionado en el nodo AST y su nombre de variable será el mismo que la propiedad *name* del AST. El PIT v generado es devuelto como el resultado de la función p_v.

processNode(Variable) → p_v:
v ∈ INSTANCENODE/
$v[c_{vi}]$ = variable[type] ∧
variable[name] ∈ $v[c_{vi}]$ ∧
p_v = v

Descripción Formal 26: Procesado de nodos AST de tipo Variable.

TypeLiteralExp

Un nodo AST TypeLiteralExp generará un único nodo de instancias *v*, con la misma clase que la descrita en el atributo *type* del nodo AST. TypeLiteralExp no incluye un nombre de variable que se pueda añadir al PIT, por lo que se generará uno único con la función *generateImplicitVar*.

processNode(TypeLiteralExp) → p_v:
v ∈ INSTANCENODE/
$v[c_{vi}]$ = variable[type] ∧
$v[A_{vi}]$ ∪ generateImplicitVar() ∧
p_v = v

Descripción Formal 27: Procesado de nodos AST de tipo TypeLiteralExp.

PropertyCallExp

Un nodo AST PropertyCallExp AST siempre tendrá dos hijos: etiquetados como *source* y *property* respectivamente. El nodo *property* no genera ningún PIT tras ser procesado. Debido a los tipos de nodos AST aceptados como hijo en el rol de *source*, el PIT generado siempre será una cadena de instancias. Si el hijo *property* es un tipo primitivo, un atributo con el nombre de la propiedad y su tipo será añadido al último hijo de la cadena de instancias que se generó para *source*. Si en cambio el hijo *property* es una clase, un nuevo nodo de instancias con un nombre de variable implícito será creado, y concatenado al final de la cadena de instancias de *source*. La arista uniendo este nuevo nodo tendrá el mismo nombre que el mostrado en el hijo *property*, y tendrá unas cardinalidades iguales a la arista de ese mismo nombre presente en el grafo de clases $E_{cd} \in G_{cd}$.

processNode(PropertyCallExp) → p_v:
    v_{p1} ∈ getChildInstanceNodes(),
    if property[type] ∈ TYPE →
        getLastChild(v_{p1})[A_{vi}] ∪ a_{vi} = (property[name], property[type]) ∧
        $p_v = v_{p1}$
    if property[type] ∈ CLASS →
        v ∈ INSTANCENODE,
        (∃a ∈ A_{cd} / a[n_a] = property [name]) ∧
        (∃ch ∈ CH_{vi} / CH_{vi} = getLastChild(v_{p1}) [CH_{vi}]) →
            v[c_{vi}] = property[type] ∧
            v[A_{vi}] ∪ generateImplicitVar()∧
            ch[d_{ch}] = v ∧
            ch[n_{ch}] = property[name] ∧
            ch[l_{ch}] = a[l_a] ∧
            ch[u_{ch}] = a[u_{ch}] ∧
        p_v = v

Descripción Formal 28: Procesado de nodos AST de tipo PropertyCallExp.

IteratorExp

Un nodo AST IteratorExp siempre tiene dos hijos, etiquetados como *source* y *body*. Una vez procesados, el hijo *source* habrá generado como PIT una cadena de instancias (que denominaremos s*ource instance chain*). El tipo de PIT que se genera tras procesar el hijo *body* depende del tipo del nodo AST, pero independientemente del PIT creado, todos sus nodos de instancias serán de las mismas clases que los presentes en *source instance chain.*

Si el hijo en *body* es un nodo AST del tipo OperationCallExp, el PIT tendrá un nodo raíz vacío, y una serie de hijos. Cuando se da este caso, la raíz del PIT de *source* es comparado con cada uno de los hijos inmediatos del PIT de *body*, y

cuando encuentra uno de la misma clase, el nodo de *body* es unido al nodo de. Este proceso se repite con cada uno de los hijos en *source instance chain*, hasta que todos los nodos de *body* han sido unidos. Todos y cada uno de los nodos de *body* terminarán siendo unidos, ya que todas las clases de *body* existen siempre en *source instance chain*.

En cambio, si el hijo en *body* es un nodo AST de cualquier otro tipo, siempre generará como PIT una cadena de instancias. En ese caso la raíz del PIT de *body instance node* se concatenará como hijo del último hijo de *source instance chain*.

processNode(IteratorExp) $\rightarrow p_v$:
s_v = getChildInstanceNode[source],
if body[type] $\in$ 0perationCallExp $\rightarrow$
B_{vb} = getChildInstanceNode[body],
$\forall v_s \in$ collectAllInstanceChainChildren(s_v),
$\forall b_v \in B_{vb}, \forall v_b \in$ collectAllDirectChildren(b_v),
$v_s[c_{vi}] = v_b[c_{vi}] \rightarrow v_s$ = merge(v_s, v_b) ^
$p_v = s_v$
else $\rightarrow$
$s_{lc} = $ getLastChild(s_{vs}),
b_v = getChildInstanceNodes[body],
$s_{lc} = $ merge(s_{lc}, $_{bv}$),
$p_v = s_v$

Descripción Formal 29: Procesado de nodos AST de tipo IteratorExp.

OperationCallExp

Finalmente, un nodo AST OperationCallExp tiene un hijo *source* (que siempre genera como PIT una cadena de instancias), un hijo *operation*, y un número indeterminado de hijos *argument*. En la mayoría de casos, el tipo de nodo AST en *operation* es irrelevante para el método por lo que puede ser ignorado, salvo una excepción. Si el nodo *operation* es del tipo *allInstances*, el proceso se realiza de un modo diferente, al ser allInstances un tipo de operación especial en OCL, como se mencionó en el capítulo 0. Las operaciones *allInstances* no tienen argumentos, por lo que los hijos del AST son solamente *source* y *type*. En esos casos, se crea un nodo de instancias para el PIT con un nombre de variable self y una clase igual a la del contexto, con otro nodo de instancias como hijo que tendrá un nombre de variable generado implícitamente y la misma clase que la presente en el nodo AST en *source*. El nombre de la arista conectando ambos será "*Class:allInstances*", con una cardinalidad mínima de 0 y una máxima *many* (*).

Si el nodo *operation* no es del tipo *AllInstances* es ignorado, y todos los PIT de *source* y los distintos nodos *argument* son unidos entre ellos cuando son del mismo tipo y comparten un nombre de variable.

$$
\begin{aligned}
&\text{processNode(OperationCallExp)} \rightarrow p_v: \\
&\quad \text{if operation[name]} = \text{"allInstances"} \rightarrow \\
&\qquad v \in \text{INSTANCENODE}, \\
&\qquad v_s \in \text{getChildInstanceNode(source)}, \\
&\qquad ch \in v[CH_{vi}], \\
&\qquad v\,[c_{vi}] = \text{context} \wedge \\
&\qquad v[V_{vi}] \cup \text{“self”} \wedge \\
&\qquad ch[d_{ch}] = v_s \wedge \\
&\qquad ch[n_{ch}] = \text{“}v_s[c_{vi}]\text{:allInstances”} \wedge \\
&\qquad ch[l_{ch}] = 0 \wedge \\
&\qquad ch[u_{ch}] = * \wedge \\
&\qquad p_v = v \\
&\quad \text{else} \rightarrow \\
&\qquad v \in \text{INSTANCENODE}, \\
&\qquad \forall V_a \in \text{getChildInstanceNodes}(), \\
&\qquad \forall V_b \in \text{getChildInstanceNodes}(), \\
&\qquad (\\
&\qquad\quad (v_a[c_{vi}] = v_b[c_{vi}]) \wedge \\
&\qquad\quad (\exists \forall var_a \in v_a[V_{vi}], \exists \forall var_b \in v_b[V_{vi}]) \,/\, v_a[var_a] = v_b[var_b]) \\
&\qquad) \rightarrow v_a = \text{merge}(v_a, v_b) \wedge \\
&\qquad V[CH_v] = v_a \wedge p_v = v
\end{aligned}
$$

Descripción Formal 30: Procesado de nodos AST de tipo OperationCallExp.

Tipos de nodos restantes

El resto de nodos AST simplemente se limitan a unir los PIT generados por sus hijos. Este algoritmo siempre termina, dado que el árbol AST es finito y el recorrido del mismo se realiza una única vez en post-orden. Cada vez que un nodo es procesado, el resultado es siempre un único PIT, por lo que al procesar finalmente el nodo raíz, todos los PIT generados se unen en un único nodo raíz, que será la raíz del árbol de instancias final resultante, el cuál siempre será de la misma clase que el contexto, y tendrá como nombre de variable *self*.

Como se mostrará en el siguiente capítulo, el proceso de recorrer un árbol AST es fácilmente implementable en cualquier lenguaje orientado a objetos utilizando el patrón de diseño *Visitor* [64] [65], que permite ejecutar diferente lógica en función del tipo de nodo AST recibido durante el recorrido.

12.7 Definición de métricas

Disponiendo de la definición formal de los árboles de instancia es sencillo realizar operaciones sobre ellas y describir formalmente como obtener las métricas descritas en el capítulo 0. Disponer de estas métricas descritas formalmente facilitará a su vez la descripción formal de las clasificaciones.

Por un lado, es necesario formalizar las métricas que especifican el número mínimo de instancias necesarias para evaluar una restricción, así como el número máximo de ellas que pueden llegar a intervenir.

Las funciones *minI*(v_i) y *maxI*(v_i) reciben un elemento INSTANCENODE, $v_{it} \in IT$, y calculan la métrica desde ese nodo raíz, recorriendo a su vez sus hijos de forma recursiva. El resultado obtenido para cada hijo es multiplicado por la cardinalidad (la cardinalidad mínima en la función *minI* y la máxima en la función *maxI*) de la arista que lo conecta con su nodo padre. Pasando la raíz del árbol de instancias a estas funciones, la función se llama sobre todos sus hijos hasta obtener el resultado final.

$$\text{minI}(v_i) = |v_i[V_{vi}]| + \Sigma_{i=0 \text{ to } n}\, ch_{vit}[l_{ch}] \cdot \text{minI}\,(ch_{vit}[d_{ch}]),\ \forall ch_{vi} \in v_i[CH_{vi}]$$
$$\text{maxI}(v_i) = |v_i[V_{vi}]| + \Sigma_{i=0 \text{ to } n}\, ch_{vit}[u_{ch}] \cdot \text{maxI}\,(ch_{vit}[d_{ch}]),\ \forall ch_{vi} \in v_i[CH_{vi}]$$

Descripción Formal 31: Funciones maxI(v_i) y minI(v_i) para el cálculo del número de instancias.

Otra métrica importante es el número de clases distintas referenciadas en la restricción. La función *c*(v_i) devolverá un conjunto con todas las clases, visitando recursivamente todos los nodos de instancias del árbol y comprobando a qué clase c_{vi} pertenecen.

$$c(v_i) = v_i[c_{vi}] \cup (\cup_{i=0 \text{ to } n}\, c(d_{chi})),\ \forall d_{ch} \in ch_{vi} \in v_i[CH_{vi}]$$

Descripción Formal 32: Función c(v_i) para la creación del conjunto de clases.

Finalmente, también es importante conocer el número de atributos involucrados en la restricción. La función *a*(v_i) obtiene todos los atributos que se mencionan visitándolos recursivamente todos los nodos y uniendo los conjuntos A_{vi} de cada uno.

$$a(v_i) = v_i[A_{vi}] \cup (\cup_{i=0 \text{ to } n}\, a(d_{chi})),\ \forall d_{ch} \in ch_{vi} \in v_i[CH_{vi}]$$

Descripción Formal 33: Función a(v_i) para la creación del conjunto de atributos.

Es importante destacar que, aunque *minI*(v_i) y *maxI*(v_i) devuelven un número entero como valor, *c*(v_i) and *a*(v_i) devuelven conjuntos de elementos. No obstante, es posible obtener el número de elementos en los mismos cuando es necesario cuantificar la cantidad de clases o atributos involucrados.

12.8 Clasificación de restricciones

La clasificación de las restricciones se realiza en base a las métricas que se obtienen de los árboles de instancias. Las restricciones pueden ser clasificadas por su tipo, o por su nivel de dependencia con el servidor.

Las restricciones de tipo previamente mencionadas informalmente pueden ser formalizadas del siguiente modo.

Restricciones de atributo: Tanto el número mínimo como el máximo de instancias debe ser uno, y el tamaño del conjunto de atributos debe también ser

igual a uno. Verificar el número de clases no es necesario dado que basta con comprobar que haya una única instancia.

$$\text{Attribute: } (\text{minI}(\text{IT}[v_i] = 1) \wedge (\text{maxI}(\text{IT}[v_i] = 1) \wedge (|\text{a}(\text{IT}[v_i])| = 1)$$

Descripción Formal 34: Restricciones de atributo.

Restricciones de objeto: El número mínimo y máximo de instancias debe ser uno, pero el tamaño del conjunto de atributos debe ser mayor que uno.

$$\text{Object: } (\text{minI}(\text{IT}[v_i] = 1) \wedge (\text{maxI}(\text{IT}[v_i] = 1) \wedge (|\text{a}(\text{IT}[v_i])| > 1)$$

Descripción Formal 35: Restricciones de objeto.

Restricciones de clase: El número de instancias máximas debe ser mayor que uno, pero el tamaño del conjunto de clases debe ser uno.

$$\text{Class: } (\text{maxI}(\text{IT}[v_i] > 1) \wedge (|\text{c}(\text{IT}[v_i])| = 1)$$

Descripción Formal 36: Restricciones de clase.

Restricciones de dominio: Basta con verificar que el tamaño del conjunto de clases es mayor que uno.

$$\text{Domain: } |\text{c}(\text{IT}[v_i])| > 1)$$

Descripción Formal 37: Restricciones de dominio.

La clasificación en función del nivel de dependencia con el servidor puede ser formalizada del siguiente modo:

Independiente del servidor: La restricción tiene un número mínimo y máximo de instancias igual a uno, y la clase que referencia pertenece al CDM.

$$\text{ServerIndependent: } (\text{minI}(\text{IT}[v_i]) = 1) \wedge (\text{maxI}(\text{IT}[v_i]) = 1) \wedge (\forall c \in \text{c}(\text{IT}[v_i]), c \in V_{cd})$$

Descripción Formal 38: Restricciones independientes del servidor.

Potencialmente dependiente del servidor: La restricción tiene un número máximo de instancias mayor que uno, y todas las clases que referencia pertenecen al CDM.

$$\text{PotentiallyServerDependent: } (\text{minI}(\text{IT}[v_i]) > 1) \wedge (\forall c \in \text{c}(\text{IT}[v_i]), c \in V_{cd})$$

Descripción Formal 39: Restricciones potencialmente dependientes del servidor.

Dependiente del servidor: En el conjunto de clases de la restricción existen clases que no pertenecen al CDM.

$$\text{ServerDependent: } \exists i \in \text{c}(\text{IT}[v_i]) \;/\; i \notin V_{cd}'$$

Descripción Formal 40: Restricciones dependientes del servidor.

Disponer de una descripción formal facilita la implementación de este método utilizando cualquier tipo de tecnología, como se mostrará en el próximo capítulo.

13 IMPLEMENTACIÓN DEL MÉTODO EN UN PROTOTIPO

El método presentado formalmente muestra que es posible generar un modelo para el cliente y clasificar sus restricciones de forma automatizada. Aunque es posible definir todo el proceso de forma no ambigua mediante el uso de teoría de conjuntos y grafos, también es importante demostrar que se trata de un método práctico y aplicable mediante el uso de herramientas existentes en la actualidad.

En este capítulo se describirá la implementación del método formal en un prototipo utilizando herramientas convencionales de desarrollo de software.

13.1 Objetivos

El objetivo del prototipo a desarrollar será demostrar la viabilidad del desarrollo de herramientas basadas en este método. El prototipo se limitará a una prueba de concepto que implemente todas las características del método presentado formalmente.

- Debe ser capaz de llevar a cabo todas las fases descritas en el método.
- Debe respetar las definiciones formales que se han descrito para el método.
- Debe demostrar que es aplicable sobre herramientas de modelado ya existentes.
- La herramienta no debería condicionar los estándares habituales de modelado y desarrollo de software, sino complementarlos.

Siendo una herramienta de asistencia al diseño, se beneficiaría enormemente de estar integrada con interfaces gráficas. No obstante, tales

funcionalidades, aunque útiles, no contribuyen a demostrar la viabilidad del método. No entra dentro de los objetivos de este prototipo el desarrollo de una herramienta gráfica usable y profesional, atractiva para la labor diaria de diseño de software. El prototipo se limitará a una API que reciba como datos de entrada ficheros con los modelos en alguno de los estándares utilizados en la industria, y devuelva a su vez ficheros del mismo tipo con los resultados. Demostrada la viabilidad técnica del modelo, el desarrollo de una herramienta profesional basada en interfaces gráficos queda fuera del alcance de este trabajo.

13.2 Herramientas utilizadas

Eclipse Modeling Framework [44] es uno de los *frameworks* más populares y soportados en la ingeniería dirigida por modelos. Dentro del entorno de desarrollo de Eclipse, EMF es la base sobre la que se desarrollan la mayoría de herramientas basadas en modelos.

El objetivo de este *framework* es permitir al desarrollador definir el modelo de forma independiente, y a partir de él generar una variedad de artefactos relacionados con él durante el desarrollo del software. Es posible utilizarlo para para realizar manipulaciones sobre el modelo, realizar validaciones, generar código de forma automatizada o definir la persistencia del modelo.

Dresden OCL [63] es un conjunto de herramientas desarrolladas como plugins para la plataforma Eclipse que permiten analizar, procesar sintácticamente e interpretar código OCL, así como generar código Java y SQL a partir de él.

El desarrollo del prototipo se basará en estas dos herramientas. Por un lado, se tratan de estándares de facto ampliamente utilizados en la industria. Estas herramientas disponen de funcionalidades muy amplias y diversas, pero para el desarrollo del prototipo solo serán necesarias aquellas relacionadas con la creación y de modelos UML y código OCL. No obstante, un prototipo compatible con estas herramientas implica que el resto de funcionalidades adicionales que ofrecen, como la generación de código, podrán ser complementadas con el prototipo presentado.

13.3 Descripción del prototipo

El prototipo está desarrollado en Java, apoyándose en las herramientas proporcionadas por los plugins de EMF y Dresden Tools.

La herramienta ha sido desarrollada como una API programática. El programador puede importar el paquete en Java, y utilizar los métodos que la herramienta ofrece. Aunque la API ofrece numerosos métodos para realizar consultas sobre el modelo y las restricciones, para poner en marcha el proceso basta con instanciar la API y hacer una única llamada a un método.

```
IConstraintsAnalyzer constraintsAnalyzer =
toolFactory.createConstraintAnalyzer(ecoreFileName, oclFileName);

constraintsAnalyzer.createSubsetModel(classesList);
```

Ejemplo de código 32: Código básico para el lanzamiento del proceso.

Por un lado, la API se instancia con los ficheros de entrada del modelo, es decir el fichero con modelo de clases del SDM y el fichero que detalla sus restricciones OCL. Con la API instanciada, es posible llamar al método que lanza el proceso, el cual recibe como parámetro una lista con los nombres de las clases pertenecientes al SDM que deben formar parte del CDM. Una vez ejecutado el método, generará una serie de ficheros de salida con los resultados finales.

Ficheros de entrada

El prototipo procesa los modelos de clase en formato *Ecore*. Estos modelos pueden ser creados gráficamente utilizando plugins basados en el EMF como por ejemplo *EcoreTools* [66].

Las restricciones se definen en un fichero *.ocl*. Simplemente contiene el código de cada restricción. El plugin de *Dresden Tools* permite disponer de un editor que proporciona sintaxis OCL resaltada y detecta errores sintácticos en las mismas. No obstante, cualquier editor de texto es válido para crear este fichero.

Finalmente, las clases del SDM que deben utilizarse para la creación del CDM se proporcionan como una lista de cadenas de texto, cada una con el nombre de una de las clases. Esta lista puede proporcionarse como un fichero con un nombre de clase en cada línea, o directamente como una lista de elementos String a través del código.

Fases 1: Extracción de restricciones implícitas

La herramienta pasa por las cuatro fases descritas en nuestro método, de las cuales la primera es analizar el modelo UML para extraer las restricciones implícitas.

Dentro de la API del prototipo existe un módulo destinado a facilitar la consulta sobre el modelo de clases, la cual permite extraer información acerca de los elementos y estructura del modelo de clases. Por ejemplo, permite consultar qué relaciones tiene una determinada clase, listar todas las relaciones de herencia existentes en el modelo, u obtener un listado con las relaciones que cumplen unas determinadas características.

Los métodos de esta API son utilizados para obtener la información de cada relación existente, y generar una relación explícita adecuada en función de sus características. Antes de este procesamiento, el fichero *.ocl* proporcionado originalmente es copiado y renombrado, para posteriormente concatenar sobre él las nuevas restricciones. Éstas son generadas utilizando la información

extraída del modelo y siguiendo las reglas descritas en los capítulos 8.1 y 12.5. Por cada relación se evalúa si es necesario generar una restricción, y si es así la cadena con el código OCL es concatenada al final del nuevo fichero.

Fase 2: Simplificación de las restricciones

Una vez este fichero, que contiene tanto las restricciones originales como las nuevamente generadas, se pasa a la segunda fase del método, para lo cual se crea un nuevo fichero *.ocl* vacío. Cada una de las restricciones existentes es procesada y se aplican sobre ella el proceso de simplificación descrito en el capítulo 8.2. Todas las restricciones, hayan requerido simplificación o no, son añadidas a este nuevo fichero *.ocl*, que será el que el prototipo utilizará en los próximos pasos del modelo.

Aunque sería posible limitarse a modificar el código del fichero *.ocl* proporcionado originalmente, se ha optado por generar varias iteraciones con el fin de proporcionar al diseñador la posibilidad de consultar la evolución del proceso. A efectos prácticos, el único fichero relevante sería el último generado.

Fase 3: Generación del CDM

La tercera fase consistente en la generación del CDM, y para ello inicialmente se crea un nuevo modelo que contiene únicamente las clases mencionadas en la lista seleccionada por el usuario, y las relaciones entre ellas.

Todas estas clases son evaluadas en el modelo original, y se comprueba si alguna de las relaciones contenidas en las mismas cumple las condiciones descritas en los capítulos 9.3 y 12.4. Si es así, las clases y relaciones pertinentes son copiadas al nuevo modelo.

Si durante esta evaluación se añade alguna clase nueva al modelo, el proceso se repite de nuevo, hasta que se llegue a una comprobación en la que no se añada ningún elemento nuevo. El modelo resultante será el CDM, y se guardará como un nuevo fichero *.ecore*. Este fichero puede ser consultado por el diseñador y visualizado gráficamente mediante herramientas como *EcoreTools*.

La implementación de esta fase se realiza haciendo uso de uno de los módulos de la API del prototipo destinada a la creación y modificación de los modelos de clases de forma programática.

Fase 4: Análisis de las restricciones, extracción de métricas y clasificación

Finalmente, la cuarta fase se encarga del procesamiento de las restricciones simplificadas, y tiene a su vez varias etapas.

— Generación de árboles de instancias.

— Extracción de métricas.

— Clasificación de restricciones.

Las restricciones son procesadas mediante *Dresden Tools*, que facilitan la creación de su árbol AST, así como recorrerlo. Mediante la implementación del patrón Visitor [64], ese árbol AST puede ser recorrido para crear el árbol de instancias. El patrón Visitor permite crear métodos distintos para cada tipo de nodo. Mientras el árbol AST está siendo recorrido, en función del tipo de nodo AST que se recorra, se ejecutará el método apropiado para procesarlo. Las reglas implementadas para cada tipo de nodo son las descritas en los capítulos 10.4 y 12.6, y mediante ellas a medida que se recorre el árbol AST se genera iterativamente el árbol de instancias. Esta estructura de datos es guardada para futuras referencias.

Adicionalmente, para generar documentación que facilite el trabajo del diseñador, se generan ficheros de tipo *PDF* para cada restricción con la representación visual tanto del árbol AST original como del árbol de instancias. Para la elaboración de estos ficheros se utiliza la herramienta Graphviz [67], que permite crear representaciones gráficas basándose en código escrito en un fichero *.dot*. La creación de estos ficheros se realiza también mediante la implementación del patrón Visitor. Según se van recorriendo los nodos de los árboles, se añade al fichero *.dot* la información correspondiente para su representación gráfica. Finalmente, a partir de los ficheros *.dot*, Graphviz genera ficheros *PDF* que muestran visualmente los árboles. Durante la cuarta fase, en una segunda etapa se realiza la extracción de métricas una vez se ha creado para cada restricción un árbol de instancias. La API dispone de un módulo que permite consultar las métricas de cada uno de estos árboles. Con las métricas proporcionadas por estos métodos, y la información disponible en el modelo creado para el CDM, se pasa a la tercera etapa donde es posible realizar la clasificación de cada restricción tal y como se describe en los capítulos 11.2, 11.3, 12.8.

Por un lado, se crea un fichero de texto plano al que se añadirá una lista con desglosando para cada restricción las métricas obtenidas, así como la clasificación tanto por tipo como por nivel de dependencia respecto al CDM.

Además de esto, se crea un nuevo fichero *.ocl*, que incluirá única y exclusivamente aquellas restricciones aplicables al CDM de forma totalmente independiente.

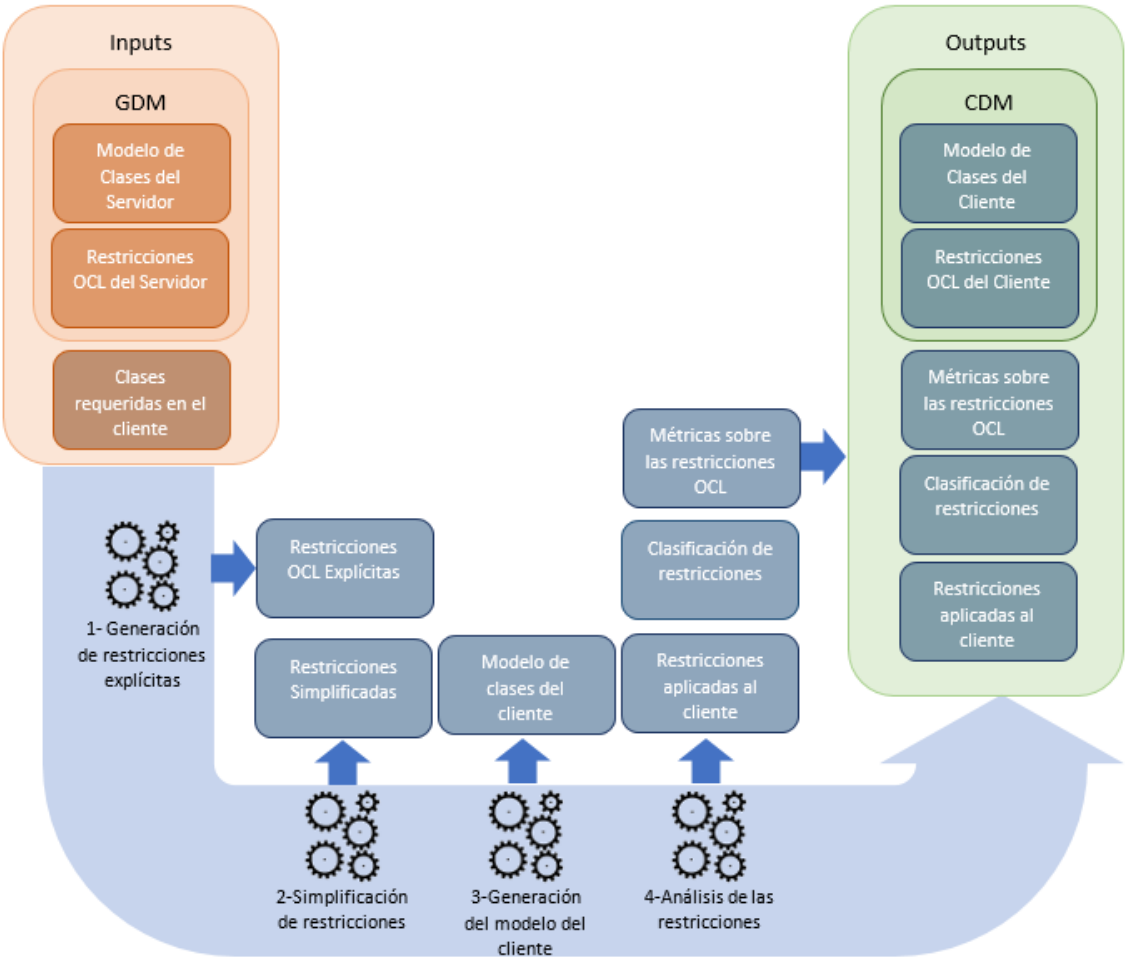


Fig 63: Diagrama describiendo el proceso completo.

Lista de ficheros de salida

Una vez todas las fases del método finalizan, los ficheros que se producen son los siguientes:

- Modelo de clases del CDM en formato *.ecore*
- Ficheros OCL:
  - Restricciones originales + restricciones implícitas en formato *.ocl*.
  - Todas las restricciones simplificadas en formato *.ocl*.
  - Restricciones independientes para el modelo de clases del CDM.
- Documentación adicional:
  - Fichero de texto con métricas y clasificación de cada restricción.
  - Un fichero *PDF* por cada restricción con su árbol AST.
  - Un fichero *PDF* por cada restricción con su árbol de instancias.

Para poner en marcha el proceso al completo basta ejecutar un único método que recibe los parámetros de entrada. Dado que se generan una variedad de ficheros como resultado, el método está sobrecargado de forma que el desarrollador pueda elegir las rutas y nombres de estos ficheros. Si no se especifican la herramienta genera estos ficheros eligiendo rutas y nombres por defecto basados en los nombres de los modelos y las restricciones.

No obstante, es posible acceder a los distintos módulos de la API y ejecutar cada fase del método por separado, o utilizar métodos de consulta y modificación programáticamente y con independencia del método presentado. Gracias a esto la API puede extenderse fácilmente para añadir nuevas funcionalidades si fueran necesarias.

13.4 Caso de ejemplo utilizando el prototipo

A continuación, se mostrará cómo utilizar el prototipo para procesar el caso de ejemplo. De este modo se pueden observar claramente qué tareas debe realizar el diseñador, y cuáles lleva a cabo la herramienta automáticamente.

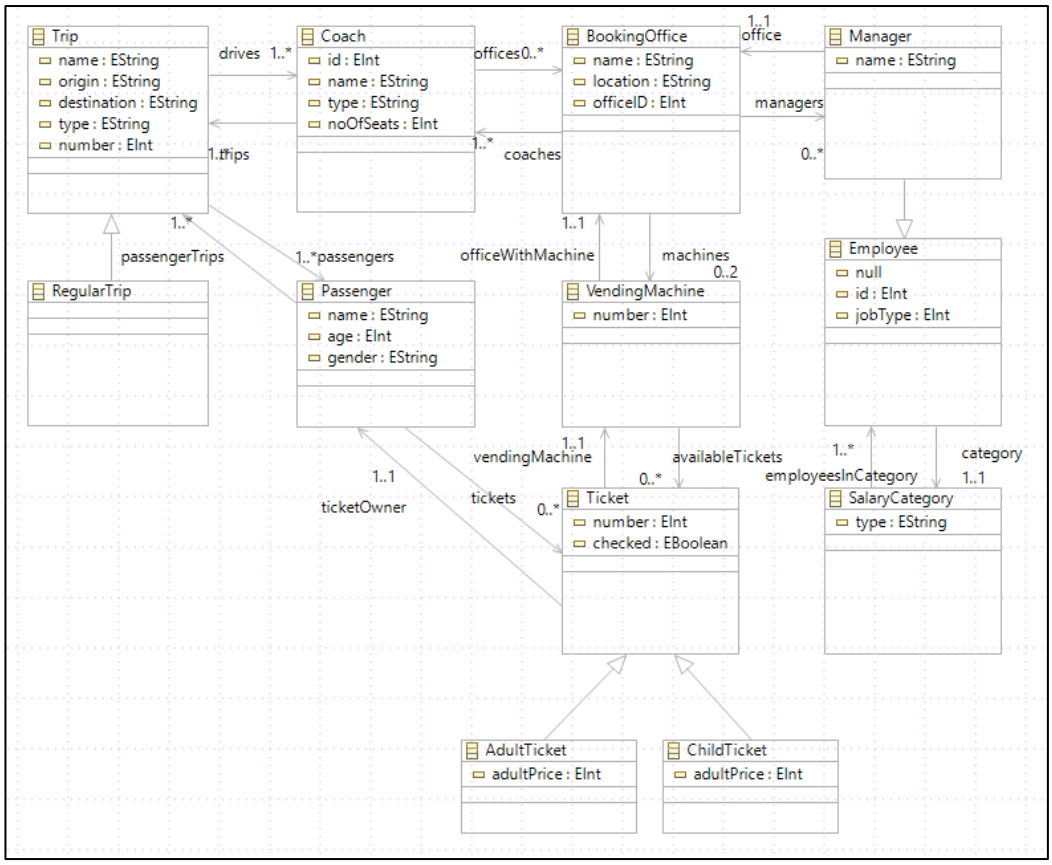


Fig 64 Modelo de clases para el SDM

Antes de utilizar el prototipo, el diseñador deberá haber creado el modelo para el servidor. Esta tarea puede llevarse a cabo de forma gráfica mediante el uso de *EcoreTools*. Este modelo queda guardado como un fichero *.ecore*.

Además, deberá definir cuáles son las restricciones, en un fichero *.ocl*.

```
context Passenger inv ocl1_nonNegativeAge:
    self.age > 0

context Coach inv ocl2_typeOfCoach:
    self.type = 'big' implies self.noOfSeats>30

context Ticket inv ocl3_ticketNumberPositive_alternative:
    Ticket.allInstances()->forAll(a, b | a<>b implies a.number<>b.number)

context ChildTicket inv ocl4_correctAgeChild:
    self.ticketOwner.age < 12

context Passenger inv ocl5_correctAgeChild2:
    self.age<12 implies self.tickets->forAll(a|a.oclIsKindOf(ChildTicket))

context Trip inv ocl6_enoughSeats:
    self.passengers->size() <= self.drives.noOfSeats->sum()

context BookingOffice inv ocl7_enoughTickets:
    self.coaches.noOfSeats->sum() >= self.machines.availableTickets->size()
```

Ejemplo de código 33: Restricciones OCL para el SDM.

Estas son las dos tareas relacionadas con el diseño que se deben realizar. La única tarea adicional será poner en marcha el prototipo, indicándole dónde encontrar estos dos ficheros, y proporcionándole la lista de clases que deben formar parte del CDM. Esta tarea se realiza programáticamente del siguiente modo.

```
private static String ecoreFileName = "modelfile/SDM.ecore";
private static String oclFileName = "modelfile/SDM_constraints.ocl";
private static String[] classesList = new String[] { "RegularTrip", "Coach",
"Passenger","AdultTicket", "ChildTicket" };

IConstraintsAnalyzer constraintsAnalyzer =
toolFactory.createConstraintAnalyzer(ecoreFileName, oclFileName);

constraintsAnalyzer.createSubsetModel(classesList);
```

Ejemplo de código 34: Lanzamiento del proceso.

De este modo, *createSubsetModel* generará automáticamente todos los ficheros una vez ejecutado. Opcionalmente, *createSubsetModel* está sobrecargado ofreciendo diversas opciones para definir los directorios los directorios donde dejar los ficheros creados.

```
private static String generatedEcoreFilePath = "modelfile/submodel";
private static String generatedOclFilePath = "modelfile/submodel";
private static String generatedConstraintsClassificationFilePath =
"modelfile/submodel/constraintdocs";

constraintsAnalyzer.createSubsetModel(classesList, generatedEcoreFilePath,
generatedOclFilePath, generatedConstraintsClassificationFilePath);
```

Ejemplo de código 35: Lanzamiento del proceso pasando rutas específicas.

El CDM resultante podrá ser consultado gráficamente mediante *EcoreTools*, así como el fichero OCL que contiene las restricciones que se pueden aplicar directamente a este modelo.

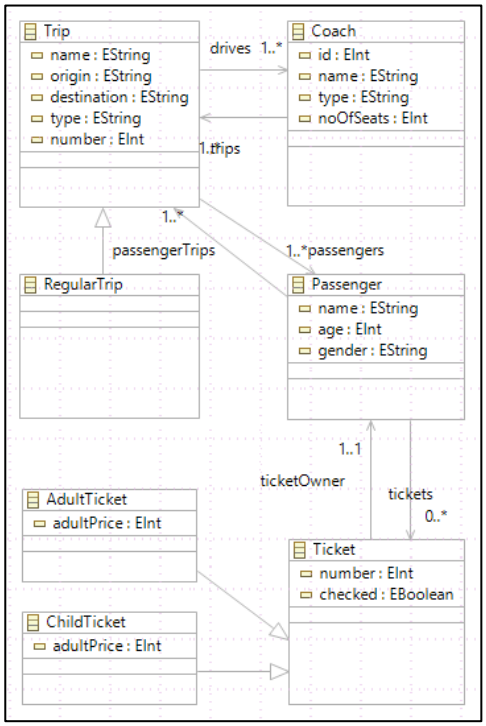


Fig 65 CDM generado automáticamente por el prototipo.

```
context Passenger inv ocl1_nonNegativeAge:
    self.age > 0

context Coach inv ocl2_typeOfCoach:
    self.type = 'big' implies self.noOfSeats>30

context Ticket inv ocl3_ticketNumberPositive_alternative:
    Ticket.allInstances()->forAll(a, b | a<>b implies a.number<>b.number)

context ChildTicket inv ocl4_correctAgeChild:
    self.ticketOwner.age < 12

context Passenger inv ocl5_correctAgeChild2:
    self.age<12 implies self.tickets->forAll(a|a.oclIsKindOf(ChildTicket))

context Trip inv ocl6_enoughSeats:
    self.passengers->size() <= self.drives.noOfSeats->sum()

context BookingOffice inv ocl7_enoughTickets:
    self.coaches.noOfSeats->sum() >= self.machines.availableTickets->size()

/*Explicit constraints*/
context Trip inv generatedImplicitReference_drives:
    self.drives->size() >= 1

context Trip inv generatedImplicitReference_passengers:
    self.passengers->size() >= 1

context Coach inv generatedImplicitReference_trips:
    self.trips->size() >= 1

context Passenger inv generatedImplicitReference_passengerTrips:
    self.passengerTrips->size() >= 1

context BookingOffice inv generatedImplicitReference_machines:
    self.machines->size() <= 2
```

Ejemplo de código 36: Fichero OCL generado con las restricciones implícitas añadidas a las originales.

Se generarán además tres ficheros OCL, uno que añade las restricciones implícitas a las originales, otro que las simplifica todas, y finalmente uno para el CDM que incluye únicamente aquellas clasificadas como independientes.

El fichero OCL para el CDM incluye además comentadas las restricciones clasificadas como potencialmente dependientes. De este modo el diseñador puede tenerlas como referencia y utilizar aquella que estime apropiadas para el cliente.

```
context Passenger inv ocl1_nonNegativeAge:
    self.age > 0

context Coach inv ocl2_typeOfCoach:
    self.type <> 'big' or self.noOfSeats>30

context Ticket inv ocl3_ticketNumberPositive_alternative:
    Ticket.allInstances()->forAll(b | self=b or (self.number<>b.number))

context ChildTicket inv ocl4_correctAgeChild:
    self.ticketOwner.age < 12

context Passenger inv ocl5_correctAgeChild2:
    self.age>=12 or self.tickets->forAll(a|a.oclIsKindOf(ChildTicket))

context Trip inv ocl6_enoughSeats:
    self.passengers->size() <= self.drives.noOfSeats->sum()

context BookingOffice inv ocl7_enoughTickets:
    self.coaches.noOfSeats->sum() >= self.machines.availableTickets->size()


/*Explicit constraints*/
context Trip inv generatedImplicitReference_drives:
    self.drives->size() >= 1

context Trip inv generatedImplicitReference_passengers:
    self.passengers->size() >= 1

context Coach inv generatedImplicitReference_trips:
    self.trips->size() >= 1

context Passenger inv generatedImplicitReference_passengerTrips:
    self.passengerTrips->size() >= 1

context BookingOffice inv generatedImplicitReference_machines:
    self.machines->size() <= 2
```

Ejemplo de código 37: Fichero OCL con las restricciones implícitas después del proceso de simplificación.

```
/*Independent*/

context Passenger inv ocl1_nonNegativeAge:
self.age > 0

context Coach inv ocl2_typeOfCoach:
self.type <> 'big' or self.noOfSeats>30

/** Potentially Dependent

context Ticket inv ocl3_ticketNumberPositive_alternative:
Ticket.allInstances()->forAll(b | self=b or (self.number<>b.number))

context ChildTicket inv ocl4_correctAgeChild:
self.ticketOwner.age < 12

context Passenger inv ocl5_correctAgeChild2:
self.age>=12 or self.tickets->forAll(a|a.oclIsKindOf(ChildTicket))

context Trip inv ocl6_enoughSeats:
self.passengers->size() <= self.drives.noOfSeats->sum()

*/
```

Ejemplo de código 38: Fichero OCL para el CDM. Incluye únicamente las restricciones clasificadas como independientes. Las potencialmente independientes se incluyen comentadas.

También se genera un fichero de texto plano que incluye las métricas de cada restricción y la clasificación, tanto respecto al tipo como al nivel de dependencia con el servidor.

```
------------------------------------------------
context Passenger inv ocl1_nonNegativeAge:
self.age > 0
------------------------------------------------
    Min instances: 1
    Max instances: 1
    Attributes: [Passenger.age:Integer]
    Classes: [Passenger]
    Type: Attribute
    Dependency level: Independent


------------------------------------------------
context Coach inv ocl2_typeOfCoach:
self.type <> 'big' or self.noOfSeats>30
------------------------------------------------
    Min instances: 1
    Max instances: 1
    Attributes: [Coach.type:String, Coach.noOfSeats:Integer]
    Classes: [Coach]
    Type: Object
    Dependency level: Independent

------------------------------------------------
context Ticket inv ocl3_ticketNumberPositive_alternative:
Ticket.allInstances()->forAll(b | self=b or (self.number<>b.number))
------------------------------------------------
    Min instances: 1
    Max instances: *
    Attributes: [Ticket.number:Integer]
    Classes: [Ticket]
    Type: Class
    Dependency level: Potentially-Dependent

------------------------------------------------
context ChildTicket inv ocl4_correctAgeChild:
self.ticketOwner.age < 12
------------------------------------------------
    Min instances: 2
    Max instances: 2
    Attributes: [Passenger.age:Integer]
    Classes: [ChildTicket, Passenger]
    Type: Domain
    Dependency level: Potentially-Dependent

------------------------------------------------
```

Ejemplo de código 39: Fichero de texto con las métricas de cada restricción y su clasificación.

Durante el proceso, también se generarán ficheros *.dot* de *Graphviz* para crear los árboles de instancia y AST de cada restricción.

```
digraph instanceModel_ocl_2016_enoughSeats{

1873[ label="Instance\nvariable = \"self\"\ntype = \"Trip\" "]
1874[ label="Instance\nvariable = \"a\"\nattribute = \"noOfSeats:Integer\"\ntype = \"Coach\" "]
1877[ label="Instance\nvariable = \"b\"\ntype = \"Passenger\" "]
1873->1874 [ label="drives (1, *)" ]
1873->1877 [ label="passengers (1, *)" ]

}
```

Ejemplo de código 40: Fichero .dot de la restricción enoughSeats.

Los ficheros *.dot* consisten en texto describiendo el grafo que posteriormente *Graphviz* transformará en un fichero PDF con su representación visual.

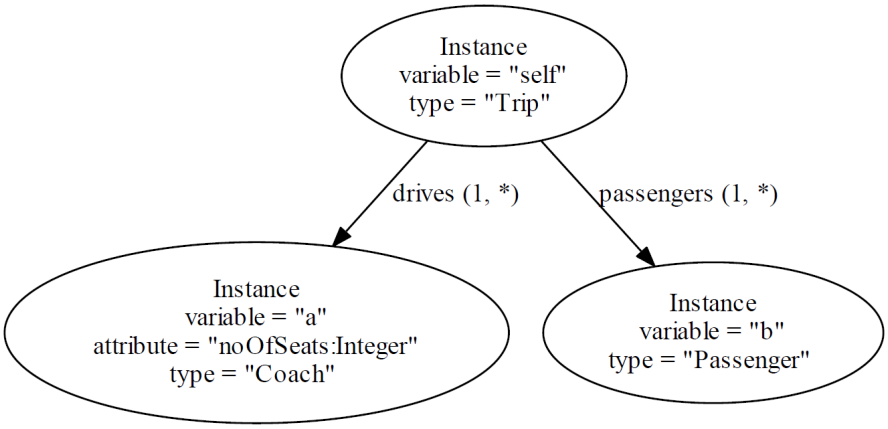


Fig 66 Representación visual en PDF generada por Graphviz a partir del fichero .dot de la restricción enoughSeats mostrada en el Ejemplo de código 40:.

Todos estos ficheros simplifican la tarea del diseñador, ya sea por tratarse de artefactos utilizables en el proceso de desarrollo, o como documentación de consulta y análisis que ayudará a tomar decisiones sobre las restricciones.

14 CONCLUSIONS, LIMITATIONS AND FUTURE WORK

The proposed solution supports the automatic generation of client domain models that ensure the consistency with an already existing server domain model, preventing the developer from facing the challenges described in Chapter 2.

The only task required is to determine the subset of the classes of the server model that are relevant to the client. The solution is able to generate a client class model, and identify which of the server's constraints can be verified locally from the client, and which ones cannot.

Besides this verifiable constraints, the proposed model identifies those that could be checked but may require gathering some information from the server upon request. However, the information provided by the class model and the constraints themselves is not enough to automatically assess if these constraints will require this type of communication or not. The method also describes how to automatically generate documentation about the structure of the constraints based on both objective metrics and visual representation. That eases the understanding of how this restrictions affect the model, and supports the decision of where to validate and manage them.

The proposed solution is technology independent, provided in a formal way that permits its implementation on any existing platform. It can also be generalized to other alternative modeling standards based on visual diagrams and predicate constraints. The formal description has been used to successfully

implement a prototype for the *Eclipse Modeling Framework*, one of the most mature and used platforms in the industry.

14.1 Achieved Objectives

Chapter 3 enumerates the proposed objectives for this work.

— The CDM creation should be based on the available information from the GDM and as automatable as possible.

— The stages of the process that are not automatable should be supported by the generation of documentation that classifies the constraints.

— The proposal should be implementable on existing platforms or technologies, without forcing a specific one

We consider all of them have been addressed for the following reasons.

The CDM creation should be based on the available information from the GDM and as automatable as possible

The proposal describes how to use the existing information from the server domain model to generate the client model. The rules that must be followed during this process are thoroughly described and generate a consistent client domain model that is guaranteed to be consistent with both the designer initial selection of classes and the server domain model.

Through the analysis of the constraints, the extraction of their metrics and their classification, both the independent and dependent constraints are successfully identified. The only aspect of the process that requires manual intervention is related to the restrictions that are potentially dependent on the server, and are addressed on the following objective.

The stages of the process that are not automatable should be supported by the generation of documentation that classifies the constraints

The proposed classification shows how the constraints classified as *potentially-dependent* can be evaluated on the client. It is not possible, however, to infer automatically for each of this constraints if they will require communication with the server for their verification or not. The information provided by the class model and the restrictions is not enough to decide on how to manage these constraints, since a further understanding of the domain model is required.

However, all the relevant information that can be assessed from the model and constraints is automatically generated. Chapters 0 and 1 have explained how to automatically generate a visual representation of the constraints in the form of instance trees, and how this data structure can be navigated to quantify important aspects on how the restriction is evaluated, such as the number of instances, classes or attributes involved. The process of obtaining this

information is described formally and provides objective data about the complexity of a constraint.

These metrics are also used to classify the constraints, providing the designer information on the variety of elements affected by a constraint and its level of dependency. The combination of these three elements (instance trees, metrics and classifications) provides the developer further insight on the complexity of each constraint and how to manage them.

The proposal should be implementable on existing platforms or technologies, without forcing a specific one

The proposal is described exclusively over the UML standards without forcing any specific technology or tool, and furthermore, its formalization, although based on the UML standards, can be easily applied over similar alternatives if needed.

If a specific development methodology is required, this method can be integrated seamlessly without forcing the process in any particular direction. Developers who choose a MDE approach can use the generated models during the process. Methodologies where changes in the model are frequent (iterative and incremental methods) and encourage frequent prototyping and testing can be also benefited by this method, since each time the server domain model changes there is no need to manually adapt the client, but generated again from scratch ensuring that human errors are avoided.

To avoid forcing the implementation of the proposal using a specific technology, we provide a formal description based on graph theory, which describes appropriately diagram based models such as UML. OCL is a predicate based language, but by formalizing it into instance trees that represent the elements from the class graph affected by it, it becomes compatible with the graph theory representation.

Other modeling alternatives such as entity-relationship and SQL, or domain-driven development diagrams can also be represented through this approach. Modeling the diagrams through graphs, and translating the predicates into trees that represent those graph elements affected by it is possible with little adaptation. The only adaptations required would be the inference rules for graphs and the AST traversal for predicates.

The formal description provided is able to describe unambiguously how to generate all the models. It guarantees that the produced models are consistent with the server at all times. These descriptions have been used to implement a working prototype for one of the most mature and used domain modeling platforms.

14.2 Limitations

The method is focused on OCL invariants, excluding other constraint types such as preconditions or postconditions. We hope to be able to build upon this initial foundation to include other constraint types in the future.

Currently, we focus on the design aspects of the development cycle, and more specifically, on the domain model and its constraints. The whole development cycle of such clients, such as the implementation of the constraints, presents challenges that are not addressed by this work. Working at the level of class diagrams also offers some limitations in certain cases. In modern distributed applications, RESTful web services [68] are one of the main paradigms adopted by the industry. In those cases, the state is usually maintained on the client; and the server acts mainly as an intermediary between client and database. Although there may not be a server domain model, there is still a data model on the database, which can have its own integrity constraints, and can be related to the client domain model. Although our current method is heavily based on the UML standard and may not yet fit this scenario, OCL constraints and UML models can be related to relational databases [69]. It is also important to highlight that the formal approach we undertake makes our method general enough to be adapted to other ways of representing models and restrictions, and adjusting it to entity-relationship models. Database integrity constraints would be an interesting path to follow.

One important aspect of our method is that it delimits the designer's responsibilities, relieving them from the need to model and maintain the CDM while checking compliance with the server model. With the method presented here, the only task required would be to identify the core client-required classes. Nevertheless, this approach has room for improvement. In the slicing process, we define certain rules on how additional elements can be added to the initial selection. This way, the core classes can use any other element that is tightly coupled to them, even if the designer did not select them initially. However, in some cases this approach might not be enough.

In summary, there is still room to maximize the scope of this approach, but the level of analysis to act on such cases is out of the scope of the current proposal. We believe this work is an interesting step towards providing a foundation for easing the development of robust rich clients.

14.3 Future work

In addition to addressing the limitations described in the previous section, this work is open to many possible improvements.

Integration with automatic error recovery

One of the strengths of this proposal is that it highlights the elements affected by a constraint. Our research group has dealt with automatic error recovery techniques in rich clients [11] that require the developer to identify which specific elements of the domain model should be recoverable after an error. These two approaches could be combined, using the information from the constraints to automatically tag the elements of the domain model that are affected by them as automatically recoverable in case a constraint is violated.

```
context Trip inv enoughSeats:
      self.passengers->size() <= self.drives.noOfSeats->sum()
```

Ejemplo de código 41: enoughSeats restriction.

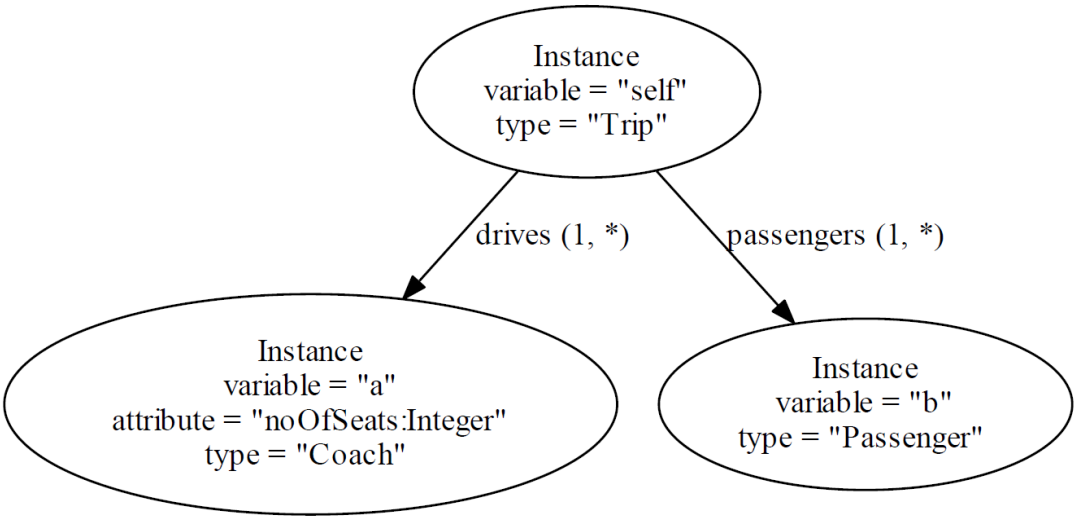


Fig 67 Instance Tree of the enoughSeats constraint.

For example, the *enoughSeats* constraint could be used to identify that the passengers collection should be automatically recoverable. If someone adds to a Trip a passenger that causes the evaluation of the constraint to fail, the collection could be reverted back to its previous state automatically. The same idea can be applied if modifications that causes the constraint to fail are applied to the *drives* collection, or the *noOfSeats* attribute.

Improved efficiency through precalculated values

To address those cases where a restrictions are verifiable on the client, but require communication with the server, it would also be possible to add design elements between client and server that could ensure the information is retrieved to the client in the most efficient way for the evaluation of those specific constraints. This efficiency could be achieved by identifying and retrieving only the essential objects from the server. It could also be achieved by precalculating on the server

the values required for the evaluation of the constraints on the client. That approach would avoid the need to send complete subparts of the object graph. These solutions, however, may be complex and require a thoughtful analysis of the constraints involved. A challenging task that may get even more complicated as the development cycle progresses and changes are applied to the domain model and its constraints.

```
context Trip inv enoughSeats:
      self.passengers->size() <= self.drives.noOfSeats->sum()
```

Ejemplo de código 42: enoughSeats restriction.

Going back to the *enoughSeats* constraint, let us imagine that in order to evaluate it, it is required to either retrieve all the passengers' instances to the client, or delegate it to the server. However, in both cases the computation of these elements as described in the constraint could be performance intensive. The information of the constraint could be used to add information to the domain model and generate a new class of the model on both client and server that holds relevant values for the evaluation of constraints and defines the way to calculate it. In this scenario, it could be an integer that keeps track of the current number of passenger instances. Retrieving this single value could be more efficient that requesting all the instances and counting them when evaluating the constraint.

Further constraint analysis

Currently, our method is focused on the constraint-affected elements, but not the predicates that will be calculated over them. By analyzing the operations involved in those predicates, and fine grained aspects of the domain model further automated decisions could be made.

For instance, the *enoughSeats* constraint tries to avoid that the number of passengers grows higher than the number of seats available.

```
context Trip inv enoughSeats:
      self.passengers->size() <= self.drives.noOfSeats->sum()
```

Ejemplo de código 43: enoughSeats restriction.

However, if a specific client does not have a functionality that allows to add more passengers, or a way to reduce the number of coaches or seats, it would be safe to automatically remove this constraint from the client.

These approach could also be used to modify the constraints that are not evaluable on the client. One possible approach could be breaking them down strategically into several ones that affect fewer elements, so that at least part of its predicate can be checked locally on the client. Achieving this with constraints that involve complex predicate logic would require to build a new level of analysis over the already existing one.

15 RECONOCIMIENTOS

Este trabajo ha sido financiado por la Unión Europea, a través de los Fondos Europeos de Desarrollo Regional (FEDER); y el Principado de Asturias, a través de su Plan de Ciencia, Tecnología e Innovación. Proyecto GRUPIN14-100.

16 BIBLIOGRAFÍA

[1] J. Duhl, "White paper: Rich internet applications," *Avail-able http//wwwmacromedia. comn/ …*, 2003.

[2] A. Bozzon, S. Comai, P. Fraternali, and G. Carughi, "Conceptual modeling and code generation for rich internet applications.," in *Proceedings of the 6th international conference on Web engineering ICWE 06*, 2006, vol. 1, p. 353.

[3] J. C. J. Preciado, M. Linaje, S. Comai, and F. Sanchez-Figueroa, "Designing rich internet applications with web engineering methodologies," in *Web Site Evolution, …*, 2007, pp. 23–30.

[4] A. Mesbah, A. Deursen, and A. Van Deursen, "An Architectural Style for Ajax," in *2007 Working IEEEIFIP Conference on Software Architecture WICSA07*, 2007, pp. 9–9.

[5] K. Schmidt, R. Stühmer, and L. Stojanovic, "Gaining reactivity for rich internet applications by introducing client-side complex event processing and declarative rules," in *AAAI 2009 Spring Symposium: Intelligent Event Processing*, 2009, pp. 67–72.

[6] J. Louwsma *et al.*, "Specifying and Implementing Constraints in GIS—with Examples from a Geo-Virtual Reality System," *Geoinformatica*, vol. 10, no. 4, pp. 531–550, Jan. 2007.

[7] Z. L. Z. Liang and S. J. S. Jianling, "A field-oriented approach to web form validation for Database-Isolated Rule," in *2009 IEEE International Conference on Systems Man and Cybernetics*, 2009, no. October, pp. 4607–4612.

[8] D. Fernández Lanvin, R. Izquierdo Castanedo, A. A. Juan Fuente, and A. M. Fernández Álvarez, "Extending object-oriented languages with backward error recovery integrated support," *Comput. Lang. Syst. Struct.*, vol. 36, no. 2, pp. 123–141, Jul. 2010.

[9] A.-M. Fernández-Álvarez, D. Fernández-Lanvin, and M. Quintela-Pumares, "Invariant Implementation for Domain Models Applying Incremental OCL Techniques," Springer, Cham, 2016, pp. 137–154.

[10] M. Quintela-Pumares, D. Fernández-Lanvin, R. Izquierdo, and A.-M. Fernández-

Álvarez, “Implementing Automatic Error Recovery Support for Rich Web Clients,” in *WISE’10 Proceedings of the 11th international conference on Web information systems engineering*, 2010, pp. 630–638.

[11] M. Quintela-Pumares, B. Cabral, D. Fernandez-Lanvin, and A.-M. Fernandez-Alvarez, “Integrating automatic backward error recovery in asynchronous rich clients,” in *Proceedings of the 38th International Conference on Software Engineering Companion - ICSE ’16*, 2016, pp. 192–201.

[12] M. Quintela-Pumares, D. Fernández-Lanvin, and A.-M. Fernandez-Alvarez, “Domain model slicing and constraint classification for local validation on rich clients,” *J. Syst. Softw.*, 2017.

[13] M. Quintela-Pumares, D. Fernández-Lanvin, A.-M. Fernández-Álvarez, and R. Izquierdo, “Automatic Classification of Domain Constraints for Rich Client Development,” in *ICSEA 2014, The Ninth International Conference on Software Engineering Advances*, 2014, pp. 570–576.

[14] A. M. F. Á. Daniel Fernández Lanvin, , Raúl Izquierdo Castanedo , Aquilino Adolfo Juan Fuente, “Extending object-oriented languages with backward error recovery integrated support,” *Comput. Lang. Syst. Struct.*, vol. 36, no. 2, pp. 123–141, Jul. 2009.

[15] Alberto-Manuel Fernández-Álvarez ; Daniel Fernández-Lanvin; Manuel Quintela-Pumares, “OCL for Rich Domain Models Implementation - An Incremental Aspect based Solution,” in *Proceedings of the 10th International Conference on Software Paradigm Trends, Colmar, Alsace, France, 20-22 July, 2015 [part of: ICSOFT 2015, 10th International Joint Conference on Software Technologies]*, 2015, pp. 121–129.

[16] J. M. Kieley, “CGI scripts: Gateways to World-Wide Web power,” *Behav. Res. Methods, Instruments, Comput.*, vol. 28, no. 2, pp. 165–169, Jun. 1996.

[17] J. Duhl, “Rich Internet Applications (White Paper),” 2003. [Online]. Available: https://www.adobe.com/platform/whitepapers/idc_impact_of_rias.pdf. [Accessed: 06-Jul-2017].

[18] A. Bozzon, S. Comai, P. Fraternali, and G. G. T. Carughi, “Conceptual modeling and code generation for rich internet applications.,” in *Proceedings of the 6th international conference on Web engineering ICWE 06*, 2006, vol. 1, p. 353.

[19] R. Zhou, S. Shao, W. Li, and L. Zhou, “How to define the user’s tolerance of response time in using mobile applications,” in *2016 IEEE International Conference on Industrial Engineering and Engineering Management (IEEM)*, 2016, pp. 281–285.

[20] A. Leff and J. Rayfield, “Programming model alternatives for disconnected business applications,” *Internet Comput. IEEE*, vol. 10, no. 3, pp. 50–57, May 2006.

[21] W. Z. W. Zhang, “2-Tier Cloud Architecture with maximized RIA and SimpleDB via minimized REST,” in *2010 2nd International Conference on Computer Engineering and Technology*, 2010, vol. 6, pp. V6-52-V6-56.

[22] S. Hallé and R. Villemaire, “Browser-based enforcement of interface contracts in web applications with BeepBeep,” *Comput. Aided Verif.*, pp. 648–653, 2009.

[23] L. de Haan and T. Koppelaars, *Applied Mathematics for Database Professionals*. Berkeley, CA: Apress, 2007.

[24] E. Teniente and J. Cabot, “Computing the Relevant Instances That May Violate an OCL Constraint,” *LNCS*, vol. 3520, pp. 48–62, 2005.

[25] E. Miliauskaite and L. Nemuraite, “Taxonomy of Integrity Constraints in Conceptual Models,” 2005.

[26] J. Cabot and E. Teniente, “A metric for measuring the complexity of OCL expressions,” in *In Model Size Metrics Workshop co-located with MODELS’06*, 2006.

[27] J. Cabot, “Computing the Relevant Instances That May Violate an OCL Constraint,” pp. 48–62, 2005.

[28] P. Bottoni, M. Koch, F. Parisi-presicce, and G. Taentzer, “A Visualization of OCL using Collaborations,” in *≪UML≫ 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, vol. 2185, M. Gogolla and C. Kobryn, Eds. Springer Berlin Heidelberg, 2001, pp. 257–271.

[29] J. Allaire, “Macromedia Flash MX—A next-generation rich client,” *Macromedia white Pap.*, no. March, 2002.

[30] A. Flex, “Adobe Flex 2.”

[31] M. MacDonald, *Silverlight and ASP.NET Revealed*. Berkeley, CA: Apress, 2007.

[32] T. Allen, “Java Applet element proposal.” p. 4, 1995.

[33] J. Weaver, *JavaFX Script: Dynamic Java Scripting for Rich Internet/Client-Side Applications*. Apress, 2007.

[34] “Microsoft Announces ActiveX Technologies - Stories.” [Online]. Available: https://news.microsoft.com/1996/03/12/microsoft-announces-activex-technologies/. [Accessed: 30-Oct-2018].

[35] G. Paolini, “Netscape and Sun Announce JavaScript, the Open, Cross-Platform Object Scripting Language for Enterprise Networks and the Internet [Press Release],” *Dec 4, 1994*, 1994. [Online]. Available: http://tech-insider.org/java/research/1995/1204.html. [Accessed: 06-Sep-2017].

[36] J. Garrett, “Ajax: A new approach to web applications,” pp. 1–5, 2005.

[37] V. Halarnkar, “Web Browsers,” 2009.

[38] N. Iscoe, G. B. Williams, and G. Arango, “Domain modeling for software engineering,” in *[1991 Proceedings] 13th International Conference on Software Engineering*, 1991, pp. 340–343.

[39] E. Evans, *Domain-Driven Design-Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2003.

[40] M. Fowler, “Anemic Domain Model,” *www.martinfowler.com*, 2003. [Online]. Available: http://www.martinfowler.com/bliki/AnemicDomainModel.html.

[41] J. Rumbaugh, I. Jacobson, and G. Booch, *The unified modeling language reference manual*. Addison-Wesley, 2005.

[42] OMG, “Object Constraint Language,” *http://www.omg.org/spec/OCL/2.4/*, 2014. .

[43] J. Chimiak-Opoka *et al.*, “OCL Tools Report based on the IDE4OCL Feature Model.,” *Eceasst*, vol. 44, 2011.

[44] “EMF | projects.eclipse.org,” *Eclipse Modeling Framework*. .

[45] "Apache Struts 2 Validation." [Online]. Available: http://struts.apache.org/docs/validation.html. [Accessed: 10-Sep-2017].

[46] "jQuery Validation Plugin." [Online]. Available: http://jqueryvalidation.org/. [Accessed: 10-Sep-2017].

[47] "Simfatic Forms." [Online]. Available: http://www.simfatic.com/. [Accessed: 10-Sep-2017].

[48] P. Heidegger and P. Thiemann, "Contract-Driven Testing of JavaScript Code," pp. 154–172, 2010.

[49] K. Schmidt, R. Stühmer, and L. Stojanovic, "Gaining reactivity for rich internet applications by introducing client-side complex event processing and declarative rules," in *AAAI 2009 Spring Symposium: Intelligent Event Processing*, 2009, pp. 67–72.

[50] C. L. Forgy, "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem," *Readings Artif. Intell. Databases*, pp. 547–559, Jan. 1989.

[51] E. Tilevich and Y. Smaragdakis, "J-Orchestra: Automatic Java Application Partitioning," in *ECOOP '02 Proceedings of the 16th European Conference on Object-Oriented Programming*, 2002, pp. 178–204.

[52] G. Hunt and M. L. Scott, "The Coign Automatic Distributed Partitioning System," in *OSDI '99 Proceedings of the third symposium on Operating systems design and implementation*, 1999, p. 278.

[53] F. Yang *et al.*, "A unified platform for data driven web applications with automatic client-server partitioning," in *Proceedings of the 16th international conference on World Wide Web - WWW '07*, 2007, p. 341.

[54] K. Hassam, S. Sadou, V. Le Gloahec, and R. Fleurquin, "Assistance System for OCL Constraints Adaptation during Metamodel Evolution," in *2011 15th European Conference on Software Maintenance and Reengineering*, 2011, pp. 151–160.

[55] J. Cabot and E. Teniente, "Incremental evaluation of OCL constraints," *Adv. Inf. Syst. Eng.*, 2006.

[56] R. Singh and C. Vinay Arora, "Literature Analysis on Model based Slicing," *Internatonal J. Comput. Appl.*, vol. 70, no. 16, pp. 45–51, 2013.

[57] H. Kagdi, J. I. Maletic, and A. Sutton, "Context-free slicing of UML class models," in *21st IEEE International Conference on Software Maintenance (ICSM'05)*, 2005, vol. 2005, pp. 635–638.

[58] R. Kollmann and M. Gogolla, "Metric-based selective representation of UML diagrams," in *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering*, 2002, pp. 89–98.

[59] J. T. Lallchandani and R. Mall, "A Dynamic Slicing Technique for UML Architectural Models," *IEEE Trans. Softw. Eng.*, vol. 37, no. 6, pp. 737–771, Nov. 2011.

[60] A. Shaikh, U. K. Wiil, and N. Memon, "Evaluation of Tools and Slicing Techniques for Efficient Verification of UML/OCL Class Diagrams," *Adv. Softw. Eng.*, vol. 2011, pp. 1–18, Sep. 2011.

[61] J. Cabot and E. Teniente, "Incremental integrity checking of UML/OCL conceptual schemas," *J. Syst. Softw.*, vol. 82, no. 9, pp. 1459–1478, Sep. 2009.

[62] A. Shaikh, R. Clarisó, U. K. Wiil, and N. Memon, “Verification-driven slicing of UML/OCL models,” *Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng. - ASE ’10*, no. November 2015, p. 185, 2010.

[63] Claas Wilke, Dr.-Ing. Birgit Demuth, and Prof. Dr. rer. nat. habil. Uwe Aÿmann, “Java Code Generation for Dresden OCL2 for Eclipse.” Feb-2009.

[64] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns – Elements of Reusable Object-Oriented Software*. 1994.

[65] J. Palsberg and C. B. Jay, “The essence of the Visitor pattern,” in *Proceedings - International Computer Software and Applications Conference*, 1998, pp. 9–15.

[66] “EcoreTools - Graphical Modeling for Ecore.” [Online]. Available: https://www.eclipse.org/ecoretools/. [Accessed: 22-Aug-2018].

[67] “Graphviz - Graph Visualization Software.” [Online]. Available: https://www.graphviz.org/. [Accessed: 23-Aug-2018].

[68] A. Rodriguez, “Restful web services: The basics,” *Online Artic. IBM Dev. Tech. Libr.*, no. November, pp. 1–11, 2008.

[69] B. Demuth and H. Hussmann, “Using UML/OCL Constraints for Relational Database Design,” in *Proceedings of the 2nd international conference on the Unified Modeling Language*, vol. 6, R. France and B. Rumpe, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 598–613.