Journal of Systems and Software

Domain Model Slicing and Constraint Classification for Local Validation on Rich Clients

Manuel Quintela-Pumares[a]*, Daniel Fernández-Lanvin[a], Alberto-Manuel Fernandez-Alvarez[a]

[a]*Computer Science Department, University of Oviedo, Oviedo, Spain.*

ARTICLEINFO

Article history:

Keywords:
Constraints
Domain modeling
Rich client
UML
OCL
Model slicing

ABSTRACT

Web-based rich client applications 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. This local model is often a subset of the application's global domain model (GDM) that is managed on the server. Both ends should always manage their entities, relationships and constraints consistently between them. Designing such client model manually implies identifying the GDM domain elements and constraints that should also be present on the client and adapting each one of them if needed. This is a complex and error-prone task, and any additional modification to the server model requires reviewing the client side. In our opinion, all the information needed for automating the client model generation can be derived from the GDM and the set of entities involved in the client functionality. This work includes a formal description of a method that, from that initial information, combines model slicing and constraint analysis techniques to create the client domain model, and classifies the constraints according to their server independency.

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 every change [7]. 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 [4]. Because of this, the use of rich clients has become quite popular in different platforms, mainly as web applications for browsers or native applications for different smartphone operating systems.

However, this step back to the classic 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 the client can lead to inconsistencies with respect to the new state of the server object model. So every model constraint involved needs checking once the client object model is reintegrated into the server object model.

* *Corresponding author.* Tel.: +34 985103397
E-mail address: quintelamanuel@uniovi.es

Peer review under responsibility of xxxxx.

xxxx-xxxx/$ – see front matter
http://dx.doi.org/xxxxxxx

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 [23], 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 [20][28]. This would lead to a more robust client and provide a more fluid user experience, since the client would detect and prevent incorrect actions 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 GDM. In this case, the designer must decide if it may be adapted or ignored on the CDM [18].

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 [17].

This work is based on the idea that all the information needed to design a robust and solid CDM can be deduced from the GDM. We propose a method that is based on the set of GDM classes required on the CDM, and a full GDM with its OCL constraints. It detects the overlapping between CDM and GDM, and classifies the CDM OCL constraints indicating (i) the elements they affect, (ii) which ones are directly verifiable on the CDM, (iii) which are not related to the CDM at all, and (iv) which could be adapted to fit on the CDM.

This information would support the automation of the CDM design and its maintenance through the development cycle. It would help developers to design CDMs maximizing the local checking of GDM constraints insofar as possible.

This paper presents the following contributions:

- A formal description of UML class diagrams based on graphs.
- A formal graphical notation using trees for OCL invariant constraints that focuses on the visualization of elements from the class model that it constrains. These structures are called “instance trees”.
- A method that, from the abstract syntax tree (AST) of an OCL expression, generates an instance tree.
- Based on the class diagram formalization, we define a model slicing technique to generate a CDM from a GDM, using the set of classes that are relevant to the client as slicing criteria.
- A formal description, based on the instance tree notation, of a classification algorithm according to the constraint complexity level.
- A formal description based on the instance tree notation, of a classification algorithm according to server dependency for the constraints of the CDM generated.

These methods are formally described based on graph theory, so that they can be not only applied to UML class models and OCL constraints, but also to any other modeling or notation technique as long as they can be formally described as a graph —such as entity-relationship diagrams.

The rest of the paper is structured as follows. Section 2 explores the problems associated with designing rich clients, and provides a running example that will be used throughout the paper. Section 3 reviews the state of the art in different areas relevant to our work. Section 4 details our approach to address rich-client-design related issues. Sections 5 and 6 provide a formal description for both UML class models as well as OCL constraints that will be required for the formal description of our method. This is done in sections 7 and 8, where we formally describe a method to automatically generate a rich client domain model from an existing server model, and the formal foundation for the automatic classification of constraints. Section 9 briefly describes the working prototype we developed based on this method. Section 10 addresses the limitations of the current proposal. Section 11 closes this paper with conclusions and future work.

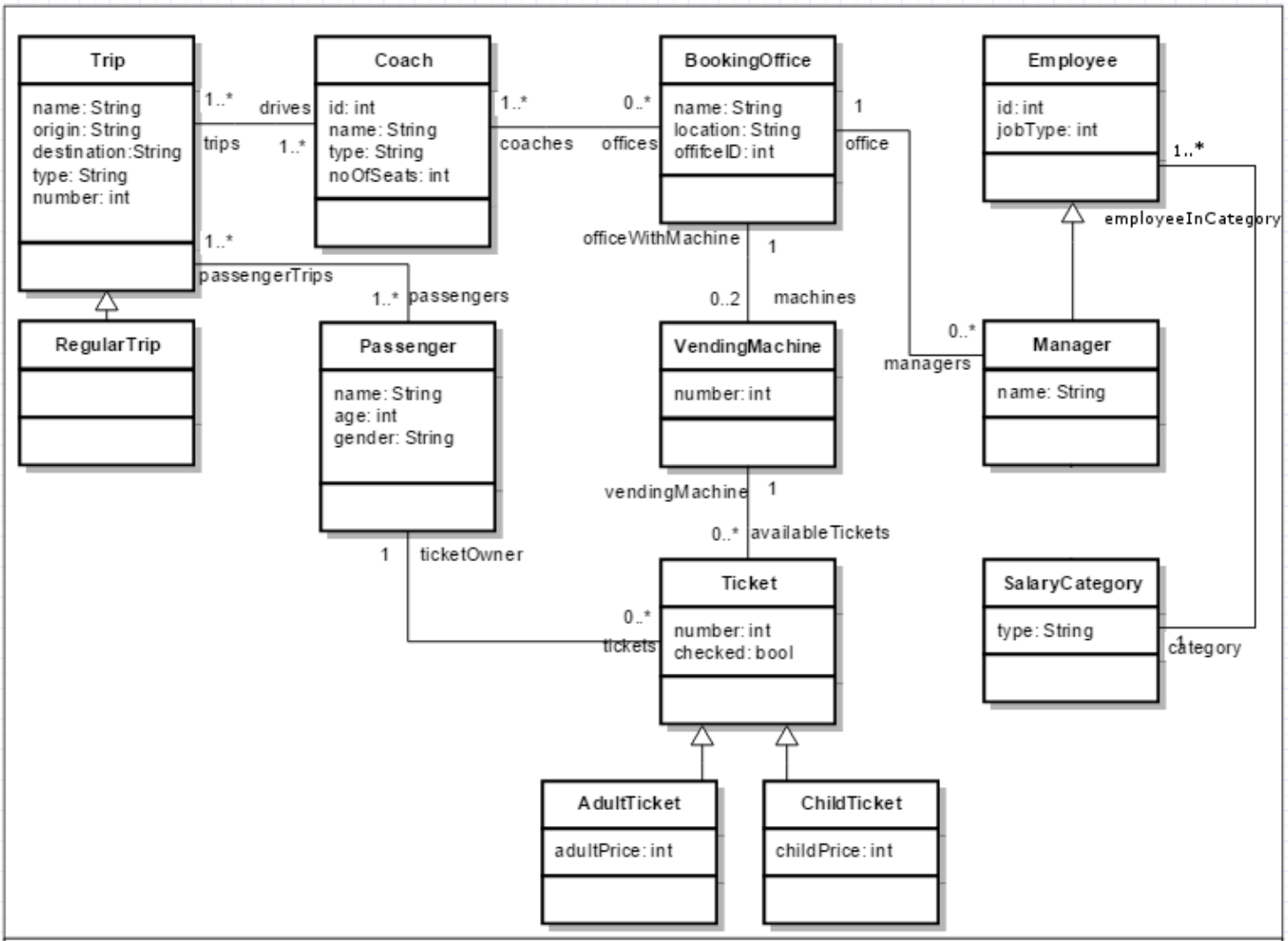

Figure 1: SDM for a coach company that manages booking offices, managers, ticket sales, passenger information, trips and coaches. Both the class diagram and the OCL constraints are managed from the server.

2. Motivation

In this work, the term "rich client" is used to refer to any kind of application —regardless of the technology used in its development or the platform where it is executed— that (i) is able to communicate with a server to send and receive data, (ii) contains its own logic that allows it to manipulate data locally, and (iii) offers interactivity without having to reload the UI or notify every single action or modification that occurs locally back to the server. As stated before, a client being able to both work with the data retrieved from the server and locally check constraints related to it would require, in most cases, a design where part of the CDM is a GDM subset. There may be other elements and constraints for the client model that have no relation at all to the GDM, but in this paper, when the term CDM is used without further specification, it refers exclusively to the part of the client model overlapping with the GDM.

This paper includes a running example to illustrate the challenges associated to managing constraints on rich clients. This model shown in Figure 1 is an adaptation of an example provided by Shaikh et al. [29]. The model is a simplification of a coach trip management system that includes the organization of trips, offices, managers, coaches, passengers, and ticket sales. This model includes several OCL constraints that should be respected at all times. This running example does not focus on providing the most complete approach towards that type of system, but rather on showing clearly the main problems found when designing domain models for rich clients. It also contains specific features relevant to our method and it is simple enough so that the processes described are easy to understand and follow.

Let us consider the possibility of implementing a client that the coach drivers could use to verify tickets and passengers as they arrive for their trip, as shown in Figure 2.

The domain model elements of this client would be based on the ones present on the GDM. Also, all the constraints eligible to be checked over the CDM are already present on the GDM. The verification of GDM-related constraints has to be done on the server; even if a previous verification has been made on the client —for security and consistency reasons—, since the data could have been compromised during the communication [20]. Although this scenario implies a design that has classes and constraints repeated on both client and server, it is not enough to select and copy the client-relevant elements from the GDM. Instead, they need to be analyzed and adapted by the developer.

Regarding the class model for the CDM, some of the relationships that on the GDM where linked to classes will cause conflict on the client. The developer needs to decide what to do depending on the type of relationship: association, aggregation, composition, or dependency. Some of these relationships may be eliminated and ignored on the CDM, which implies deciding how this affects the class, which of the methods present on the class are yet justified, or on the contrary, which could be ignored on the CDM. On the other hand, the degree of coupling in some relationships can be tight enough to justify the inclusion of the participant classes into the CDM, even though they were not initially selected for the client.

The problem becomes more complicated when the OCL constraints come into play. As part of the model, they must also be validated on the CDM. They are originally represented as predicates over the GDM, and depending on their composition, they can affect one or more elements of different types. The more elements involved in the constraint, the more complicated its adaptation to the CDM will be. We can classify constraints according to the elements they constrain.

- Attribute constraints: They constrain a single attribute of a class. Constraint 1 from the GDM (Figure 1) would be an example of this.
- Object constraints: They constrain several attributes of the same class. Constraint 2 from the GDM (Figure 1) involves comparing two attributes of the same object.
- Class constraints: They constrain several objects of a single class. Constraint 3 from the GDM (Figure 1) checks that there are not two ticket objects with the same ID.
- Domain constraints: They constraint objects of different classes. Constraints 4, 5, 6 and 7 from the GDM (Figure 1) all involve objects from several classes.

Domain constraints affect several classes, but eventually, some of them may not be required on the CDM. In those cases, the developer must decide (i) whether just to delegate that constraint to the GDM, (ii) if there is a way to divide different parts of the predicate (so that at least part of the constraint can be checked on the client), or (iii) if it can be checked on the client by retrieving certain data from the server.

All these tasks and decisions must be taken by the developer, and are dependent on the information present on the GDM. Any task that requires duplicating logic, and eventually adapting it depending on the information

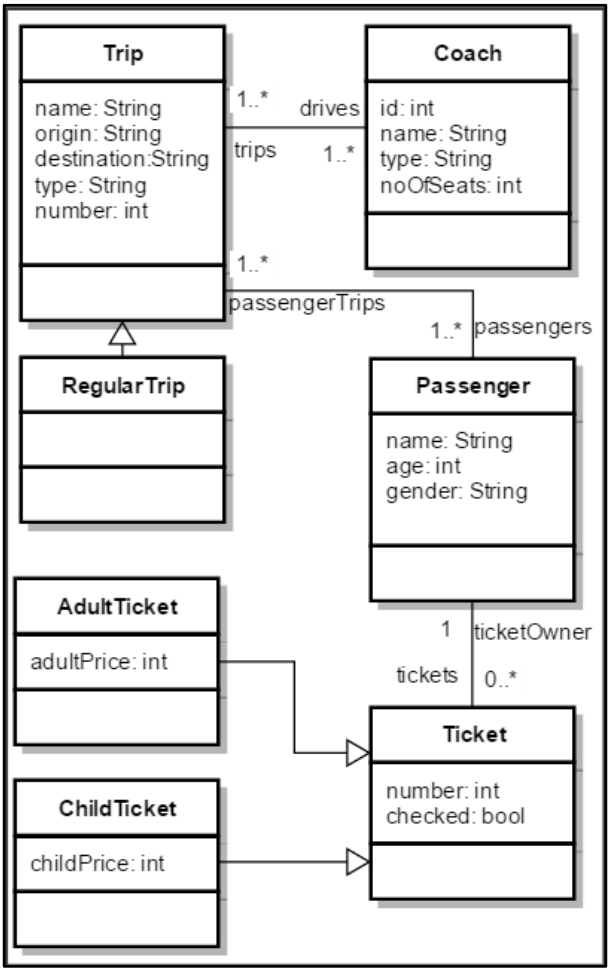

Figure 2 Class diagram for the CDM. It would be desirable to be able to verify on the client as many constraints from Figure1 as possible. The designer must analyze carefully all existing constraints and verify when this is possible.

from the previous model, is an error-prone, tedious and difficult to accomplish task. Furthermore, these problems are even worse considering that in the development cycle, changes are frequently made to the domain model, which will also affect other models that are depending on it. Besides, different parts of a system may be carried out by different teams, adding a whole set of new problems, such as communication problems.

3. Background

3.1. Rich Client Architecture

There have been several works proposing the reorganization of client-server responsibilities to increase the benefits in user experience and the robustness of rich clients. The proposals by Zhang [33] or Leff and Rayfield [16] state that the mainstream approaches for rich clients do not exploit their full potential. Typical design architectures too often delegate too much functionality to the client. The consequences of this approach go beyond a worse user experience —for example, overly complex architectures or development cycles [33], or applications that do not respond well in situations where the server is not available at all times [16]. Although the local management of robustness and constraints on the client is encouraged by these proposals, no specific means to facilitate this task are suggested.

Enterprise solutions for client side validation such as Struts[1], jQuery Validation Plugin[2], or Simfatic[3] are limited to simple form checking and are not designed to cover the complexity that client side business rules demand. There are, however, some proposals that try to fill this gap. Hallé and Villemaire [8] propose a way to describe contracts for web services, and a monitor that checks whether these contracts are being violated breached before the request is sent to the server. That way, they mean to avoid the resource expenses on the server and the delay waiting for responses that will fail. Heidegger and Thiemann [9] suggest an attribute-oriented language to define pre- and post-conditions in JavaScript.

All these proposals provide means to help the developer define and implement the constraints on the client. However, in all of them it is up to the developer to decide which constraints are relevant on the client.

Some authors acknowledge the need to have coordinated constraints in both client and server sides. Liang et al. [16] put forward a system in which validations are defined in a XML file on the server, managing constraints that involve a combination of several attributes in the client forms (attribute and object constraints). This automates the implementation of part of the client side constraints, and improves the maintenance process. However, they explicitly left out of their scope the most complex and problematic *class* and *domain* constraints. Schmidt et al. [17] designed a rule engine for the client side based on the RETE algorithm, where the constraints are defined on a file on the server. While they support the definition of complex constraints and even their validation by the server, the specific constraints affecting the client must be manually specified. Louwsma et al. [18] propose a framework based on UML and OCL to define constraints for GIS rich clients. These constraints must be manually implemented later on, and are delegated to the server side, acknowledging that classifying and evaluating some types of constraints on the client side would be useful as a future work.

Apart from these, some other approaches try to address these problems automatically. The proposals closest to our work are focused on distributing the domain logic between client and server. They analyze an existing application and automatically redistribute its components between client and server, with resulting tools such as J-Orchestra [31], Coign [11], or the platform designed by Yang et al. based on Hilda language [32]. These solutions are motivated by some of the same problems that we are facing, but are focused on optimizing existing applications following different criteria —such as memory usage, client hardware capacity, or user demand of certain functionalities over others. None of them are focused on making the most of rich client capabilities with optimal user interface response and robustness.

3.2. Model Slicing

The idea of splitting up a GDM into different parts is not new. In fact, it has been evidenced as a useful approach in many tasks, such as identifying sets of dependent components, better visualization of the different parts of the model, or evaluating smaller parts of a complex system. This kind of techniques is known as *model slicing*, a breaking-down process to extract and identify relevant model parts or related elements across a model corresponding to a specific slicing criterion [30].

Some works propose the use of these techniques for better visualization of big and complex models, trying to automatically identify different subparts that can be presented visually while being cohesive enough, according to different criteria selected by the user. Kagdi et al. [11] use the selection of an initial set of the slice as a slicing criterion. Their method determines the dependent elements that should be showed alongside the initial set. Kollman and Gogolla [13] suggest extracting metrics from the model to detect the level of coupling and dependency of the different classes, and then identifying the submodels automatically. Lallchandani and Mall [15] define a method to analyze different model views, both static and behavioral, in order to generate an intermediate Model Dependency Graph that gathers all the information about how different elements from all views are related. This intermediate graph is used to generate submodels for different types of UML diagrams (class, sequence, …) related to the slicing criteria elements.

Aside from better visualization and understanding of complex models, slicing techniques are useful to optimize model validations, as Saikh et al. [27] propose. A model using Class diagrams and OCL constraints can be validated on design to detect inconsistencies in the design before starting its implementation. Unfortunately, these techniques are computationally heavy in terms of time, and scale up very poorly as the models get more complex, even if the new additions do not affect the constraints to be checked. The solution that Saikh et al. propose is to use each OCL constraint as a slicing criterion to generate only the submodel for the class diagram that is required to evaluate that specific constraint. In this way, the model validations are applied for each constraint, but over a much smaller model, improving the efficiency and requiring much less time to complete.

[1] Apache Struts 2 Validation: *http://struts.apache.org/development/2.x/docs/validation.html*. Accessed: 2016-02-24.

[2] jQuery Validation Plugin: *http://jqueryvalidation.org/*. Accessed: 2016-02-24.

[3] Simfatic Forms: *http://www.simfatic.com/*. Accessed: 2016-02-24.

As can be seen, there are proposals for dividing class diagrams using a set of core classes as slicing criteria, but these do not take into account that the resulting model has to be functional. They are intended to improve the visualization of the model, and not to generate new design elements. Although their algorithms serve as inspiration, under some circumstances necessary submodel elements are left out —such as parent classes—, so using them to generate a CDM from a GDM would require some adjusting.

Furthermore, none of these proposals provides slicing criteria strategy over OCL constraints. Although Saikh et al. use OCL constrains as their main slicing criteria, they generate a submodel for that constraint. However, our aim is precisely the opposite, to find relevant and proper OCL constraints from a submodel of a class diagram. Nevertheless, their proposal is useful since their analysis of the elements affected by an OCL constraint can be used to infer whether that constraint is valid for a given submodel or not.

3.3. Domain Model Formalization and Alternative Representations

Visual representations of models have proved to be an expressive and intuitive way of representing information about the components of a design, their topology, and their interactions. On the other hand, constraints are often represented using languages based on predicate logic. The most widespread example is the UML standard, with visual representations for static and dynamic views of the model, and OCL predicates to represent its constraints.

UML is a complex standard designed for flexibility, but lacks a formal description [19] that is essential to describe algorithms for automated model analysis and validation, metrics extraction, enabling transformations, or as a general representation that can be applied to different standards that share some similar features (as may be class diagrams and entity relationship diagrams).

There are several proposals for the formalization of UML diagrams, but they are limited to specific parts of the language. So their formalization is tailored to the specific area of study that encloses their target issues. For instance, Liang et al. [22] provide a formalization of class, state and collaboration diagrams using graph theory, aiming to be compatible with prover tools such as PVS. With the same requirement —namely, being verifiable by existing tools—, Kyas et al. [34] formalize both UML models and OCL constraints. Kuhlmann and Gogolla [14] provide a transformation of both into relational logic.

While models defining software topology, structure and behavior have been mainly diagram-based, constraints tend to be usually represented by languages based on first-order logic and set theory. Although this allows for accurate and precise constraint definitions, as predicates get more complex, it is harder to visualize the elements involved in the evaluation of a constraint. Many proposals for addressing this drawback using graph-based representation of constraints have been suggested. Constraint diagrams [12] get inspiration from Venn diagrams and graph-like directed edges to represent the objects taking part in the invariants, their cardinalities and relationships. This concept was later extended and evolved into Spider diagrams [10]. Conceptual graphs [21] use different types of nodes and edges, labeled with logic predicates to define a model´s structure, behavior and constraints. Collaborations [3] seek to provide a visualization of the navigation paths required to evaluate a constraint, with elements to visually describe if a navigation is to a single object or an object collection. They would help developers to get a sense of the scope of the object graph required to evaluate a constraint, and simultaneously, define the logic predicate that applies to it separately. Arendt et al. [1] provide a nested graph-based proposal, where navigations are translated as attributed graphs. while Boolean operations are applied directly to these constructs. Bottoni et al. [2] have also proposed language-independent visual representations of patterns to analyze conflicts and dependencies that are usually limited to textual representations. Rensink [26] also provides a graphical representation of first-order logic based on edge-labeled graphs.

Most of these proposals try either to analyze and automate the validation of models, or to provide a better visualization of the constraints. Formal descriptions of UML models based on graph theory can aid us in formally defining the slicing process to automatically generate a CDM. In addition, the graphical representation of constraints is useful to formally define the structure pattern of the instance model, the cardinality of their relationships and the hierarchy that an OCL constraint requires to be evaluable. In this way, we can map if an OCL constraint is evaluable for a given CDM, or (in case it is not) effectively determine which missing elements prevent the constraint from being evaluable. This would provide useful information to determine how to adapt these constraints to the client.

To achieve this, a formal way of describing the elements affected by an OCL expression would allow us to address the challenges associated with knowing when a constraint can be evaluated and when it cannot. Although there are already proposals on how to represent predicates graphically, for our method we require only a simple representation free from the additional information about the operations that are going to be evaluated over it, which is not required for the set of challenges analyzed on this work.

4. Proposal

The main goal of this work is to provide technical support and assist the designer insofar as possible during the GDM slicing process. We propose a method consisting of three stages. The scope of the method is currently limited to OCL invariants, excluding other types of constraints such as preconditions or postconditions. The input is an existing GDM with its OCL constraints, and the designer's selection of the classes from the GDM that should be present on the client. With this information, a proper CDM will be automatically generated, including the constraints that are locally verifiable and an additional classification and information about them that can aid the designer in further decisions beyond the limitations of the automatic method.

During the first stage, the class model and the relationships between classes will be analyzed. The cardinalities of the relationships are considered as implicit constraints contained within the diagram. To ease the analysis of the constraints related to the model, these relationships are used to generate explicit predicates that will be added to the rest of the model constraints. This will support a homogeneous processing of all of them.

In a second stage, model slicing techniques will be applied. A specific CDM will be generated from the GDM, based on the sets of classes that the designer requires to be present on the client.

The final stage identifies which constraints are related to the newly created CDM model and classifies them according to their level of dependency with the GDM. It also generates information for each constraint defining how many objects from each class, and attributes it requires to be checked. This information is intended to be easily accessible for the designer, who can use it to decide how to make further manual adaptations to the model, if needed.

These processes will be described based on formal notations, so that the algorithms are described in a precise way independent of any specific notation or technology. This also has the advantage of providing a more

general solution, since any diagram-based notation that can be expressed in a similar formal way may benefit from this method, such as entity-relationship diagrams or domain-driven design diagrams, as well as different graphical or predicate-based languages for constraint definitions.

4.1. Generation of Explicit Constraints from Class Diagrams

Class diagrams contain many implicit constraints that are expressed graphically without the need for additional languages such as OCL constraints. This approach makes these constraints easily understood and intuitive for designers and developers. However, when trying to apply constraint automatic analysis, having two different sets of representations (graphs and predicates) for different types of constraints makes things more complicated. In order to solve this, every relationship present on the class diagram is analyzed, and an OCL constraint generated for each of the cardinalities present. OCL constraints are also generated for composition relationships to ensure that there is compliance with the relationship semantics.

Once all the constraints of the model are in predicate form, the simplifications proposed by Cabot et al. [5] are applied to reduce the amount of expressions in further analysis.

4.2. Model Slicing

The model slicing process can be divided into two main stages [28]. During the first stage, the goal is to determine the core elements relevant to the slice —in our case, the CDM. And during the second one, it is to iteratively detect the elements that are closely coupled to these core elements, and automatically add them to the slice.

With the appropriate slicing criterion, we make things easier for the designer, since they will only need to describe their core needs, and the rest of the related elements will be deduced automatically.

In our case, the core elements will be the class names from the GDM that will be required on the client. After this, we will detect which relationships and additional classes are closely coupled to those core classes and, therefore, will form part of the slice. With this approach, classes outside the initial set determined by the designer may be automatically included in the final slice. And elements from classes such as methods may be deleted if they do not meet specific criteria for the CDM. This would produce a valid domain model that is a subset consistent with the server model.

There is no check performed over the initial selection by the designer. For any selection made, the method will ensure that the appropriate elements coupled to the selected ones are included. It is the designer's responsibility to appropriately select the core client type they intend to model.

The next step in the model slicing process is to determine which of the server constraints belong to the client slice.

4.3. Classifying Constraints by Dependency Level

Once we have a GDM subset, some of the constraints defined for the server could also be checked at runtime independently on the CDM, while others cannot. Three potential server dependency levels are identified for each constraint: (i) server-dependent, (ii) potentially server-dependent, and (iii) server-independent.

OCL constraints that refer to elements that do not exist on the CDM are not relevant. They are server-dependent constraints and could never be evaluated on the client side. Constraints that only refer to elements present on the client can be potentially server-dependent, or server-independent. If a client retrieves objects from the server asynchronously on demand, and a constraint requires accessing several different object instances to be verified, but not all the required instances are currently downloaded on the client, it is a potentially server-dependent constraint. That constraint will not be completely verifiable until the client retrieves the appropriate information from the server. If a constraint can always be checked without having to communicate with the server at all, we consider it a server-independent constraint. For instance, constraints that require a single object to be checked are always completely independent constraints. If the object is on the client, it can be verified without further communication.

To detect the level of dependency of an OCL constraint for a given client, we need to evaluate its expression and detect the elements it constrains. It is not enough to know the classes affected by it. We also need to know which properties are accessed, and the minimum number of objects required to evaluate the constraint —given that a constraint that requires several objects to be checked may be potentially server-dependent.

4.4. Formalization

The OCL language is aimed to define constraints that any instance of the object graph defined by the class model must satisfy. However, the definition of these constraints is built by referring to the elements of the class model that defines its structure, but the verification of the constraints is done over the object graph state at any point in time.

For an OCL constraint to be verified, it requires a minimum number of objects of a certain class, with a specific topography of relationships linking them in the object graph. In other words, any constraint could verify a set of potential object graphs that share certain features. If we want to identify which of the constraints defined for the GDM are verifiable on the client, we need a precise and unambiguous representation of the generic features of the type of object graph necessary for a specific constraint to be evaluated. Knowing this, if a class model can produce that variety of object graphs, then the constraint is verifiable in that class model.

UML class diagrams and OCL constraints have been evidenced as powerful tools for model definition. Nevertheless, they are not well suited to easily represent different aspects of the problem we are dealing with, nor to reason about them, or to provide a logical foundation to classify constraints. We require a formal description to tackle this problem unambiguously. And it must be easy to translate from standards into formalization and the other way around. This formalization should focus on the representation of OCL constraints, but since they refer to the class diagram, a formal way to describe it would also be required.

The graphical representation of UML class diagrams is based on nodes that contain different types of attributes, and various types of relationships between them that also may contain attributes. This kind of representation is easily converted into graphs with attributed nodes and edges. Regarding the constraints, in order to identify the level of dependency, it is necessary to know not only the elements from the diagram that they refer to, but also

the number of potential instances of each type involved, and what the relationships are like. The notation should be a single representation able to express a variety of potential valid object graph states.

The formal description should work with the standard UML and OCL approach, but should be general enough to be compatible with other related scenarios or alternative representations where a graph structure is bounded by predicate constraints. Similar scenarios can fit our proposal and be formalized in the same way, such as entity-relationship diagrams. Approaches like domain-driven design define elements such as aggregates, value objects and entities that can be graphically expressed in ways that could also be adaptable to this formalization.

The notation we will use will be mainly based on set and graph theory. We will define the elements in a set using the SET={a, b, c, …} notation. Most of the sets we will define will consist of tuples, that will use the ∀ element∈SET element = ($a_{element}$, $b_{element}$) notation; where the elements of the tuple in lower case represent single values, and the elements in upper case represent sets. When referring to the value that has a specific attribute of an element that is a tuple, we will use the element[attribute] notation, while the number of elements present on a set will be represented by |SET|.

5. Formalization of UML Class Models

A class diagram can be formalized as an attributable directed graph, where each class is a node and each class property is an attribute of the node. The association, composition and inheritance relationships can be represented as edges with attributes that specify type, name and cardinality constraints of the relationship.

The formalization used to describe our method considers only attribute properties for the classes, and will ignore association classes or other relationships like “uses”. Describing other elements such as methods would not make the description of our method any clearer and would result in a more convoluted explanation. And once the method is made clear, taking them into account would involve the same principles. Other authors have developed more complete formalizations to describe different aspects of UML [19][22][34][14], and their insights could be easily adaptable to our method, should a more complex and complete description be needed.

A class diagram will be represented by a directed graph with attributed nodes. It will have two edge types (for association and inheritance relationships) which can also be attributed to describe the cardinality constraints of the relationship.

A node will have attributes representing the attributes of the class, and each one of them will have a name and a type. A type is a set defining a range of values, for example, INTEGER is the set of integers, STRING is the set of all possible character strings and BOOLEAN is the set of TRUE and FALSE values. TYPE is the set of sets containing all possible existing types, TYPE = {INTEGER, STRING, BOOLEAN, DOUBLE, ...}. An ATTRNAME set is defined to refer to the set of all valid attribute names of the different elements, which will avoid confusion with the STRING type used as a type.

CLASS is the set of all possible classes. Each c element of the CLASS set is a tuple **∀ c∈CLASS, c = (n_c, A_c)** tuple; n_c being an ATTRNAME element containing the name of the class and A_c a set of pairs a representing the class attributes. Each attribute pair consists of a name that identifies it within the class, and a type, **∀ a∈A_c, a = (n_a, t_a)** type, where n_a∈ ATTRNAME, t_a∈TYPE.

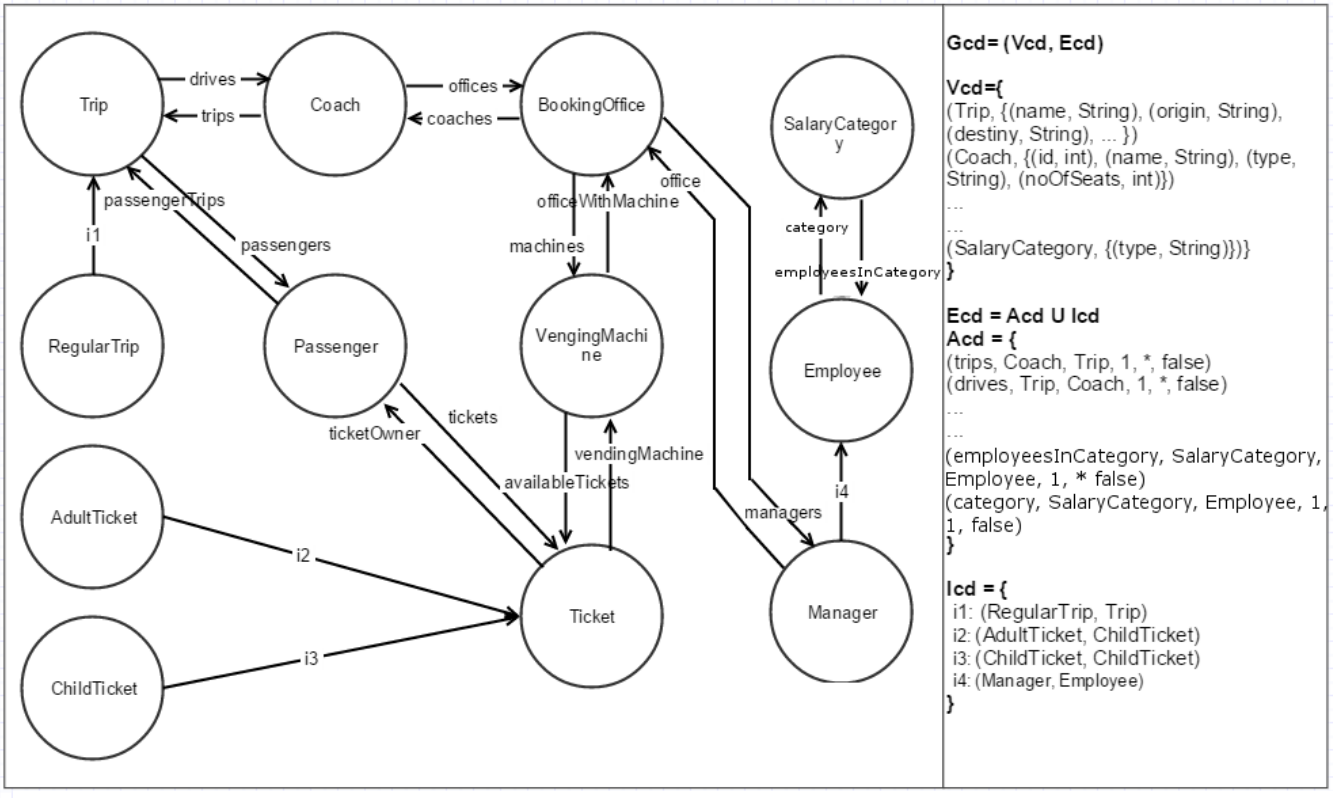

Figure 3: Formal description of the class diagram depicted in Figure 1. It includes a V_{cd} node set, an edge set that is the union of the A_{cd} associations sets and the I_{cd} inheritance relationships set. Only some elements of the sets are shown.

A class diagram can be represented as a graph $\mathbf{G_{cd} = (V_{cd}, E_{cd})}$ graph where V_{cd} is a set of nodes representing the classes and E_{cd} is the set of edges representing the relationships. Each c element of the V_{cd} set is an element of the CLASS set, $V_{cd} \subseteq CLASS$. E_{cd} is a set of edges representing the relationships between the classes. E_{cd} is the union of the sets of the two edge types, $E_{cd} = A_{cd} \cup I_{cd}$ where A_{cd} is a set representing associations and I_{cd} is a set representing inheritance relationships.

A_{cd} is a set representing the associations between classes. Each element of the set is a tuple $\mathbf{\forall\ a \in A_{cd}\ ,\ a = (n_a, o_a, d_a, l_a, u_a, c_a)}$ tuple where $n_a \in$ ATTRNAME contains the name of the relationship, $o_a \in V_{cd}$ and $d_a \in V_{cd}$ are its origin and destination classes, $l_a \in$ INTEGER and $u_a \in$ INTEGER $\cup$ * its lower and upper bounds representing the cardinality of the relationship (the symbol * represents a "many" cardinality for the upper bound), and $c_a \in$ BOOLEAN indicates if it is a composition relationship or not.

Icd is a set representing the inheritance relationships between classes. Each element of the set $\mathbf{\forall\ i \in I_{cd},\ i = (c_i, p_i)}$ is a pair that contains the child and parent classes, $c_i \in V_{cd}$ and $p_i \in V_{cd}$.

Figure 3 shows part of the description of the class diagram shown in Figure 1 formalized as a graph.

6. Formalization of OCL constraints

A constraint is a predicate that must be true for any state of the object graph. We can break down a constraint into two fundamental parts: the structure of the elements that it constrains, and the predicates that are evaluated over those elements. An OCL constraint always has a context, the base class of the instance over which it will be evaluated. Within the body of the constraint, this base instance is referenced using the '*self*' variable. OCL expressions provide means to navigate from that instance through the associations and properties described on the class diagram, allowing flexibility to describe navigation paths that define an object graph structure. It does not describe a single fixed object graph, but a set of potential object graph states that share certain features reached through a specific navigation path. If an object graph is consistent with the scope of a constraint, it can be evaluated. If not, the constraint is ignored because it does not have the minimum elements required to calculate its validity.

For example, a constraint that verifies the age of a person is not relevant in a state where no person instances have been created yet. A constraint that compares two person instances' age cannot be evaluated unless we have a minimum of two person instances in the object graph. When navigating an association in OCL, the returning element of that expression can vary according to the cardinalities described in the class diagram. A relationship with cardinality *one* will result in a single object; a cardinality with a '*many*' relationship will result in a collection, and several consecutive navigations of collections will result in a bag.

There have been proposals to visualize these expressions as collaborations [3], Spider Diagrams [10], patterns [2] or nested models [26]. These approaches try to describe constraints in a visual way, but their goal is to include information about the structure and the predicates over them in the same diagram. For our purposes, we want to separate both aspects of a constraint, so that we can analyze the elements affected by the constraint separately from the predicate.

Most of the information required to understand the elements affected by an OCL invariant are on its expression. We can represent an OCL expression with an abstract syntax tree (AST) based on the OCL metamodel [45]. The task of obtaining the AST of an OCL expression is eased by existing modeling tools such as the Eclipse Modeling Framework[6] or Dresden Tools[7].

The resulting AST includes information about both the structural elements involved in the constraint and the operations performed over them. The method presented in the next sections is not affected by the specific logic operations involved in the evaluation; only the elements required for the operations are relevant. Due to this, most of the information represented by the AST is not relevant to us, and using that structure in the formal analysis of the elements affected by the constraint would only make our task harder.

Our approach is, assuming a valid OCL invariant, to process its AST to generate a new and much simpler tree structure that contains only the core information relevant to our method. We will refer to this structure as an instance tree. Since an invariant will always come from a single 'self' instance of the context class (as described in section 7.3.3 of the OCL 2.4 specification[5]), the rest of the instances evaluated will be reached by navigating their associations. We can represent the elements required to evaluate a constraint as a tree with attributed nodes and edges where the root node will represent the 'self' instance. Each node of the tree will be called *instance node*, and will represent a set of instances of a specific class, that is reached through a certain navigation path from the root. The edges that connect the instance nodes tree will have the same information as the associations that are defined in the class graph, including the association cardinality. The range of the potential amount of elements contained in each instance node can be assessed looking into the cardinality values defined for the different associations needed to reach that node, as defined in the following chapters.

The aim of this representation is to serve as an intermediate structure to help to identify the elements required by the logical predicate to be evaluated. So the nodes should include additional information such as which attributes are used in the constraint and which variables are used to refer to the same class type in the same OCL expression.

It is also important to keep track of the navigation path followed through the instances required to evaluate the constraint. This might be the case in constraints where a cycle is generated. For example, in an invariant with Passenger as context, the following path could be used: self.passengerTrips.passengers. This structure would not produce a graph with an edge that goes back to the initial "self" node, but a tree where the root node would be a Passenger type (self), then a Trip-type child and finally another Passenger-type child down the hierarchy. With this structure, we can understand the path required to evaluate the constraint.

It is also important to keep track of the number of variables used to refer to the same class type on the same OCL expression, since it is an indication of several instances of the same class required to evaluate the expression. For instance, if a predicate has two different variables, both

[4] http://www.eclipse.org/articles/printable.php?file=Article-HowToProcessOCLAbstractSyntaxTrees/index.html
[5] http://www.omg.org/spec/OCL/2.4/
[6] https://www.eclipse.org/modeling/emf/
[7] http://www.dresden-ocl.org/

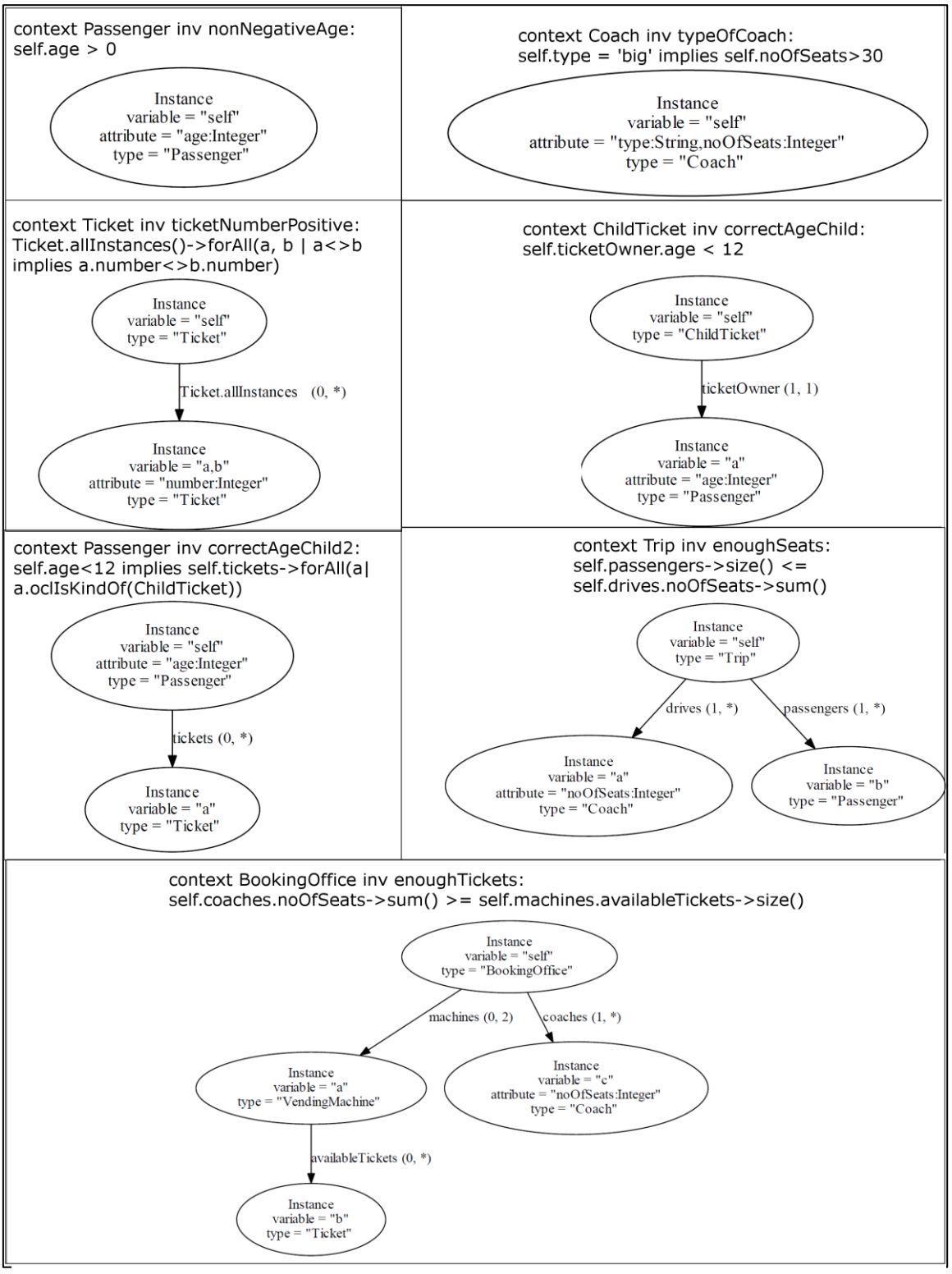

Figure 4: Instance trees for each constraint described for the GDM

Person class, to compare their age attribute, it is an indication that at least two Person instances are required to be able to evaluate said predicate.

We will define the INSTANCENODE set as the set of all possible instance nodes in the tree. Each v_i element of the INSTANCENODE set is a tuple $\forall\, v_i \in$ **INSTANCENODE,** $v_i=(c_{vi}, V_{vi}, A_{vi}, CH_{vi})$ where $c_{vi} \in$ CLASS is the instance class, V_{vi} is a set of ATTRNAME elements that represents different variables that may represent that set of instances. A_{vi} is a set of pairs representing the attributes, equal to the one described for class graphs. Each attribute of A_{vi} is a pair $\forall\, a \in A_{in}, a=(n_a, t_a)$ where $n_a \in$ ATTRNAME is the name of the attribute, and $t_a \in$ TYPE is its type. The instance nodes

will only contain the attributes referred to in the OCL expression, in order to identify the ones that are being evaluated on this expression, and not all the existing ones defined for that class on the class graph.

Each element of the set CH_{vi} is a tuple representing each child node of the instance node and the edge connecting them, $\forall\ ch_{vi} \in CH_{vi},\ ch_{vi} = (d_{ch}, n_{ch}, l_{ch}, u_{ch})$ where $d_{ch} \in$ INSTANCENODE is the child node, $n_{ch} \in$ ATTRNAME is the name of the edge connecting both nodes, $l_{ch} \in$ INTEGER is the lower cardinality constraint of the edge, and $u_{ch} \in$ INTEGER $\cup$ * is the upper cardinality constraint of the edge.

As shown in Figure 4, the instance tree structure only represents the elements needed to verify the constraint predicate, but does not provide the logical predicate itself. Although many approaches and different notations could be used to address the formal description of the predicates over this type of structure, it is currently a matter out of the scope of the method presented in this work.

6.1. Generating Instance Trees from OCL Expressions

An AST for an OCL expression will have several types of nodes representing the different parts of the expression, such as operations, property calls, arguments, variable declarations, etc.

Our method to generate the instance tree from the AST consists in traversing the AST in post-order. For each node of the tree, its children are always processed first, and no node is traversed until all its children have been already processed. As we traverse these AST nodes, we will be generating a separate structure of instance nodes that will be our instance tree. Figure 5 shows the AST for the constraint EnoughSeats, with the traversal order.

With this approach, when reaching some of the terminal nodes of the AST, we will start generating provisional instance nodes. With this method, several provisional instance nodes can be generated from each child of an AST node. When all the children of an AST node have been traversed and have generated their own provisional instance nodes, these nodes can be merged appropriately if they meet certain conditions. When the last node of the AST is traversed (its root), all the provisional trees generated through each branch are merged into the final result. Figure 6 shows this process for the AST described in Figure 5.

The way provisional instance nodes are created and merged depends on the type of the AST nodes reached. First we will define the process involved in merging two instance nodes. Once this operation is stablished, we will describe how the provisional nodes are created and merged depending on the type of AST node that is being traversed.

We will define a *merge* function that will receive two instance tree nodes v_{it} as parameters, and return a single instance tree node as a result.

$$merge(v_a, v_b) \rightarrow v_m \tag{1}$$

For two instance nodes to be able to merge, they must be of the same class, and share at least one variable name.

$$v_a, v_b \in INSTANCENODE, v_a[c_{vi}] = v_b[c_{vi}] \wedge \exists a_{va} \in v_a[V_{vi}], \exists a_{vb} \in v_b[Vv_i] / a_{va} = a_{vb} \tag{2}$$

If they share these initial conditions, the newly merged instance node will have an attribute and variable sets equal to the union of those same sets from both nodes.

$$v_a, v_b, v_m \in INSTANCENODE, v_m[A_{vi}] = v_a[Avi] \cup v_b[A_{vi}], v_m[V_{vi}] = v_a[V_{vi}] \cup v_b[V_{vi}] \tag{3}$$

The set of child nodes of the merged instance node will be equal to the union of the child sets of both nodes.

$$v_a, v_b, v_m \in INSTANCENODE, v_m[CH_{vi}] = v_a[CH_{vi}] \cup v_b[CH_{vi}] \tag{4}$$

Finally, if the two nodes share a child that has the same type and edge name, the merge function will be called recursively to merge both children. This merging process is applied only to the direct children of the two merged nodes. If two of these child nodes are merged, and they also have children, those will be treated recursively. It is important to notice that this way, it is possible to produce a tree that has several nodes of the same type, as explained in the previous section.

$$\exists ch_a \in v_m[CH_{vi}], v_{cha} \in ch_a[d_{ch}], ch_b \in v_m[CH_{vi}], v_{chb} \in ch_b[d_{ch}] / ch_a[n_{ch}] = ch_b[n_{ch}] \wedge v_{cha}[c_{vi}] = v_{chb}[c_{vi}] \rightarrow merge(v_{cha}, v_{chb}) \tag{5}$$

Once the merging process has been stablished, we can describe how and when the different instance nodes are created and merged as the AST is traversed. We will identify the AST node types that will have an effect on the instance tree generated. This method will only describe a few of the node types that represent the OCL metamodel. The ones not present here, are just traversed without any effect.

We will define the ASTNODE set as the set of all possible AST nodes, where an AST node **astn** is a tuple $astn = (t_{astn}, A_{astn}, CH_{astn})$, where $t_{astn} \in$ ATTRNAMES is the AST node type from the OCL metamodel, each element of the A_{astn} set is an attribute describing certain properties of the node $\forall\ a \in A_{in},\ a = (n_a, t_a)$ where $n_a \in$ ATTRNAME is the name of the attribute, and $t_a \in$ TYPE is its type and the CH_{astn} set contains all its children, which are a pair $\forall ch \in CH_{astn},\ ch = (n, astc)$ where n is the name of the edge joining parent and child $n \in$ ATTRNAMES and $astc \in$ ASTNODE is the child node.

We will describe each AST node type relevant to our method. It is important to note that the following list does not include all possible attributes or child types, but rather the main attributes and potential child references that are relevant to our method.

-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)...})

We will define the *processNode(astn)* function, where astn is the AST node to process. Its result is the root instance node of the provisional instance tree $p_v \in$ INSTANCENODE generated after processing this AST node and all its children.

$$processNode(astn) \rightarrow p_v$$

The *processNode* function will behave differently depending on the type of AST node being processed. Since there are multiple possibilities while managing each node type, we will stablish some previous considerations to simplify the notation.

While processing an AST node, it may be required to check the provisional instance trees that have been generated by the previous processing of each of the current AST node children. We will define a

getChildInstanceNodes() $\rightarrow P_v$ function that will provide a Pv set with the root instance nodes of the provisional instance trees generated by each of the children of the AST node being processed. In some cases, we may need to get the specific instance tree generated by one specific child of the current AST node being processed. For those situations, we will also define a *getChildInstanceNode(edgeName)* $\rightarrow p_v$, where edgeName is the name of the edge connecting to the AST child node. The *processNode* function will always return the provisional instance tree generated until now.

When defining the different function behaviors for the different AST nodes, we will use the *v* variable to denote the newly generated instance node.

During the process, some information may be required from the class graph associated to the constraint (for instance, association cardinality bounds). We will refer to the graph representing the class diagram as G_{cd}.

Each instance node has always at least one variable name that will identify it. Not all AST nodes that refer to instances have an attribute that can be used to identify that instance, but it is rather referred to through its navigation path. In those cases, we will define a *generateImplicitVar()* $\rightarrow$ *ATTRNAME* function that will generate a unique variable name for that instance.

Also, while processing some provisional instance trees, we will come across situations where we have a root instance node with a chain of children, where each one of them has only a single child. We will refer to this type of structure as an *instance chain*. In some of these situations, we will need to access the last child of that hierarchy. We will define a *getLastChild*(v_i) $\rightarrow v_l$, function that will traverse the received node to the last child of the chain, and return it. On other occasions, all children will have to be collected, from the root to the last child in the instance chain hierarchy. For that purpose, we will define a *collectAllChainInstanceChildren*(v_i) $\rightarrow V_i$ function that will return the set of all the child nodes below that node. We will also need to define a function that collects all the direct children from a single node, but not any other children further down the hierarchy, *collectAllDirectChildren*(v_i) $\rightarrow V_i$. The two first functions are only intended to be used with nodes that have an instance chain structure, and are never used in cases where a regular tree with several branches can be produced. Similarly, the last function is only used in scenarios where the structure produced is a single node with only one level of children.

After all these previous considerations, we start defining the behavior of the processNode function for each AST node type.

A variable AST node will generate a single instance node v, with the same class as the AST type and a variable name equal to its name. This provisional instance node is returned within the P_v set.

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

A TypeLiteralExp AST node will generate a single instance node v, with the same class as the AST type. TypeLiteralExp does not include a variable to refer to the instance, so we add an implicit unique variable to the instance node.

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

A PropertyCallExp AST node will always have two children: a source and a property. The property AST node does not generate any instance nodes when processed, but can be consulted from this node. Due to the nature of the AST node types allowed as source for PropertyCallExp, the provisional instance node generated by it will always be a single instance chain. If the property child of this node is of a primitive type, an attribute with the property name and type will be added to the last child of the instance chain generated by the source. If the property child of this node is a class, a new instance node with an implicit variable name will be created, and will be added as a child to the last instance node of the instance chain source. The child edge will have the same name as the property, and will have lower and upper cardinalities equal to the edge with that name on Ecd ∈Gcd.

processNode(PropertyCallExp) $\rightarrow p_v$:
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

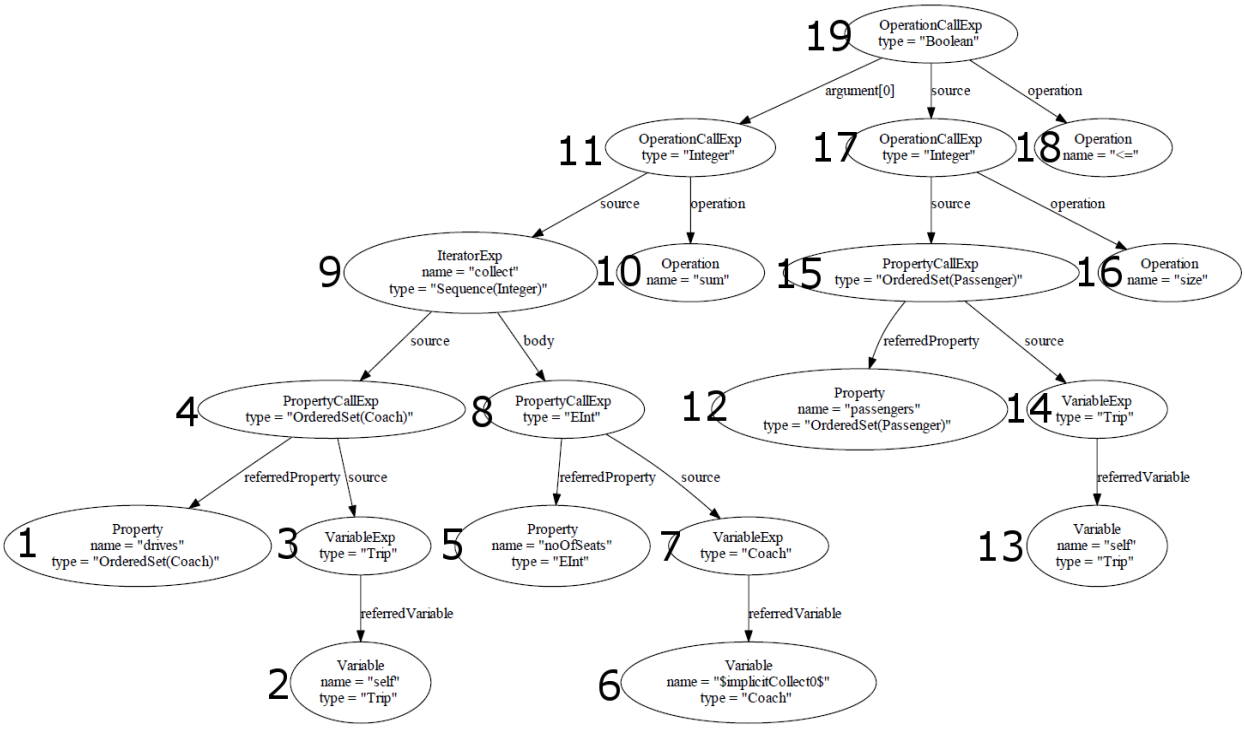

Figure 5: AST for the EnoughSeats OCL expression. The numbers represent the traversal order in which the *processNode(astn)* function will act.

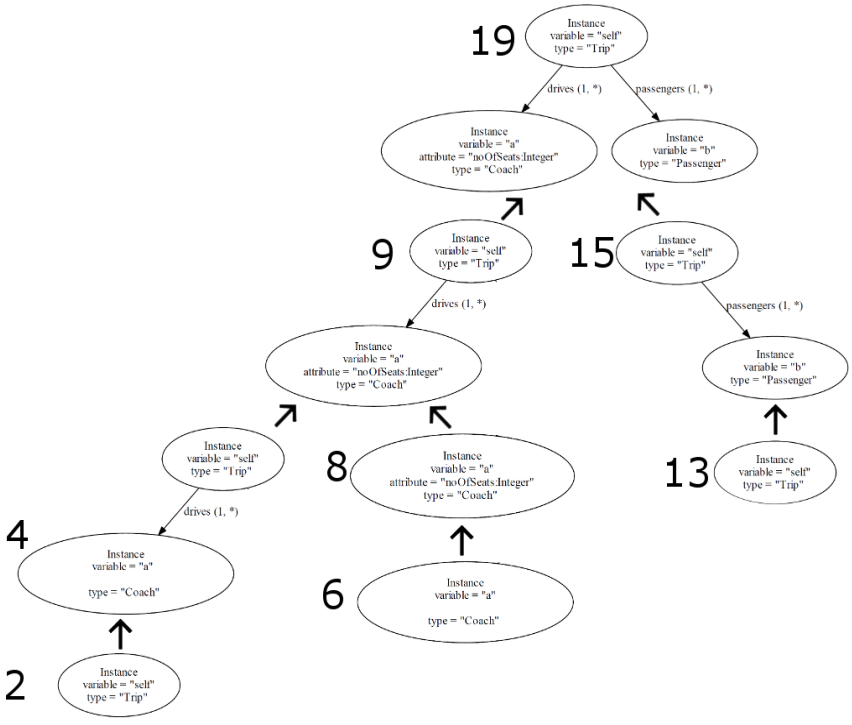

Figure 6: Representation of the generation of provisional instance nodes for the EnoughSeats OCL constraint. The nodes are generated while traversing the AST tree described in Figure 5. The numbers represent the order in which the nodes are generated and modified after the AST node is traversed by *processNode(astn)*. On processing the AST nodes 9 and 15, it can be observed how several provisional instance nodes generated by their children are merged. The resulting tree after processing the AST node 19 is the final result for the EnoughSeats OCL constraint.

An IteratorExp always has two AST children, a *source* and a *body*. Once processed, the source will always retrieve a single *instance chain* (we will call it *source instance chain*) as a provisional instance node. The instance nodes of the *body provisional instances set* will always be of the same classes as the ones present in the *source instance chain.* When an IteratorExp is processed, the *body instances set* are always merged with the source instance chain, but the way of doing this varies depending on the type of AST node found in the body of the IteratorExp.

If the body is an OperationCallExp type, the source root node is compared to each root of the different body instance nodes, and when it finds one of the same class, the body node merges with the source node. This process is repeated with each child of the source instance chain, until all the nodes from the body are merged. This will always happen, since all the classes of the body nodes exist in the source instance chain.

If the body node is of any other AST type, the body will always retrieve a single instance chain. And in that case the root of the *body instance node* becomes the child of the last element of the *source instance chain*.

processNode(IteratorExp) $\rightarrow$ p_v:
s_v=getChildInstanceNode[source],
if body[type] $\in$OperationCallExp $\rightarrow$
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}, b_v),
$p_v = s_v$

Finally, the OperationCallExp AST node usually has a source child (which will always retrieve an *instance chain*), an operation child, and an indeterminate number of argument children. In most cases, the operation type is irrelevant to our method, so it can be ignored, but there is an exception. If the operation node is an "allInstances" type, the operationCallExp is processed differently. Although on the OCL metamodel the allInstances is treated as an operation, we will treat it as a special type of navigation path, that connects the 'self' instance to a set containing all the instances of a certain class. This simplifies the way of representing this operation´s semantics for the purposes of our method.

When it is an allInstances operation, there are no arguments, only the operation source and type. In those cases, we create a provisional instance node with variable name self and class equal to the context, and another instance node with an implicit variable name and the same class as the source root node. The name of the edge connecting them will be "Class:allInstances". Its lower cardinality bound will be 0 and the upper cardinality bound will be 'many' (*).

In any other case, the operation type is ignored, and all the provisional instance nodes received from processing all the children are merged when they are of the same type and share a variable name.

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

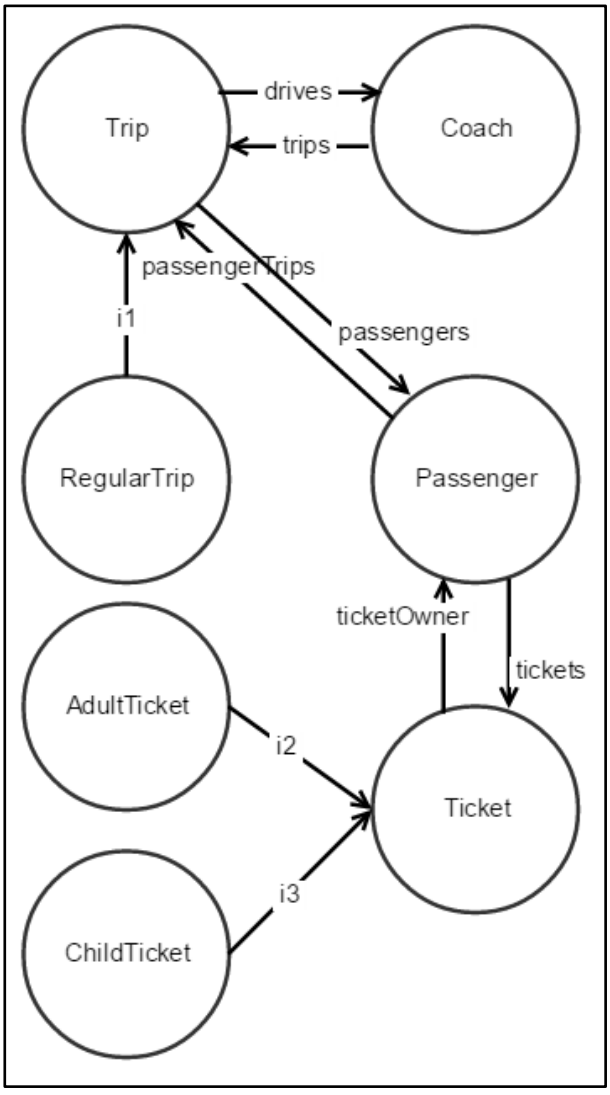

Figure 1: Resulting graph for the CDM after applying the model slicing from a class selection ={RegularTrip, Coach, Passenger, AdultTicket, ChildTicket}

We must take into consideration how to manage let expressions in invariants. With a let expression, variables can be defined and later used within the constraint body. A let expression has two parts, a "let" expression, and an "in" expression. The first one enables the definition of a variable and initializing it, while the second part is the body of the constraint that uses the previously defined variable. In our method, we need to manage how this variable is represented in the instance tree. Our solution when a let expression is used, is to modify the AST tree for the OCL expression before the whole process begins. Every time the variable defined by the "let" is used in the "in" part, instead of only including the variable name in the AST, we insert the full expression that initializes it. When, later, the AST is traversed to generate the Instance Tree and it reaches a "let" node, the "let" branch that defines the initialization of the variable is not traversed. Since we have inserted that expression already into the "in" branch that defines the body of the constraint whenever the variable is used, we can just traverse that branch generating the proper Instance Nodes. This simplifies the method while respecting the semantics of the let expression.

The rest of AST node types just give their parent access to the provisional instance trees generated by their child nodes. The whole generation of the final instance tree is a process that always terminates, since the AST is finite, and the traversal of each AST node is done once in post-order. After traversing each AST node, a single provisional instance tree is generated by it. Since the traversing of the AST is in post-order, when reaching a node with children, they have already been processed; each one having generated a provisional instance tree. After processing a parent AST node, all the provisional Instance trees produced by its children will be merged according to the specific type of AST node, as described above. The merge function always produces a single instance tree, so the result after processing an AST with children is also a single Instance tree. In all cases, the complete traversal of all the AST nodes will finish at its root node, where the result is a single instance tree, with a root instance node that will be the '*self*' variable with the context of the constraint as its class. The process of traversing the AST is easily implementable in any object-oriented programming language by using the visitor pattern, which can be used to execute different logic depending on the AST node type reached through the traversal.

7. Description and Formalization of the Model Slicing

7.1. Method Description

For the slicing process, we need to define the slicing criterion that will define the core elements that must be present on the slice. After that, the second step of the slicing process is recursively inferring what other elements of the original model must also be added to the slice.

The designer will select, from the GDM, the classes that will be required for the client. From this information, a first CDM iteration will be created containing that initial set of classes, and the relationships that exist between those classes. After this, the second step of the process will iteratively increment the CDM depending on the type of relationships between those classes on the GDM and the classes not present in the selection made by the designer.

- **Association, aggregation or uses relationships:** When, on the GDM, two classes hold any of these types of relationships, but, on the CDM, only one of both ends is present, the class present on the CDM will not contain that relationship. If the designer does not select a certain class on its initial slice, it means that it is not needed, so all relationships connected to those classes are lost, regardless of the navigability defined for it.
- **Inheritance relationships:** A parent class does not require its child classes to make sense by themselves, but a child class is partly defined by its parent and depends on it. Whenever the CDM contains a class that, on the GDM, is a child in an inheritance relationship where the parent is not present on the CDM, that class and its inheritance relationship will be automatically included to the CDM. This same principle applies to interface relationships. We do not take into consideration the opposite approach, getting the child classes of a parent class, even in the case of abstract classes. A parent can have many child relationships, and the designer may not require most of them on the client. It is the designer's responsibility to select the specific classes required. The slicing process only adds the elements required for that initial selection to make sense.
- **Composition relationships:** A composition defines a strong relationship that implies a tight bond between both classes and their lifecycles. A component class may be independent of its container class, but a container class requires its components to make sense. If this were not the case, the relationship should probably be modelled by using an aggregation or association. When the GDM holds a composition relationship, but, on the CDM, only the container is present, those classes and relationships are also added to the CDM.
- **Methods:** As we mentioned in Chapter 5, our formalization will not include the description of the methods of UML classes to provide a formalization that is cleaner and easier to understand. But informally, it also should be mentioned that in the cases were a CDM class contains a method that, in its signature, refers to classes only present on the GDM, that method would be deleted from the class. We consider that if those classes are not present on the CDM, that method will not be required on the client. It is important to notice that this approach takes only into account the dependencies made evident on the class diagram through the method signature. There could be subtler dependencies involved, not easy to describe unambiguously for each method using only class diagrams. That would require another whole level of analysis.

7.2. Formalization of the Model Slicing Process

The slicing criterion will be a selection of the classes of the original GDM that must be present on the client ClassSelection ⊆ Vcd. The slice will be another graph **Gcd' = (Vcd', Ecd')** where we will define how the Vcd' sets of classes and the Ecd' relationship are generated in a recursive way, with a basic clause, inductive clauses and an external clause.

The basic clause means that the class selection is a subset of the set of classes from the CDM diagram **ClassSelection $\subseteq V_{cd}' \subseteq V_{cd}$**. The attributes of the classes in V_{cd}' are the same as the ones in V_{cd}. In the formal definition, if two classes have the same name, they also have the same attributes:

$$\forall c \in V_{cd}, \forall c' \in V_{cd}',\ c[n_c] = c'[n_c] \rightarrow A_c \in c \ = \ A_c' \in c' \qquad (6)$$

E' consists of relationships that are equal to the association and inheritance relationships which both, origin and destination classes, are present in V':

$$\forall a \in A_{cd} / a[o_e] \in V_{cd}' \wedge a[d_e] \in V_{cd}' \rightarrow \exists a' \in A_{cd}' / a' = a$$
$$\wedge$$
$$\forall i \in I_{cd} / i[c_i] \in V_{cd}' \wedge i[p_i] \in V_{cd}' \rightarrow \exists_i' \in I_{cd}' / i' = i \qquad (7)$$

Once we have the basic elements of the slice, the inductive clauses describe how the relationships defined in E_{cd} add new elements to G_{cd}'. For

each element present in V_{cd}' that is the child part of an inheritance relationship in E_{cd}, its relationship and parent class will also be part of V_{cd}':

$$\forall i \in I_{cd}, i[c_i] \in V_{cd}' \rightarrow i[p_i] \in V_{cd}' \wedge i \in I_{cd}' \quad (8)$$

In the same way, for each element present in V_{cd}' that is the origin of a composition relationship in E_{cd}, the relationship and the destination class of it will also be part of V':

$$\forall c \in V_{cd}, \forall a \in A_{cd} / a[o_a] = c \wedge a[c_a] = true, \quad a[o_a] \in V_{cd}' \rightarrow a[d_a] \in V_{cd}' \wedge c \in A_{cd}' \quad (9)$$

The final external clause states that nothing is in G_{cd}' unless it is obtained from the basic and inductive clauses.

If we wanted a client with a class selection ClassSelection = {RegularTrip, Coach, Passenger, AdultTicket, ChildTicket} the result would be as shown in Figure 7. The parent classes Trip and Ticket would be added during the process iterations, as well as their relationships to the rest of the classes.

8. Description and Formalization of the Constraint Classification

8.1. Method Description

Once the CDM and GDM are defined, we need a way of classifying the constraints in order to help the designer to understand which of them may be checked on the client. The classification considers all three levels of server dependency based on the constraint features, the class diagram for the server and the resulting class diagram for the client produced after the model slicing process:

- **Server-independent:** All the elements needed to check attribute and object constraints are present in the instance itself, so those constraints are always server-independent as long as the class of the object involved exists on the CDM.
- **Potentially server-dependent:** Class constraints and domain constraints where all the classes involved are present on the CDM are potentially server-dependent. With only the information from the class diagrams and the constraint is not possible to know in advance how the object graph will be managed between client and server. The client may not always contain a copy of all the objects of a certain class present on the server required to evaluate the constraint. In those cases, evaluating those constraints on the client would require retrieving the missing objects from the server. Identifying constraints in this situation may help the developer to understand their situation, and decide how to manage them.
- **Server-dependent:** When a constraint present on the GDM involves instances from classes that are not present on the CDM, that constraint will never be evaluable on the client.

This classification is not only useful to automatically include the server-independent constraints onto the CDM, but it also helps the developer to understand the relationship between constraints and client, and analyze if there is a way to make the potentially server-dependent constraints viable on the client, or modify them to fit on the client.

8.2. Formal Description for Generating Explicit Constraints from Class Relationships

A class diagram contains many implicit constraints regarding the cardinality of the relationships between different classes. To be able to analyze and classify all constraints in a coherent way, all of them should be formalized in the same way. We present here a way of transforming the cardinalities of the class graph into the instance trees that represent those constraints.

The class graph has a A_{cd} set that contains every association and composition relationship. For each of the elements of that set, an instance tree will be created describing the cardinality constraint:

$$\forall a \in A_{cd}, \exists it \in IT \quad (10)$$

Each instance tree created for the *a* relationship will be generated based on the following rules. As described in Section 5, each *a* relationship is a tuple **$a = (n_a, o_a, d_a, l_a, u_a, c_a)$.** And each instance tree is a **$it=(n_{it}, v_{it})$** as described in Section 6. Every *it* element generated for each a_c element will be generated as follows. The constraint will be named after the relationship: **$n_{it} = n_a$**. The instance tree root node will be of the same class as the relationship origin class, and will only have a single child node of the same class as the relationship destination:

$$\forall a \in A_{cd} \rightarrow \exists v_{it} \in it, ch_{vi} \in v_{it}[CH_{vi}] / v_{it}[c_{vi}] = a[o_a] \wedge d_{ch}[c_{vi}] = a[d_a] \quad (11)$$

The single edge between the root and its child $ch_{vi}=(d_{ch}, n_{ch}, l_{ch}, u_{ch})$ will have the same name and cardinalities of the relationship, where: **$n_{ch}=n_a$ $l_{ch}=l_a$ $u_{ch}=u_a$.**

By having all the constraints in the same notation, the metrics and classification criteria that will be described in the following sections will also consider the relationships between classes and their cardinality.

8.3. Formal Metrics: Minimum and Maximum Number of Instances, Classes and Attributes

Using instance trees to represent constraints has the benefit that the same notation holds information about both, the constraint and the cardinality constraints usually present on the class diagram. This allows performing operations over it to retrieve metrics about the minimum number of object instances necessary to evaluate the constraint, the number of different classes involved, or the attributes required. It also lets us know if evaluating this constraint requires always a limited number of instances, or if it may potentially operate over an infinite number of elements.

These metrics are required to be able to classify the constraints according to their complexity and their level of dependency upon the GDM. We will define some of the functions required to obtain these metrics.

First, we need to know, for a given constraint, what is the minimum number of instances it requires to evaluate its predicate. It is also important to know which is the maximum number of instances that can be potentially evaluated on that constraint.

The $minI(v_i)$ and $maxI(v_i)$ functions are recursive and receive an INSTANCENODE element $v_{it} \in IT$, and calculate the metric for its node and its child nodes. The result obtained for each child is multiplied by the cardinality (the lower bound on the minI function, the upper bound on the maxI function) of the edge connecting it to the parent node. By passing the IT root node to these functions, the function is called over it and all its children until the final result are obtained.

$$\min I(v_i) = |v_i[V_{vi}]| + \sum_{i=0 \text{ to } n} ch_{vii}[l_{ch}] \times \min I(ch_{vii}[d_{ch}]), \forall ch_{vi} \in v_i[CH_{vi}] \quad (12)$$

$$\max I(v_i) = |v_i[V_{vi}]| + \sum_{i=0 \text{ to } n} ch_{vii}[u_{ch}] \times \max I(ch_{vii}[d_{ch}]), \forall ch_{vi} \in v_i[CH_{vi}] \quad (13)$$

Another important metric is the number of different classes the instances belong to. This set is obtained by recursively visiting all the instance nodes in the tree and joining all its class types c_{vi}. This will result in a set with all the classes involved in the constraint.

$$c(v_i) = vi[c_{vi}] \cup (\cup_{i=0\ to\ n}\ c(d_{chi})), \forall d_{ch} \in ch_{vi}, \forall ch_{vi} \in v_i[CH_{vi}] \quad (14)$$

Finally, it is also useful to know the number of attributes involved in the constraint. To obtain the set of attributes we also visit each child of the instance tree recursively, and join all the A_{vi} attribute sets included in each node.

$$a(v_i) = vi[A_{vi}] \cup (\cup_{i=0\ to\ n}\ a(d_{chi})), \forall d_{ch} \in ch_{vi}, \forall ch_{vi} \in v_i[CH_{vi}] \quad (15)$$

It is important to note that while $minI(v_i)$ and $maxI(v_i)$ return an integer as a value, $c(v_i)$ and $a(v_i)$ return a set containing classes or attributes in each case. These basic metrics about the constraint provide a reference to the complexity and nature of each one of them, and understanding the number of instances, classes and attributes involved in a constraint, it is possible to determine if a class model will be able to produce object graphs evaluable by that constraint. This would allow us to identify the server constraints that can be tested on the client.

The resulting constraint metrics described on the example GDM are shown in Figure 8.

Constraint 1:
minI(Con1) = 1
maxI(Con1) = 1
c(Con1) = {Passenger}
a(Con1) = {Passenger:age}
Constraint 2:
minI(Con2) = 1
maxI(Con2) = 1
c(Con2) = {Coach}
a(Con2) = {Coach:type, Coach:noOfSeats}
Constraint 3:
minI(Con3) = 1
maxI(Con3) = *
c(Con3) = {Ticket}
a(Con3) = {Ticket:number}
Constraint 4:
minI(Con4) = 2
maxI(Con4) = 2
c(Con4) = {ChildTicket, Passenger}
a(Con4) = {Passenger:age}
Constraint 5:
minI(Con5) = 1
maxI(Con5) = *
c(Con5) = {Passenger, Ticket}
a(Con5) = {Passenger:age}
Constraint 6:
minI(Con5) = 3
maxI(Con5) = *
c(Con5) = {Trip, Coach, Passenger}
a(Con5) = {Coach:noOfSeats}
Constraint 7:
minI(Con6) = 2
maxI(Con6) = *
c(Con6) = {BookingOffice, Coach, VendingMachine, Ticket}
a(Con6) = {Coach:noOfSeats}

Figure 8: Metrics for the constraints described in Figure 4

8.4. Formalization of Constraint Classification

As we stated before, we will classify constraints according to two different criteria. The first one is a quantitative classification that will assign a type to each constraint according to the variety of elements that it constraints. We described the four types informally in Section 2; attribute, object, class and domain constraints. We will proceed now to describe how to use the metrics extracted from a constraint to classify them.

- **Attribute constraints:** A constraint is an attribute constraint if it only affects a single instance and a single attribute. This means that both the minimum and maximum number of instances must be one, and the size of the attribute set for the graph must also be one. Checking the number of classes is not required, since there is a single instance.

$$Attribute: (minI(IT[v_i]) = 1) \wedge (maxI(IT[v_i]) = 1) \wedge (|\ a(IT[v_i])| = 1) \quad (16)$$

- **Object constraints:** Object constraints also affect a single instance, but they require the evaluation of several attributes. Therefore, the only difference from attribute constraints is in the size of the attribute set.

$$Object: (minI(IT[v_i]) = 1) \wedge (maxI(IT[v_i]) = 1) \wedge (|\ a(IT[v_i])| > 1) \quad (17)$$

- **Class constraints:** Class constraints affect several instances, but all belonging to the same class, so it is required to check the size of the classes set to verify that is 1.

$$Class: (maxI(IT[v_i]) > 1) \wedge (|\ c(IT[v_i])| = 1) \quad (18)$$

- **Domain constraints:** Finally, domain constraints are those that affect more than one class. In this case, the only thing to verify is the class set size.

$$Object: |\ c(IT[v_i])| > 1 \quad (19)$$

The second classification requires knowing both the CDM and GDM, to classify it according to their server dependency level.

- **Server-independent:** Any constraint that is an attribute or object constraint and which class is present on the client is always independent.

$$ServerIndependent: (minI(IT[v_i]) = 1) \wedge (maxI(IT[v_i]) = 1) \wedge (\forall c \in c(IT[v_i]), c \in Vcd') \quad (20)$$

- **Potentially server-dependent:** When a constraint requires several instances, and the classes of all of them belong to the CDM, it is a partially dependent constraint.

$$PotentiallyServerDependent: (maxI(IT[v_i]) > 1) \wedge (\forall c \in c(IT[v_i]) c \in Vcd') \quad (21)$$

- **Server-dependent:** When a constraint refers to elements that are only present on the server, they are never evaluable on the client.

$$ServerDependent: \exists i \in c(IT[v_i]) / i \notin V_{cd}' \quad (22)$$

Using these criteria, the constraints for our example would be classified as follows in Figure 9:

Constraint 1: attribute, server-independent
Constraint 2: object, server-independent
Constraint 3: class, potentially server-dependent
Constraint 4: domain, potentially server-dependent
Constraint 5: domain, potentially server-dependent
Constraint 6: domain, potentially server-dependent
Constraint 7: domain, server-dependent

Figure 9: Final classification of the GDM constraints.

This results in: constraints 1 and 2 being completely verifiable within the CDM; constraints 3, 4 and 5 are potentially server-dependent for this given CDM. Depending on how the data is managed in the system as a whole, these constraints could be checked on the client without further communication with the server, or require retrieving missing information from it before doing the verification. If the client stores a complete and updated copy of the part of the GDM object graph that is relevant to the CDM, then the verification can be done without further communication with the server. If the client works requesting information on demand, this could be different. In any case, this classification aids the designer to understand the challenges that a constraint poses, and may also help them to consider how data should flow between client and server.

Finally, constraint 6 is server-dependent. Although it refers to classes present on the CDM, it also refers to *VendingMachine* and *BookingOffice* which are not. This constraint should belong to the GDM only.

9. Testing the Method: Ecore Prototype

To verify the validity of this method, we developed a prototype in Java based on the Eclipse Modeling Framework[8], where we used the Ecore metamodel to define and manipulate the domain models. As input elements, the prototype receives the class model from the server, its OCL constraints, and the list of classes that are relevant to the client. The output is an Ecore file with the CDM, the OCL constraints that are completely independent to be used on the client, and a text file containing the metrics and classification extracted from each constraint, so that the designer can use this information to decide how the potentially dependent constraints may fit on the client, or which modifications can be done to adapt some of them to the client.

The tool works in 5 stages. First, it generates the implicit constraints, as described in section 8.2, adding those newly generated OCL constraints to the ones provided by the designer. Then it analyzes all the constraints to extract their metrics and make the first classification (attribute, object, class or domain). After this, the CDM is generated, and its information is used in combination with the metrics previously extracted to classify them according to their level of dependency for that given CDM, adding the completely independent ones to an OCL file to be used directly with that class diagram. This file can be later updated by the designer after analyzing the potentially dependent constraints that have been temporarily left aside. The tool also generates documentation automatically, including the AST for each OCL constraint, as well as its instance tree.

This tool serves as a proof of concept showing that the application of these formal methods is possible with existing and well-known tools. It also illustrates how a full domain model can be created easily by a designer using as an input only a list with the client-relevant classes. Since this is only a prototype, it is still limited in its functionality. Nevertheless, the metrics it captures to generate the CDM will allow us to expand its functionality in the future and provide a more solid support for client development. A more detailed description of this Java tool can be consulted in the original paper [25].

10. Current Method 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 presents challenges that are not addressed by our work. Working at the level of class diagrams also offers some limitations in certain cases. In modern distributed applications, RESTful web services [27] 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 [6]. It is also important to notice 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 simplifies the designer‘s responsibilities, relieving them from the need to model 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. Still, 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. As mentioned in Section 7, there may be specific domain models where further coupling exists for that specific domain, or other dependencies not easily modelled by means of class diagrams only. In addition to this, it is possible for the designer to inadvertently omit some classes required for the client they intend to model. And while the solution would be consistent with their selection, it may not be the model they intended to build, requiring them to make the selection again to generate a new model.

We believe this type of limitations would be very interesting to address, but the level of analysis to act on such cases is out of the scope of the currently proposed method. We believe this work is an interesting step towards providing a foundation for easing the development of robust rich clients.

11. Conclusion and Future Work

Designing a rich client able to maintain server consistency while being as independent as possible is a challenging task. It requires a careful study of the domain model and the constraints present on the server and a thoughtful adaptation. If done right, this allows the client to evaluate many of the constraints defined locally for the server, providing a better user experience. However, a task involving several interdependent diagrams,

[8] https://eclipse.org/modeling/emf/

and logical predicates that must be consistent on both sides is a tedious and error-prone task, that gets even harder as the development cycle evolves and server changes affect the client.

Once the designer knows which server elements (i.e. classes) are fundamental for the client domain model, the rest of the information required is already present on the global domain model. Instead of burdening the designer with the task of analyzing how to adapt these classes and constraints to the client, we propose to use the existing information from the global domain model to generate it automatically.

We provide a formal approach to describe our method, so that it can be implemented through any existing technology. We chose a graph-based approach that fits into UML class diagrams and OCL constraints. And since it is general enough, other diagram based standards can also be adapted. Our formal descriptions include a graph-based approach for class diagrams, an instance-tree-based approach for constraints, and a way of automatically generating instance trees from the abstract syntax tree of an OCL expression. We also describe formally how to calculate different metrics over the constraints, based on instance trees and logical predicates, and use them to classify them. We describe a method that successfully generates a valid client domain model based on the global domain model after the designer selects the client-required classes . These metrics are used to detect all the constraints that can be evaluated on the client, as well as the ones that should not be evaluated on the client. The constraints that can be evaluated on the client but may require communication with the server are detected, and metrics about the elements affected by them are provided. We have managed to put these formal proposals into practice through a Java tool based on the Ecore standard.

In addition to addressing the limitations described in the previous section, this work is open to many possible improvements. We are able to successfully detect potentially server-dependent constraints, but right now it is up to the designer to decide how to deal with them. One possible approach towards these constraints is breaking down a constraint into several ones that affect fewer elements, so that at least part of its predicate can be checked locally on the client.

Another possibility where communication is required, is to make a design between client and server where 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 pre-calculating 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.

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, further automated decisions could be made. The domain model elements required for these tasks could be automatically generated, which would further improve our approach and relieve the designer from even more decisions.

Since this method is particularly focused on detecting the dependency level between client and server elements, this approach could also be adapted to analyze how to create proxies that manage the communication between client and server appropriately, and aid in maintaining client independency while easing the evaluation of constraints.

Our current proposal helps the designer to generate rich clients consistent with the server automatically, making the design effort more agile, robust and adaptable to changes during development. It also avoids the common problems associated with designing an architecture where many of its elements are interdependent, and provides metrics and information about how the different elements are related so that the designer can make better informed decisions.

Acknowledgements

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 (grantGRUPIN14-100).

References

[1] Arendt, T. et al. 2014. From core OCL invariants to nested graph constraints. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 8571 LNCS, (2014), 97–112.

[2] Bottoni, P. et al. 2010. A language-independent and formal approach to pattern-based modelling with support for composition and analysis. *Information and Software Technology*. 52, 8 (2010), 821–844.

[3] Bottoni, P. et al. 2001. A Visualization of OCL using Collaborations. *≪UML≫ 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools*. 2185, (2001), 257–271.

[4] Bozzon, A. et al. 2006. Conceptual modeling and code generation for rich internet applications. *Proceedings of the 6th international conference on Web engineering ICWE 06* (2006), 353.

[5] Cabot, J. and Teniente, E. 2009. Incremental integrity checking of UML/OCL conceptual schemas. *Journal of Systems and Software*. 82, 9 (Sep. 2009), 1459–1478.

[6] Demuth, Birgit, H.H. Using UML/OCL Constraints for Relational Database Design. 6.

[7] Duhl, J. 2003. White paper: Rich internet applications. *Available at http://wwwmacromedia. comn/ …*. (2003).

[8] Hallé, S. and Villemaire, R. 2009. Browser-based enforcement of interface contracts in web applications with BeepBeep. *Computer Aided Verification*. (2009), 648–653.

[9] Heidegger, P. and Thiemann, P. 2012. JSConTest: Contract-Driven Testing and Path Effect Inference for JavaScript. *The Journal of Object Technology*. 11, 1 (2012), 6:1.

[10] Howse, J. et al. 2001. Spider Diagrams: A Diagrammatic Reasoning System. *Journal of Visual Languages & Computing*. 12, 3 (2001), 299–324.

[11] Hunt, G. and Scott, M. 1999. The Coign automatic distributed partitioning system. *OSDI '99 Proceedings of the third symposium on Operating systems design and implementation* (1999), 187–200.

[12] Kagdi, H. et al. 2005. Context-free slicing of UML class models. *IEEE International Conference on Software Maintenance, ICSM*. 2005, (2005), 635–638.

[13] Kent, S. 1997. Constraint Diagrams: Visualizing Invariants in Object-Oriented Models. *ACM SIGPLAN Notices*. 32, 10 (1997), 327–341.

[14] Kollmann, R. and Gogolla, M. 2002. Metric-based selective representation of UML diagrams. *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*. (2002), 89–98.

[15] Kuhlmann, M. and Gogolla, M. 2012. From UML and OCL to relational logic and back. *15th International Conference on Model Driven Engineering Languages and Systems, MODELS 2012*. 7590 LNCS, (2012), 415–431.

[16] Lallchandani, J.T. et al. 2011. for UML Architectural Models. *Most*. 37, 6 (2011), 737–771.

[17] Leff, A. and Rayfield, J. 2006. Programming model alternatives for disconnected business applications. *Internet Computing, IEEE*. June (2006), 50–57.

[18] Liang, Z.L.Z. and Jianling, S.J.S. 2009. A field-oriented approach to web form validation for Database-Isolated Rule. *2009 IEEE International Conference on Systems Man and Cybernetics* (2009), 4607–4612.

[19] Louwsma, J. et al. 2007. Specifying and Implementing Constraints in GIS—with Examples from a Geo-Virtual Reality System. *GeoInformatica*. 10, 4 (Jan. 2007), 531–550.

[20] McUmber, W. and Cheng, B. 2001. A general framework for formalizing UML with formal languages. *... of the 23rd international conference on ...*. (2001), 433–442.

[21] Mesbah, A. and Van Deursen, A. 2006. An Architectural Style for Ajax. *2007 Working IEEEIFIP Conference on Software Architecture WICSA07* (2006), 9–9.

[22] Mineau, G. et al. 2000. Conceptual Modeling Using Conceptual Graphs. *Krdb*. JULY 2000 (2000), 73–86.

[23] Peng Liang, Annya Romanczuk Réquilé, J.-C.R. 2003. *A translation of UML components into Formal S pecifications*.

[24] Preciado, J. and Linaje, M. 2007. Designing rich internet applications with web engineering methodologies. *Web Site Evolution*. (2007), 23–30.

[25] Quintela-Pumares, M. et al. 2014. Automatic Classification of Domain Constraints for Rich Client Development. *ICSEA 2014, The Ninth International Conference on Software Engineering Advances* (Nice, France, 2014), 570 to 576.

[26] Rensink, A. 2004. Representing first-order logic using graphs. *Graph Transformations: Second International Conference, ICGT 2004, Rome, Italy, September 28--October 1, 2004. Proceedings*. (2004), 319--335.

[27] Rodriguez, A. 2008. Restful web services: The basics. *Online article in IBM DeveloperWorks Technical Library*. November (2008), 1–11.

[28] Schmidt, K. et al. 2009. Gaining reactivity for rich internet applications by introducing client-side complex event processing and declarative rules. *AAAI 2009 Spring Symposium: Intelligent Event Processing* (2009), 67–72.

[29] Shaikh, A. et al. 2011. Evaluation of Tools and Slicing Techniques for Efficient Verification of UML/OCL Class Diagrams. *Advances in Software Engineering*. 2011, (2011), 5.

[30] Shaikh, A. et al. 2011. UOST: UML/OCL aggressive slicing technique for efficient verification of models. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 6598 LNCS, (2011), 173–192.

[31] Shaikh, A. et al. 2010. Verification-driven slicing of UML/OCL models. *Proceedings of the IEEE/ACM international conference on Automated software engineering - ASE '10*. November 2015 (2010), 185.

[32] Singh, R. 2013. Literature Analysis on Model based Slicing. *Internatonal journal of Computer Applications*. 70, 16 (2013), 45–51.

[33] Tilevich, E. and Smaragdakis, Y. 2002. J-orchestra: Automatic java application partitioning. *ECOOP '02 Proceedings of the 16th European Conference on Object-Oriented Programming* (2002), 178–204.

[34] Yang, F. et al. 2007. A unified platform for data driven web applications with automatic client-server partitioning. *Proceedings of the 16th international conference on World Wide Web - WWW '07* (New York, New York, USA, 2007), 341.

[35] Zhang, W.Z.W. 2010. 2-Tier Cloud Architecture with maximized RIA and SimpleDB via minimized REST. *Computer Engineering and Technology ICCET 2010 2nd International Conference on*. 6, (2010), V6-52-V6-56.

[36] 2004 - Marcel Kyas, Harald Fecher, Frank S. de Boer, Joost Jacob, Jozef Hooman, Mark van der Zwaag, Tamarah Arons, Hillel Kugler - Formalizing UML Models and OCL Constraints in PVS.pdf.