MDE 2.0:Pragmatic formal model verification and other challenges
HdR

Jordi Cabot
Ecole des Mines de Nantes
jordi.cabot@mines-nantes.fr

June 5, 2012

Abstract

This document presents a synthesis of the research results conducted in the field of model-driven engineering (MDE) by the author. MDE is becoming one of the dominant software engineering paradigms in the industry. Similar to any other engineering discipline, MDE advocates for the rigorous use of (software) models (expressed as typed graphs) as the main artifacts in all software engineering activities: forward engineering, reverse engineering, software evolution, systems interoperability and so on. Adoption of MDE brings several benefits to software engineering, including improvements in the productivity and maintainability of the system. Unfortunately, industrial adoption of MDE faces some hurdles (e.g. in terms of scalability and quality of MDE solutions, integration with legacy systems, usability, ...) that must be overcomed before MDE becomes mainstream and the whole software community can benefit from it.

This report describes some of my current research work on MDE. Among the different research lines we will focus on the model verification contribution. The change of perspective in MDE implies that correctness of models (and model manipulation operations) becomes a key factor in the quality of the final software product which is now (semi)automatically generated from them. In general, the problem of ensuring software correctness is still considered to be a Grand Challenge for the software engineering community. At the model-level, research results for the quality evaluation of models are still preliminary. We will detail our pragmatic but formal model verification approach to overcome the limitations of the other works in the area. We refer to our techniques as pragmatic because they pursue the best trade-off between the completeness and the usability of the verification process.

Beyond this research line, the report also includes a blueprint for a research program in MDE to answer this verification grand challenge and the other MDE challenges believed to become key aspects in the next years in order to successfully integrate MDE in the daily practice of all software professionals and effectively change the way software development (and evolution, maintenance,...) is performed.

Contents

Chapter 1

Model-Driven Engineering

Model Driven Engineering (MDE) is becoming one of the most popular software engineering paradigms nowadays. Similar to any other engineering discipline, MDE advocates for the rigorous use of (software) models (expressed as typed graphs) as the main artifacts in all software engineering activities: forward engineering, reverse engineering, software evolution, systems interoperability and so on. Adoption of MDE has the potential to bring many benefits to software engineering, including improvements in the productivity and maintainability of the system, as reported by several studies (e.g. [81]). Initially promoted by the OMG (Object Management Group; one of the most important non-profit computer industry consortiums) as a model-driven development process under the name MDA [1] (Model Driven Architecture) [82], MDE has now considerably broadened its scope. The (scientific) community is now investigating the use of MDE on most software engineering activities involved in the construction, operation or maintenance of software-intensive systems.

Companies need to have a regular and homogeneous organization where different facets of a software system may be easily separated or combined (e.g. depending on the role of each team member at the stage of the development process). The basic assumption of MDE is that models and not the classical programming code is the right representation level for managing all these facets. When needed, code can be automatically generated from the models.

Thereofere, in MDE, models are considered as the unifying concept. The MDE community distinguishes three levels of models: (terminal) model, metamodel, and metametamodel. A terminal model is a (partial) representation of a system/domain that captures some of its characteristics (different models can provide different knowledge views on the domain and be combined later on to provide a global view). In MDE, we are interested in terminal models expressed

[1] MDA defines as a two-step model-driven development process. First platform-independent aspects of the system are modeled. After, these platform-independent models are adapted and refined taking into account the technical limitations of the final implementation platform, producing as a result a set of platform-specific models. From these models, code can be automatically generated.

in precise modeling languages. The abstract syntax of a language, when expressed itself as a model, is called a metamodel. A complete language definition is given by an abstract syntax (a metamodel), one or more concrete syntaxes (the graphical or textual syntaxes that designers use to express models in that language) plus one or more definitions of its semantics. The relation between a model expressed in a language and the metamodel of that language is called conformsTo [2]. Metamodels are in turn expressed in a specific modeling language called metamodeling language. Similar to the model/metamodel relationship, the abstract syntax of a metamodeling language is called a metametamodel, and metamodels defined using a given metamodeling language must conform to its metametamodel [3]. Terminal models, metamodels, and metametamodel form a three-level architecture with levels respectively named M1, M2, and M3. A more formal definition of these concepts is provided in [18, 66]. These MDE principles may be implemented in several standards. For example, OMG proposes a standard metametamodel called Meta Object Facility (MOF) while the most popular example of metamodel in the context of OMG standards is the Unified Modeling Language (UML) metamodel [84].

The other key element in MDE is the concept of model manipulation, usually implemented by means of model transformation operations that, taking one or more models as input, generate one or more models as output (where input and output models not necessarily conform to the same metamodel). More specifically, a model transformation Mt defines the production of a model Mb from a model Ma. When the source and target metamodels are identical (MMa = MMb), we say that the transformation is endogenous. When this is not the case (MMa $\neq$ MMb), we say that the transformation is exogenous. An example of an endogenous transformation is a UML refactoring that transforms public class attributes into private attributes, while adding accessor methods for each transformed attribute. Many other operations may be considered as transformations as well. For example, verifications or measurements on a model can be expressed as transformations. One can see then why large libraries of reusable modeling artifacts (mainly metamodels and transformations) will be needed. Another important idea is the fact that a model transformation is itself a model [15]. This means that the transformation program Mt can be expressed as a model, and as such conforms to a metamodel MMt. This allows a homogeneous treatment of all kinds of terminal models, including transformations. Mt can then be manipulated using the same existing MDE techniques already developed for other kinds of models. For instance, it is possible to apply a model transformation Mt' to manipulate Mt models. In that case, we say that Mt' is a higher order transformation (HOT) [102], i.e. a transformation taking other transformations (expressed as transformation models) as input or/and producing other transformations as output.

[2] This relationship is equivalent to the program - grammar relationship in the programming community. Like models, programs written in one language must conform to the grammar rules (i.e. the metamodel) of that language.

[3] A metametamodel defines the possible characteristics of a family of modeling languages in the same way as EBNF or BNF can be used to represent certain types of grammars.

As MDE developed, it became apparent that this was a branch of language engineering. In particular, MDE offers an improved way to develop DSLs (Domain-Specific Languages). DSLs are programming or modeling languages that are tailored to solve specific kinds of problems, in contrast with General Purpose Languages (GPLs) that aim to handle any kind of problem. Java is an example of a programming GPL and UML an example of a modeling GPL. DSLs are already widely used for certain kinds of programming; probably the best-known example is SQL, a language specifically designed for the manipulation of relational data in databases. The main benefit of DSLs is that they allow everybody to write programs/models using the concepts that actually make sense to their domain or to the problem they are trying to solve (for instance Matlab has matrices and lets the user express operations on them, Excel has cells, relations between cells, formulas and allows the user to express simple computations in a visual declarative style, etc.). As well as making domain code programmers more productive, DSLs also tend to offer greater optimization opportunities. Programs written with these DSLs may be independent of the specific hardware they will eventually run on. Similar benefits are obtained when using modeling DSLs. In MDE, new DSLs can be easily specified by using the metamodel concept to define their abstract syntax. Models specified with those DSLs can then be manipulated by means of model transformations.

MDE has reached a maturity level where core tools and techniques for defining and manipulating all kinds of models are widely available. Nevertheless, the widespread use of MDE is raising new challenges that may impair the increasing adoption of MDE in practice, limiting the benefits that MDE can bring to software engineering. More research is needed to guarantee that MDE continues its progression both at the scientific and industrial level. Next chapter briefly summarize my research results in MDE achieved so far, distinguishing those part of my thesis work from those belonging to separate research lines (for full details please check the curriculum vitae attached to this document). Then, chapter 3 singles out one of these lines (model verification, a key issue to make MDE mainstream) and describes it in detail, listing all the papers published, their summary and the relationship between them. Finally, last chapter presents the major challenges that MDE will face in the next years and sketches the research program that we plan to pursue in the future in order to address them.

Chapter 2

My main contributions to MDE

We describe herein the main MDE research lines that I have been working on during my career. For each line we provide a short description and the most important publications. The initial work focused on modeling was mainly developed during my period as PhD student and, afterwards, lecturer at the Technical University of Catalonia and Open University of Catalonia in Spain. The Social MDE line was developed during my postdoc stay at the University of Toronto while most of my work on model management, transformations and verification started once I joined the Ecole des Mines de Nantes. For a full list of research lines and publications see http://jordicabot.com/research.html or the attached curriculum vitae.

2.1 PhD Thesis - Incremental evaluation of OCL constraints

Integrity constraints play a fundamental role in the definition of conceptual schemas (CSs) of information systems. An integrity constraint defines a condition that must be satisfied in each state of the information base (IB), e.g. a relational database, a NoSQL repository,.... Hence, the information system must guarantee that the state of the IB is always consistent with respect to the integrity constraints of the CS. This process is known as integrity checking. Unfortunately, current methods and tools do not provide adequate integrity checking mechanisms since most of them only admit some predefined types of constraints. Moreover, the few ones supporting a full expressivity in the constraint definition language present a lack of efficiency regarding the verification of the IB.

In the thesis, we proposed a new method to deal with the incremental evaluation of the integrity constraints defined in UML/OCL CS. We say that our

method is incremental since it adapts some of the ideas of the well-known methods developed for incremental integrity checking in deductive and relational databases. The main goal of these incremental methods is to consider as few entities of the IB as possible during the evaluation of an integrity constraint. This is achieved in general by reasoning from the structural events that modify the contents of the IB. Our method is fully automatic and ensures an incremental evaluation of the integrity constraints regardless their concrete syntactic definition. The main feature of our method is that it works at the conceptual level. That is, the result of our method is a standard CS. Thus, the method is not technology-dependent and, in contrast with previous approaches, our results can be used regardless the final technology platform selected to implement the CS. In fact, any code-generation method or tool able to generate code from a CS could be enhanced with our method to automatically generate incremental constraints, with only minor adaptations. Moreover, the efficiency of the generated constraints is comparable to the efficiency obtained by existing methods for relational and deductive databases.

A summary of the results of the thesis can be read here [35].

2.2 Definition of new modelling abstractions

A key element in the success of a modeling language is to provide modeling abstractions that allow modelers to represent in an easy and precise way the reality of the domain.

One example is the proposal of a new modeling pattern for the role concept. Roles are meant to capture dynamic and temporal aspects of real-world objects. The role concept has been used with many semantic meanings: dynamic class, aspect, perspective, interface or mode. In this paper [33] we identified common semantics of different role models found in the literature and presented a set of conceptual modelling patterns for the role concept that include both the static and dynamic aspects of roles. In particular, we proposed the Role as Entity Types conceptual modelling pattern to deal with the full role semantics.

A second relevant example is the representation of temporal information. The UML is a non-temporal conceptual modeling language. Conceptual schemas in the UML assume that the information base contains the current instances of entity and relationship types. For many information systems, the above assumption is acceptable. However, there are some information systems for which that assumption is a severe limitation. This happens when the functions of the information system require the knowledge of past states of the information base. In this paper [31] we extended the UML to define a set of temporal features of entity and relationship types, and to provide notational devices to refer to any past state of the information base. Using this extension, a designer may use the UML/OCL as if it were a temporal conceptual modeling language. We also presented a method for the transformation of a conceptual schema in this extended language into a conventional one.

2.3 Code-generation techniques

Despite what many model-driven tools advertise, we are still far away from being able to completely generate the full implementation of a software application from complex software models. By complex we mean models including business rules and business processes for instance or stating the requirement of the system using a declarative style (the preferred alternative [111]).

Relevant examples of my work in the area are:

- An initial (and automatic) generation of the dynamic aspects of the specified software application. Given an initial class diagram, our method automatically generates the OCL contracts of a set of operations that describe how users can modify and evolve the data managed by the application. This set is complete (all data can be modified through the operations) and executable. See [2, 3] for details
- A heuristic-based transformation to translate declarative operation specifications (i.e. operations specified by means of a set of OCL pre and postconditions) into equivalent imperative specifications (i.e. operations where the set of actions or structural events that will be issued when executing the operation are explicitly defined). The main problem we must face when performing this transformation is the ambiguity of declarative specifications. The same declarative specification can be translated into many different imperative ones. The heuristics help us to determine which alternative is the one that, most probably, the designer prefers. See [25] for details.
- In [21], a method to integrate business processes and structural aspects of the same domain is provided. Thanks to this integration, existing code-generation techniques for static models can automatically generate software systems that satisfy the process constraints.

2.4 Social aspects of MDE

Beyond the purely technical aspects, introduction of MDE in any company disrupts the development team (new roles are required, new skills, new dependencies between members,...). As with any other technology we have to make sure that adoption of MDE is done in the right way.

Together with Eric Yu (University of Toronto) have proposed the application of social modelling techniques to define the structure of a development team and study how this structure maps the requirements of a model-driven development process. The idea is to see the socio-technical congruence of the team wrt to the new technology and identify its main shortcomings. We have generalized the results to use social modeling as a useful way to understand the requirements of any development process (or parts of it) before integrating it into the daily practice of a software company, e.g.see [48, 49].

2.5 Model Management and Transformation

Since my integration in the AtlanMod team[1] I have collaborated in several new improvements for the ATL model-to-model transformation language [64]. For instance, we have improved the support for Higher-order transformations [101] (i.e. transformations that take as input and/or output another model transformation), improved the support for refinement/refactoring transformations (i.e. transformations that perform only small changes on the source model and that can benefit from an in-place execution strategy) and we are working towards the definition of complete ATL compilers able to execute tranformations in an incremental mode (changes on the source model do not trigger the full transformation execution but only those parts that are relevant for the subset of model changed) or in a lazy mode [103] (transformation is executed on-demand to produce only those parts of the target model the user wants to read).

I have also participated in the current team work on model management. The achievements on model transformation helped to identify quite early the necessity of providing additional model representation and manipulation techniques that could be grouped in a toolbox for MDE. That was the beginning of the AmmA toolbox (AmmA stands for AtlanMod Model Management Architecture). As part of the toolbox we can find a model weaving component (that allows defining relationships between models at a high abstraction level) and the megamodeling concept among others. A megamodel is a specific kind of terminal model whose elements represent models themselves as well as the existing relationships between these models. Therefore, designers can create a megamodel to store useful references to all modeling artifacts of the development project, including all the metadata required to understand the relationships between them. Recently, we have added the possibility of writing modeling scripts (MoScript [67]) that can be executed in batch mode to automate the manipulation of all components in a megamodel.

Some of these topics are still work in progress and as such also mentioned in the research program at the end.

2.6 Model verification

Development of new algorithms that perform model verification on UML/OCL class diagrams by translating the diagram into a Constraint Satisfaction Problem (CSP) such that iff the CSP has a solution the input model is correct. This approach has been adapted for the verification of several correctness properties for model transformations. Full details of this research line are developed in the next chapter.

[1] www.emn.fr/x-info/atlanmod

Chapter 3

Model Verification Research Line

This chapter elaborates on, in our opinion, the main challenge among the research lines introduced in the previous one. We motivate the problem, describe the research results achieved (and how they relate to each other in a coherent evolution) and explain them in some detail in order to give the reader a comprehensible view of this area. The interested reader can refer to the original publications listed herein for full details of our approach.

3.1 Overview

As we have said before, model-driven engineering (MDE) is becoming one of the dominant software engineering paradigms in the industry. MDE advocates for the use of software models and model transformations (to perform manipulations on models) as the main artifacts in all software engineering activities. This higher-level view of the software system (thanks to the emphasis on the use of models to describe the system) brings many benefits, including improvements in its productivity and maintainability, as reported by several studies.

This change of perspective implies that correctness of models (and model transformations, that can be, in fact, regarded as special kind of models [16]) becomes a key factor in the quality of the final software product. For instance, in MDE, code is no longer written from scratch but synthesized from models (semi-)automatically. Therefore, any defect in the model will propagate into defects in the code.

Although the problem of ensuring software quality has attracted much attention and research, it is still considered to be a Grand Challenge [63] for the software engineering community. This initiative considers that the degree of maturity achieved in software development, formal methods and verification is sufficient to achieve substantial contributions in the near future. We believe that this challenge must be adapted and extended to cover the verification of model-

ing notations commonly used in MDE approaches, like UML [84] and OCL [83] (textual language to annotate UML models with constraints, derivation rules and pre/postcondition contract expressions). It is essential to provide a set of tools and methods that help in the detection of defects at the model-level and smoothly integrates in existing MDE-based tool-chains without an excessive overhead.

Such method and tools do not exist yet. The best proof is the fact that so far no commercial modeling tool integrates any correctness analysis beyond purely syntactic checks to make sure the model is a valid instance of its metamodel. Characteristics of existing research approaches, like the lack of automation (methods based on theorem proving might require user assistance during proofs, e.g. HOL-OCL [23]), restrictions on the input notation (e.g. some methods work on a restricted subset of OCL (e.g. [91]) and some others do not support OCL at all (e.g. [11])), low efficiency (e.g. reasoning on UML class diagrams is EXP-complete [13] and thus, current tools do not scale-up well which makes efficiency a concern for most non-trivial models) and poor usability (verification tools are often disappointing from the point of view of a designer, one of the main reasons being that tools do not directly manipulate the input model but first translate it into a formal language (Alloy [4], CP [29], HOL [23], DL [107]) where the verification process takes place; therefore, in many cases a good knowledge of this underlying language may be required to operate effectively, e.g. while selecting adequate parameters, tuning the model for the analysis or interacting with the tool) have seriously impaired their usability in practice.

Learning from these previous attempts, we believe that in order to succeed in an MDE context any method for model verification should fulfill the following list of requirements:

- Understand the input notation used by the designer (e.g. UML/OCL), not a formal notation nor a subset of that notation. If an internal formal notation is used, it should be transparent to the designer.
- Analyze the designer's model *as is*, without requiring any type of manual annotation.
- Perform the analysis automatically and without requiring user interaction.
- Provide results in a format meaningful to the designer.
- Be efficient and scale up to support large real-life examples.
- Provide a tool implementation satisfying all the above requirements and that integrates seamlessly into the designer tool chain.

Unfortunately, model verification is a hard problem, undecidable in general. Therefore, to tackle this problem while at the same time fulfilling the previuos requirements to make sure our approach is usable in practice, we propose the notion of *pragmatic formal model verificaton*. Our formal verification approach tries to find a good trade-off between the completeness of the evaluation and the

usability (termination, efficiency, automation,...) of the verification process. As we will see, many of our works in this area follow a bounded verification strategy built on top of the constraint programming paradigm. In our approach, the model and the correctness property to verify are expressed as a constraint satisfaction problem (CSP) such that the CSP has a solution iff the model satisfies the property. In this case, the search space to evaluate when determining the correctness of the model is finite and therefore termination problems are avoided. The size of the search space will determine the efficiency of the process but at the same time may affect its completeness.

In the rest of this chapter, we introduce four of our main representative works in this area. First, our CSP-based approach is introduced as a solution to the problem of checking the correctness of UML class diagrams[1] annotated with OCL constraints. Among others, we focus on a fundamental semantic correctness notion in static models: *model satisfiability*. (Strong) Satisfiability consists in deciding whether it is possible to create a non-empty and finite instantiation of the model in such a way that all integrity constraints are satisfied. Clearly, an unsatisfiable model is useless since every time users try to create a new object, e.g. instantiating one class of the model, at least one of the integrity constraints will become violated. The importance of satisfiability comes from the ability to define many other correctness properties, such as liveliness, constraint redundancy, subsumption and so forth, in terms of the satisfiability problem. For example, a designer can check if an integrity constraint C is redundant by formulating a satisfiability problem where $\neg C$ replaces C in the model. If that model is satisfiable, it means it is possible to satisfy the remaining integrity constraints while violating C, so C is not a redundant constraint.

Later this CSP-based approach is extended to cover the verification of dynamic properties (like applicability, executability and determinism) for behavioural models expressed as a set of operations defined by means of UML/OCL pre/postcondition contracts.

Finally, we show how these core ideas can be applied also to the verification of other key component in MDE: model transformations. We first address the correctness of declarative model-to-model transformations (by analysing the implicit invariants stated by the transformation) and then the problem of correctness of graph transformations (largely used to express dynamic aspects of Domain-Specific Languags) by translating different rule semantics to OCL and then analyzing the resulting OCL expressions using the previous techniques. As you can see, there is a clear correspondence between the UML/OCL static properties analysis and the declarative model transformations and between the dynamic UML/OCL analysis and the more imperative graph-based transformations. In both cases, the latter is an adaptation and evolution of the former.

Other components of our pragmatic model verification approach (like the use of incremental evaluation techniques to optimize the reevaluation of existing models after changes, the idea of normalizing models prior to its verification to

[1] Even if we focus on UML, the same approach could be applied to the verification of Domain-Specific Languages; we focus on UML since it is by far the most used modeling language in both industry and academia

simplify manipulation operations on them or complementing verification with validation and testing approaches) are mentioned in the list of publications (next section) and/or in the future research lines (last chapter).

3.2 Published Research Results

Before entering into the technical details of this research line we present in this section the list of publications on this topic to give an idea of the results achieved so far. Publications are grouped according to the main topics described in the previous overview. Only publications related to the research work presented in this document are mentioned. Papers in an ERA (formerly CORE) conference with rank A and JCR-indexed journal publications are highlighted. When a journal publication on a specific topic has been published, preliminary conference papers on that same topic are omitted (again, see the CV for a complete list of papers). Next to each publication we indicate a very short summary of its contribution.

Note that next sections focus on a subset of these publications, in particular those based on the use of constraint programming as a verification tool since this is the core of our contribution in this area. Other papers exploring complementary approaches are mentioned in the list but not detailed in the rest of the chapter.

3.2.1 Verification of UML/OCL models

Initial focus of our reserach in the model verification area. After examining the different formalisms available, we choose constraint programming as a paradigm to use for the formal verification of UML/OCL models due to its bounded search characteristic that helps us to ensure termination of the verification process (even if this implies sacrifying the completeness of the results). Main papers on this line

- Jordi Cabot, Robert Clariso, Daniel Riera: Verifying UML/OCL Operation Contracts. 7th International Conference on Integrated Formal Methods (IFM 2009), LNCS 5423, pp. 40-55. Application of our CSP-based approach for the verification of dynamic aspects of UML/OCL models, more specifically focusing on the problem of verifying declarative operation specifications (expressed as OCL pre/postcondition contracts).

- Jordi Cabot, Robert Clariso: UML/OCL Verification in practice. 1st Int. Workshop on Challenges in Model-Driven Software Engineering (MoDELS'08). Position paper summarizing the main challenges in the area.

- Jordi Cabot, Robert Clariso, Daniel Riera: Verification of UML/OCL Class Diagrams Using Constraint Programming. MoDeVVA 2008 (Model Driven Engineering, Verification, and Validation: Integrating Verification

and Validation in MDE). Introduction to our CSP-based bounded verification method for the correctness evaluation of UML class diagrams with OCL constraints.

- Jordi Cabot, Robert Clariso, Daniel Riera: UMLtoCSP: a Tool for the Formal Verification of UML/OCL Models using Constraint Programming. ASE 2007, pp. 547-548. Presentation of our CSP-based tool for the verification of UML/OCL models.

With the popularity of Executable UML approaches (Executable UML defines a subset of UML that can be directly executable including a kind of pseudocode language that allows defining precise method behavioural specifications in an imperative manner), there is a growing need to be able to provide some verification techniques for Executable UML specifications. Following our goal of providing simple and efficient verification methods we have also proposed a dependency-analysis method to quicly evaluate basic correctness properties of these kinds of models. In this case we do not base our approach on constraint programming due to the imperative nature of these specifications.

- **Elena Planas, Jordi Cabot, Cristina Gmez: Verifying Action Semantics Specifications in UML Behavioral Models. The 21st International Conference on Advanced Information Systems (CAiSE 2009), LNCS 5565, pp. 125-140**. Lightweight verification approach to check the executability of imperative operation specifications based on a static analysis of the dependencies between the different atomic actions included in each operation.

3.2.2 Validation of UML/OCL models

To complement the verification research line we also did some relevant work on the validation aspect of the same kind of models. In contrast with verification (that tries to guarantee that *a model is right*), validation makes sure that what we are building *is the right model*. Only the client can assert this so validation techniques involve the client in the process, in this case by means of natural language techniques:

- **Jordi Cabot, Raquel Pau, Ruth Raventos: From UML/OCL to SBVR Specifications: a Challenging Transformation. Information Systems Elsevier Journal 35(4): 417-440 (2010)**. In this work, the models specified by the designer are reexpressed in natural language (using an intermediate SBVR representation) to allow stakeholders validate that the understanding of their requirements by the designers is correct.

3.2.3 Verification of model transformations

Extension of previous approaches for UML/OCL models to deal with the verification of model transformations. In fact, the last two papers correspond to

extensions of the papers in the first subsection. The first paper deals with a complementary challenge. Instead of focusing on the quality of models, it focuses on the quality of the instances (i.e. the data) of those models and makes sure that the system never evolves into an inconsistent state and does so by preventing the execution of the operations/rules that may induce this violation (instead of rolling back the system once we detect that a violation has been produced, clearly more inefficient).

- Jordi Cabot, Robert Clariso, Esther Guerra, Juan de Lara: Synthesis of OCL Pre-Conditions for Graph Transformation Rules, ICMT2010 - Intl. Conference on Model Transformation, LNCS 6142, pp. 45-60. Transformation engines must include a repair/rollback component to avoid inconsistent output models. Our work avoids the need for this component by adding preconditions that the input model must comply with to ensure that an inconsistent output model is never produced.

- **Jordi Cabot, Robert Clariso, Esther Guerra, Juan de Lara: A UML/OCL Framework for the Analsysis of Graph Transformation Rules. Software and Systems Modeling, vol 9, issue 3, pp. 335-357**. Analysis of refinement transformation rules by analyzing the pre/postconditions implicitly stated by the rule (OCL is used to make explicit these contracts deduced from the rules).

- **Jordi Cabot, Robert Clariso, Esther Guerra, Juan de Lara: Verification and Validation of Declarative Model-to-Model Transformations Through Invariants. Journal of Systems and Software 83(2): 283-302** . Verification of declarative model-to-model transormations by analyzing the invariants between the input and output (meta)models implicitly stated by the transformation.

3.2.4 Incremental evaluation

- **Jordi Cabot, Ernest Teniente: Incremental Integrity Checking of UML/OCL Conceptual Schemas. Journal of Systems and Software, vol 82, issue 9, pp. 1459-1478.**. Techniques to avoid a complete recomputation of a model after a set of changes on it (very frequent scenario in any iterative development process). These techniques are key to achieve an efficient model reevaluation, e.g. when checking if a previously correct model is still correct after updating some parts of it.

3.2.5 Model normalization

- **Jordi Cabot, Ernest Teniente: Transformation Techniques for OCL Constraints. Science of Computer Programming Journal, vol. 68/3, pp. 152-168**. Equivalence rules among OCL expressions to reduce OCL expressions to a kind of normal form. These simplifications facilitate the development of algorithms in charge of manipulating OCL

expressions thanks to reducing the number of expression combinations that must be considered.

3.3 Basic concepts of Constraint Programming

Some notions of constraint programming are needed in order to follow the technical content of next sections. This section gives a very short introduction of constraint programming.

Constraint Programming [5, 77] is a problem solving paradigm where the programming process is limited to the definition of the set of requirements (constraints). A *constraint solver* is in charge of finding a solution that satisfies the requirements.

Problems addressed by Constraint Programming are called *constraint satisfaction problems* (CSPs). A CSP is represented by the tuple CSP $= \langle V, D, C \rangle$ where V denotes the finite set of *variables* of the CSP, D the set of *domains*, one for each variable, and C the set of *constraints* over the variables. Typically, most constraints can be defined as equalities ($=$), disequalities ($\neq$) or inequalities ($<, >, \leq, \geq$) of arithmetic expressions over variables, or a boolean combination of such constraints, e.g. $(x = y) \vee (2x^2 \geq 0)$. A *solution* to a CSP is an assignment of values to variables that satisfies all constraints, with each value within the domain of the corresponding variable. A CSP that does not have solutions is called *unfeasible*.

The most traditional technique for finding solutions to a CSP is backtracking. A possible backtracking implementation called *labeling* orders variables according to some heuristic and attempts to assign values to variables in that order. If any constraint is violated by a partial solution, the solver reconsiders the last assignment, trying a new value in the domain and backtracking to previous variables if there are no more values available. This systematic search continues until a solution is found or all possible assignments have been considered. To ensure termination, the search space must be finite, thus, all variable domains must be finite.

The efficiency of the search process is largely improved by *constraint propagation* techniques: using information about the structure of constraints and the decisions taken so far in the search process, the unfeasible values in the domains of unassigned variables can be identified and avoided, pruning the search tree in this way. These techniques are an effective mechanism to reduce the search space and are implemented by default in most constraint solvers.

As an example, consider the simple CSP of Fig. 3.1. The CSP consists of two variables X and Y whose domain ranges from -100 to $+100$. There are three constraints: $X > 20$, $Y \leq 15$ and $X = Y$. For this example, constraint propagation techniques suffice to directly prove the unfeasibility of the CSP with neither instantiations nor backtracks required. First, the lower and upper bounds for the domains can be tightened by leaving only feasible values inside the domain. Furthermore, in the last step, the fact that the domains for X and Y are disjoint can be used to deduce that $(X = Y)$ is impossible: the set of

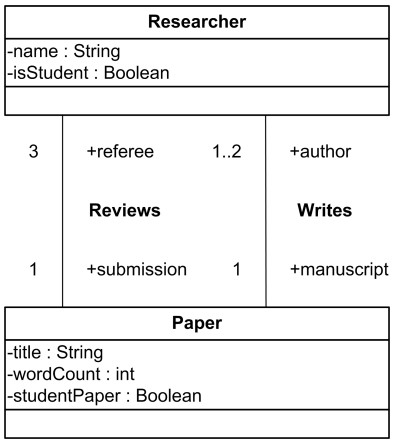
$$
\begin{aligned}
V &= \{\ X,\ Y\ \} \\
D &= \{\ domain(X) = [-100, +100], \\
&\quad\ domain(Y) = [-100, +100]\ \} \\
C &= \{\ X > 20,\ Y \leq 15,\ X = Y\ \} \\
&\quad\ \downarrow\ \textbf{Propagating constraint } (\mathbf{X > 20}) \\
D' &= \{\ domain(X) = [21, +100], \\
&\quad\ domain(Y) = [-100, +100]\ \} \\
&\quad\ \downarrow\ \textbf{Propagating constraint } (\mathbf{Y \leq 15}) \\
D'' &= \{\ domain(X) = [21, +100], \\
&\quad\ domain(Y) = [-100, 15]\ \} \\
&\quad\ \downarrow\ \textbf{Propagating constraint } (\mathbf{X = Y}) \\
D''' &= \{\ domain(X) = \emptyset, \\
&\quad\ domain(Y) = \emptyset\ \}
\end{aligned}
$$

Figure 3.1: Constraint propagation example

feasible values in the domain becomes empty ($\emptyset$), so we conclude that the CSP is unfeasible. Typically, constraint propagation is not that successful, but it is an effective mechanism to reduce the search space. Some types of arithmetic constraints, e.g. linear inequalities, have specialised numerical solvers to perform complex propagations among variables.

Without loss of generality, in this chapter we will describe CSPs using the syntax provided by the $\mathsf{ECL^iPS^e}$ Constraint Programming System [5, 100]. In $\mathsf{ECL^iPS^e}$, constraints are expressed as predicates in a logic Prolog-based language while variables may be either *simple*, *structured* (tuples) or *lists*. The environment provides several solvers and it is capable of reasoning about boolean, interval, linear and arithmetic constraints among others.

The proposed approach can easily be codified in any other Constraint Programming language, as all of them provide support for suspensions, lists and finite domain solvers. Some examples of alternative solvers would be GNU-Prolog [51], Oz [86], CHIP V5 [38], ILOG CP [59], JaCoP [62], Comet [39] or Cream [40]. The translation of UML/OCL diagrams into other families of restricted constraint problems, such as SAT or SAT Modulo Theories (SMT), requires completely different encoding strategies which are out of the scope of this chapter. These strategies depend on the specific formalism being used, e.g. see [97] for a translation of UML/OCL into SAT.



context Researcher **inv** NoSelfReviews:
self.submission−>excludes(self.manuscript)

context Paper **inv** PaperLength:
self.wordCount < 10000

context Paper **inv** AuthorsOfStudentPaper:
self.studentPaper = self.author−>exists(x | x.isStudent)

context Paper **inv** NoStudentReviewers:
self.referee−>forAll(r | not r.isStudent)

context Paper **inv** LimitsOnStudentPapers:
Paper::allInstances()−>exists(p | p.studentPaper) and
Paper::allInstances()−>select(p | p.studentPaper)−>size() < 5

Figure 3.2: Running example: a UML class diagram with OCL constraints.

3.4 From UML/OCL to Constraint Programming: static aspects

Errors can be inadvertently introduced even in very small UML/OCL models. As an example, consider the simple class diagram of Fig. 3.2 that will be used as a running example. The diagram models the relationship between researchers and the papers they write (association *Writes*) or review (association *Reviews*).

This class diagram is complemented with a set of OCL expressions that specify additional constraints for the model. For instance, *NoStudentReviewers* states that the referees of a paper cannot be students. In this constraint, the *self* variable represents an arbitrary instance of the context type chosen to define the constraint, in this case *Paper*, and the constraint must be true for all possible values of *self*; the expression *self.referee* retrieves the Set of all the researcher objects linked to the paper *self* through the association *Reviews* and the *forAll* quantification evaluates the *not isStudent* condition on the collection of researchers retrieved by that expression and returns true if all of them satisfy it.

Notice that several expressions in these OCL invariants, such as *self.referee*, are computing collections of objects and operating with them. Some examples of these operations are checking if an object is not included in a Set (operation *excludes* in invariant *NoSelfReviews*), computing the number of elements in the Set (operation *size* in invariant *LimitsOnStudentPapers*), computing the subset of elements satisfying a property (*select* in invariant *LimitsOnStudentPapers*) or checking existential or universal properties on elements of the Set (*forAll* and *exists* quantifiers). As it is shown in this example, OCL collections allow the concise definition of complex properties and they are an important notion in the OCL notation.

Even if perhaps it is not easy to see at first sight, this model is wrong because

it does not satisfy a basic correctness property: *strong satisfiability*. A model is strongly satisfiable if it is possible to create at least a valid and non-empty instantiation of the model, i.e. if a user can possibly create a finite set of new objects and links over the classes and associations of the model so that no constraint is violated. Therefore, non-satisfiable models are completely useless since users will never be able to populate them at run-time in a way that all constraints evaluate to true. In particular, this example is unsatisfiable due to two different reasons:

1. The multiplicities of association *Reviews* require exactly three distinct researchers per paper, as indicated by the number three next to the *referee* role that the Research class plays in the association Reviews. If we denote by $|X|$ the number of objects of a given class, this multiplicity means that $|Researcher| = 3 \cdot |Paper|$. Meanwhile, the multiplicities of *Writes* requires one or two researchers per paper (multiplicity "1..2" next to the author role), and therefore $|Paper| \leq |Researcher| \leq 2 \cdot |Paper|$. Only an infinite or empty instantiation may satisfy both constraints simultaneously.

2. Students cannot be referees according to constraint *NoStudentReviewers*. However, all researchers must be authors (due to the minimum 1 multiplicity in *Writes*), all authors must review papers (minimum 1 multiplicity in *Reviews*) and there must be at least one student paper (constraint *LimitsOnStudentPapers*) with an student author (constraint *AuthorsOfStudentPaper*).

We have shown that this simple example does not satisfy a property which is assumed by default by any model designer: that it is possible to build a finite and non-empty instantiation of the model without violating the visual and textual constraints it contains. In addition to this notion of strong satisfiability, another reasonable assumption is that the model does not contain *subsumed constraints*, i.e. a constraint which can be removed without changing the set of legal instances. A subsumed constraint may be a symptom of an unexpected interaction among constraints or the incorrect definition of some constraints. If we do not fix these type of errors in the modeling phase at design-time, developers will waste their time implementing this model in the final technology platform before realizing, when testing the system at run-time, that it contains fundamental errors.

In this sense, the main goal of this section is to present a method for the fully automatic, decidable and expressive verification of UML/OCL class diagrams. Decidability is achieved by defining a *finite* solution space, i.e. establishing finite bounds for the number of instances and finite domains for attribute values to be considered during the verification process. This way, the constraint solver is able to perform a *complete search* within the solution space. We will argue that considering a finite solution space is a reasonable trade-off regarding the features offered by other existing verification methods.

Our method uses the Constraint Programming paradigm [77] as an underlying formalism. We have developed a systematic procedure for the transformation of a UML class diagram annotated with OCL constraints into a Constraint Satisfaction Problem (CSP). A predefined set of correctness properties about the original UML/OCL diagram, such as satisfiability of the model, liveliness of a class, redundancy of a constraint and so forth, can then be checked on the resulting CSP. Our choice of using UML/OCL models as input is based on the wide adoption of the UML within the software community and its high-level modeling constructs, not tied to any particular implementation technology. However, we believe that many of the concepts introduced can be useful for verifying correctness of models specified with other modeling languages as well, such as Domain-Specific Modeling Languages.

Moreover, in order to improve the usability of our verification method, we have developed a graphical front-end tool called UMLtoCSP [106] which hides the underlying analysis process. The input of the tool is a UML class diagram encoded in an XMI or Ecore file formats plus (optionally) a text file with the OCL constraints. Meanwhile, the output of the tool is a UML object diagram that proves the property (if it holds). Users of the tool do not need to be familiar with Prolog or CSPs to use the tool: the input and output notations are amenable to UML designers and the entire verification process is completely automated and hidden from the user. In this sense, we follow the paradigm of *hidden formal methods* [14] to improve the usability of the tool and its results.

3.4.1 Overview of the approach

To determine the correctness of a model, our method follows the procedure depicted in Figure 5.1. First, the designer provides an input UML/OCL model, created using an existing UML CASE tool. Then, the designer selects the correctness property to evaluate on the model. These correctness properties study the feasibility of creating legal instances of classes and associations in the model (satisfiability properties) and the interactions among different integrity constraints (subsumption and redundancy properties). Next, the model plus the correctness property is translated into a CSP such that the CSP has a solution if and only if the model satisfies the property. The translation process uses our own specialized CSP library encoding the semantics of the UML and OCL constructs in order to simplify the transformation.

The actual evaluation of the CSP is made with a state-of-the-art constraint solver. The results reported by the solver are interpreted and passed back to the user as an object diagram that proves the property (if there is a solution to the CSP) or as a text message informing that the property is not satisfied.

Intuitively, the generated CSP describes the possible set of valid instantiations of the model by using (list) variables that encode the objects and links in the instantiation, the values of the attributes of those objects, etc. The domain of the variables maps the structure and types of the elements in the model. Integrity constraints in the model such as multiplicity constraints or OCL invariants are translated into constraints in the CSP that restrict the legal values

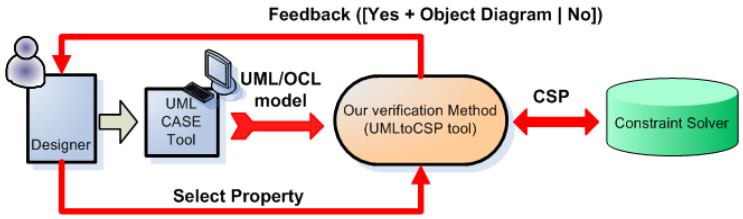


Figure 3.3: Schema of our method.

for these variables. The correctness properties are represented as additional constraints in the CSP. For instance, satisfiability (non-emptiness of the instantiation) can be imposed as a new constraint: a lower bound on the number of objects and links, i.e. a constraint on the minimum size of the corresponding lists. To find a solution, the constraint solver tries to assign a value to all variables without violating any constraint. If no legal assignment is possible, the model fails to satisfy the property. The next Section provides more information about the search in a CSP.

As an example, the CSP for the running example about Papers and Researchers would roughly consist of four list variables that represent the population of the *Paper* and *Researcher* classes and of the *Writes* and *Reviews* associations. The structure of the elements of each list mirrors the structure of the corresponding model elements. Several constraints restrict the possible number and values of elements in the lists. For instance, constraints will ensure that each paper in the *Papers* list appears at least once in the *Writes* list (all papers must be written by at least a *Researcher* according to the constraints in the model) and that its *wordCount* value is lower than 10000 (as forced by the *PaperLength* constraint). On top of this initial CSP, we need to add the constraints to ensure that the model satisfies the correctness property we are interested in. As an example, when checking for strong satisfiability our method would add a new constraint into the CSP stating that none of the four list variables can be empty.

The analysis of this CSP by the solver would conclude that it is not possible to find a solution since the solver will be unable to create and assign elements to the lists in such a way that (1) all previous constraints are fulfilled and (2) at the same time, the lists are not left empty (forbidden by the constraint imposed by the satisfiability property we are trying to determine). Therefore, we may conclude that this model is not strongly satisfiable.

3.4.2 Translation of UML/OCL Class Diagrams

This section describes the transformation of a UML/OCL class diagram into a Constraint Satisfaction Problem. A class diagram CD is defined as $CD = \langle Cl, As, AC, G, IC \rangle$, where Cl is the set of classes, As is the set of associations, AC the set of association classes, G the set of generalisation sets and IC the set of constraints (either graphical or textual) included in CD.

Each element is translated into a set of variables, domains and constraints in the CSP system. As stated before, domains must be finite. These finite domains can be ensured in several ways: first of all, arbitrary bounds for the domains can be chosen or provided by the designer during the translation process. On the other hand, the analysis of the constraints in IC may reveal a finite set of relevant values in the domain. From the point of view of efficiency, we are interested in the smallest domains that suffice to identify inconsistencies in the model, but the automatic computation of these domains from the constraints in IC is a complex problem which will not be addressed here. Instead, we will assume in this section that these values are provided as inputs (parameters) of our translation procedure.

In the following we present the transformation of the elements of a class diagram into the CSP. Note that some of the constraints generated by our method in the CSP are implicit in the semantics of UML but must be made explicit in the CSP. For example, we need to state explicitly that all instances of a class are also instances of its superclasses. In [54] a translation of all these graphical constraints into an OCL representation is proposed.

Section 3.4.2 describes the translation of the UML elements in the model while section 3.4.2 focuses on the translation of the OCL integrity constraints. Both parts rely on our UML/OCL CSP library, introduced in Section 3.4.3. The library extends predicates available in the Prolog dialect used by ECL^{i}PS^e with a new set of predicates that map the semantics of the predefined OCL operations. Rather than hard-coding these OCL operations in the method translation procedure, we have decided to group them in a separate library to simplify the translation.

Transformation of UML constructs

To illustrate the translation of UML constructs, we will refer to the example in Figure 3.4 throughout this section. Figure 3.4(a) shows a class diagram and Figures 3.4(b) and (c) show the translation of this diagram into variables and constraints of a CSP, plus a potentially legal instantiation.

Transformation of classes

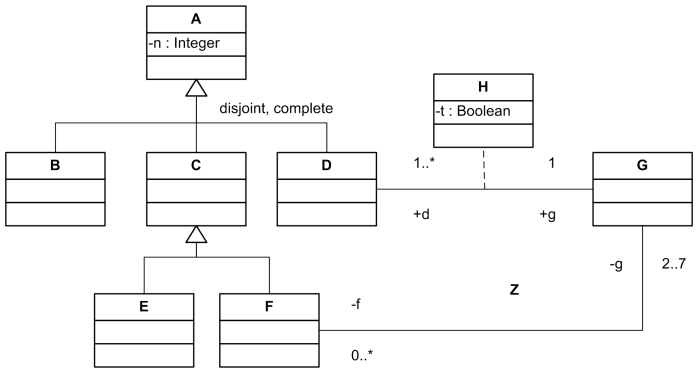
The set of variables and domains to be defined for each class $c \in Cl$ is:

- A variable $Instances_c$ of type list[2]. Each element in the list represents an instance of c. Therefore, the domain of these elements is represented by the structure $struct(c) = (oid, f_1, \ldots, f_n)$, where: oid represents the explicit object identifier for each object, and each f_i corresponds to an attribute $at \in c.ownedAttribute$[3].

  The domain of the oid field is the set of positive integers. The domain of an f_i field is defined as a finite subset of the domain of the corresponding at attribute in c. Boolean and enumerated types are already finite. Finite

[2] Sets of instances are defined as Prolog lists with additional constraints to avoid duplicates.

[3] *ownedAttribute* is the UML metamodel navigation expression that returns the set of attributes of a class.



(a)

A (oid_a, n)	**B** (oid_b)	**C** (oid_c)	**D** (oid_d)	**E** (oid_e)	**F** (oid_f)	**G** (oid_g)	**H** (oid_h, t, d, g)	**Z** (f, g)
$\{(1,10),(2,14),(3,14),$ $(4,20),(5,90),(6,20),$ $(7,10)\}$	$\{(1)\}$	$\{(2),$ $(3)\}$	$\{(4),$ $(5)\}$	$\{(2)\}$	$\{(2)\}$	$\{(1),$ $(2)\}$	$\{(1,true,4,1),$ $(2,false,5,2)\}$	$\{(2,1),$ $(2,2)\}$

(b)

Constraints on association Z

Existence of referenced objects	$\{Z.f\} \subseteq \{oid_f\}, \{Z.g\} \subseteq \{oid_g\}$
Bounds on cardinalities	$(Size_z \leq Size_f \cdot Size_g), (2 \cdot Size_f \leq Size_z \leq 7 \cdot Size_f)$
Multiplicities of role "g"	$\forall x \in \{oid_f\} : 2 \leq \{\#l \in Z : l.f = x\} \leq 7$

Constraints on association class H

Existence of referenced objects	$\{H.d\} \subseteq \{oid_d\}, \{H.g\} \subseteq \{oid_g\}$
Bounds on cardinalities	$(Size_h \leq Size_d \cdot Size_g), (Size_g \leq Size_h), (Size_d \leq Size_h \leq Size_d)$
Multiplicities of role "g"	$\forall x \in \{oid_d\} : 1 \leq \{\#l \in H : l.d = x\} \leq 1$
Multiplicities of role "d"	$\forall y \in \{oid_g\} : 1 \leq \{\#l \in H : l.g = x\}$

Constraints on generalisation set A-B-C-D

Number of instances	$(Size_a \geq Size_b), (Size_a \geq Size_c), (Size_a \geq Size_d)$
Existence of oids in supertype	$\{oid_b\} \subseteq \{oid_a\}, \{oid_c\} \subseteq \{oid_a\}, \{oid_d\} \subseteq \{oid_a\}$
Disjointness (cardinalities)	$Size_a \geq Size_b + Size_c + Size_d$
Disjointness (oids)	$\{oid_b\} \cap \{oid_c\} = \{oid_b\} \cap \{oid_d\} = \{oid_c\} \cap \{oid_d\} = \emptyset$
Completeness (cardinalities)	$Size_a \leq Size_b + Size_c + Size_d$
Completeness (oids)	$\{oid_a\} = \{oid_b\} \cup \{oid_c\} \cup \{oid_d\}$

Constraints on generalisation set C-E-F

Number of instances	$(Size_c \geq Size_e), (Size_c \geq Size_f),$
Existence of oids in supertype	$\{oid_e\} \subseteq \{oid_c\}, \{oid_f\} \subseteq \{oid_c\}$

(c)

Figure 3.4: Example of the translation of UML class diagram constructs: (a) class diagram, (b) corresponding variables in the CSP with a possible legal instantiation and (c) a selection of corresponding constraints in the CSP.

domains for integer types requires at least a lower and upper bound for the attribute. For real types we need also a maximum decimal precision. For string types, the possible "alphabet" and the maximum string length should be defined.

To increase the efficiency of the generated CSP, during the translation we discard all attributes that do not participate in any of the constraints in IC. A correct instantiation may contain any value in those attributes.

- A variable $Size_c$ of type integer, encoding the number of instances of class c. Its domain is $domain(Size_c) = [0, PMaxSize_c]$, where $PMaxSize_c$ is a parameter that indicates the maximum number of instances of class c that must be considered when looking for a solution to the CSP.

Additionally, the following constraints are added to the CSP:

- *Number of instances*: $Size_c = \text{length}(Instances_c)$
- *Distinct oids*: $\forall x, y \in Instances_c : x \neq y \rightarrow x.oid \neq y.oid$

For example, figure 3.4(b) illustrates the structure of the $Instances$ variable for classes A to G, where the constraints on the number of instances and the uniqueness of oids within a class are omitted for brevity. Notice how even in classes without attributes, it is still necessary to keep track of the *oid*, for example, to keep track of inheritance relations or the participation of each object in associations.

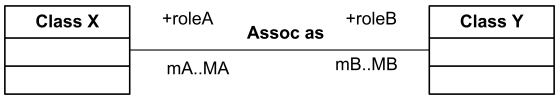
Transformation of associations

For each association $as \in As$ between classes $C_1 \dots C_n$, the following variables and domains must be created in the CSP:

- A variable $Instances_{as}$ of type list. Every member of the list represents an instance of the association (i.e. a link), each being of type $struct(as) = (p_1, \dots, p_n)$, where $p_1 \dots p_n$ are the role names of the participant classes. The domain of each p_i is that of positive integers, that is, each link records the collection of oids of the participant objects, not the objects themselves.
- A variable $Size_{as}$ encoding the number of instances of the association. Its domain is $domain(Size_{as}) = [0, PMaxSize_{as}]$. As before, $PMaxSize_{as}$ is the parameter indicating the maximum number of links of as to be considered when looking for valid solutions of the CSP.

Let n be the number of roles in the association as, and given a role i, let $T(i)$ be its type and $[m_i, M_i]$ be its multiplicity. Then, the following constraints must also be added to the CSP:

- *Number of links*: $Size_{as} = \text{length}(Instances_{as})$
- *Existence of referenced objects*: $\forall l \in Instances_{as} : \forall i \in [1, n] : \exists x \in Instances_{T(i)} : x.oid = l.p_i$



$$Size_{as} \leq Size_X \cdot Size_Y$$
$$m_a \cdot Size_Y \leq Size_{as} \leq M_a \cdot Size_Y$$
$$m_b \cdot Size_X \leq Size_{as} \leq M_b \cdot Size_X$$

Figure 3.5: Implicit cardinality constraints due to the association multiplicities [36]

- *Uniqueness of links*: $\forall x, y \in Instances_{as} : x \neq y \rightarrow (\exists i \in [1, n] : x.p_i \neq y.p_i)$ unless the property *isUnique* [84] of the association is set to false.
- *Bounds on cardinalities*: The multiplicities of an association impose constraints on the number of instances of the participant classes and the association. The explicit representation of these constraints in the CSP is presented in Fig. 3.5.
- *Multiplicities of the association*: Multiplicity constraints must also be satisfied by each individual object of the participant classes. For n-ary associations, several multiplicity constraints among participant objects may be defined [84]. In particular, for binary constraints the following constraint must hold:

  $(\forall x \in Instances_{T(1)} : \ m_2 \leq \{\#l : l \in Instances_{as} : l.p_1 = x\} \leq M_2) \ \wedge$
  $(\forall y \in Instances_{T(2)} : \ m_1 \leq \{\#l : l \in Instances_{as} : l.p_2 = y\} \leq M_1).$

Figure 3.4(b) illustrates the $Instances$ variables for a binary association Z: it keeps track of the oids of the participating objects from classes F and G. Then, Figure 3.4(c) shows relevant constraints added to the CSP for this association. The notation $\{Z.f\} \subseteq \{oid_f\}$ is a shorthand denoting that all oids referenced from role f in $Instances_z$ should correspond to one of the oids in $Instances_f$. Regarding the rest of constraints, it should be noted that there is a role with multiplicity "0..*", i.e. any number of participating objects. This multiplicity does not generate any constraint for this role in the CSP.

Associations that are not referenced (i.e. navigated) in any constraint and that do not state any multiplicity constraint (all participants have a "0..*" multiplicity) can be discarded during the translation process. The population of those associations does not affect the existence of solutions to the CSP.

Compositions and aggregations are just two special kinds of associations and thus are translated following the procedure explained in this section complemented with the translation of the OCL constraints needed to enforce their specific *containment* or *whole-part* semantics. These constraints are taken from [54] and added to the pool of OCL constraints of the model translated as explained in the next Section.

Transformation of association classes

An association class $ac \in Ac$ is, at the same time, a class and an association. Therefore, transformation of association classes can be regarded as the union of the translation process for classes plus the translation process for associations.

More specifically, variables for association classes are $Instances_{ac}$ and $Size_{ac}$ where the structure of elements in $Instances_{ac}$ includes all fields corresponding to the transformation of the class facet of ac plus the fields corresponding to the transformation of the association facet of ac. Likewise, constraints for ac are the combination of constraints for classes and for associations. Therefore, this transformation considers the special semantics of association classes [54,84] stating that each instance of the association class should correspond to a link in the underlying association.

For example, Figures 3.4(b) and (c) illustrate the variables and constraints for the associative class H from our example. Notice that the structure of the *Instances* variable includes oids and attributes like objects, and also role names like associations.

Transformation of generalisation sets

Generalisation sets do not imply the definition of new variables but the addition of new constraints among the classes involved in the generalisation.

Let class $sub \in Cl$ be a subclass of a class $super \in Cl$. The following constraints should be added:

- *Existence of oids in supertype*: $\forall x \in Instances_{sub} : \exists y \in Instances_{super} : x.oid = y.oid$
- *Number of instances*: $Size_{sub} \leq Size_{sup}$
- *Disjointness*: For a disjoint generalization set among a supertype S and subtypes $S_1..S_n$:
  - $Size_S \geq \sum_i Size_{Si}$
  - $\forall i, j \in [1, n] : \forall o_1 \in Instances_{Si}, \forall o_2 \in Instances_{Sj} : o_1.oid = o_2.oid \rightarrow i = j$
- *Completeness*: For a complete generalization set among a supertype S and subtypes $S_1..S_n$:
  - $Size_S \leq \sum_i Size_{Si}$
  - $\forall o_1 \in Instances_S : \exists i \in [1, n] : \exists o_2 \in Instances_{Si} : o_1.oid = o_2.oid$

For example, Figure 3.4(c) describes the set of constraints involved in two different generalization sets: the subclasses of A and the subclasses of C. Also, Figure 3.4(b) shows an instantiation of these classes which helps to illustrate the role of oids through inheritance: an object preserves the same oid in the subclass and the superclass. Even though this approach does not support multiple inheritance, it is able to describe complex inheritance scenarios. For example, it supports *overlapping* inheritance, i.e. the same oid is used in two or more subclasses of the same superclass, and also *complete inheritance*, i.e. all oids from the superclass must be used in at least one of the subclasses.

Transformation of the running example

Before describing the translation of OCL constraints, we retake our running example from Figure 3.2 and illustrate the translation process described so far.

```
% Unconstrained attributes are not in the CSP
:-local struct researcher(oid,isStudent).
:-local struct paper(oid,wordCount,studentPaper).
:-local struct reviews(submission,referee).
:-local struct writes(manuscript,author).
```

Figure 3.6: Structs for the translation of classes and associations of the running example.

This time we introduce the syntax of the ECL^{i}PS^e code for the CSP, which will be used extensively in the following Section.

Figure 3.6 illustrates the *Instances* variables for the classes and associations in the running example. Notice that some attributes such as *name* from class *Researcher* or *title* from class *Paper* do not appear in the *Instances* variables to improve the efficiency of the solver. Regarding the constraints for the UML constructs in this diagram, they primarily involve the multiplicities of associations *Writes* and *Reviews*. The ECL^{i}PS^e representation for these constraints is provided in the Appendix. Some example predicates would be `differentOids`, which checks that all oids of a single class are different, or `linksConstraintMultiplicities`, which ensures that the oids of participants in a binary association preserve the multiplicities of each role.

Translation of OCL constraints

Integrity constraints in OCL [83] are represented as invariants defined in the context of a specific type, named the *context type* of the constraint. Its body, the boolean condition to be checked, must be satisfied by all instances of the context type. In our approach, each OCL constraint is translated into an equivalent constraint in the CSP. Fig. 3.7 shows an example of the translation process presented in this Section. This example has been simplified to improve the legibility and understandability of the constraints. The interested reader can find the original output in the Appendix.

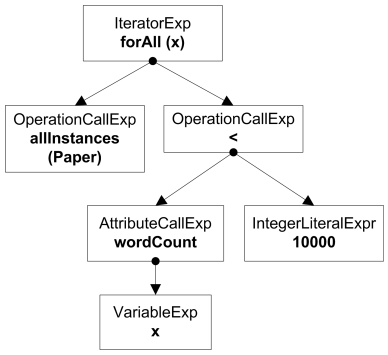
An OCL constraint can be viewed as an instance of the OCL metamodel with a tree shape, with a close resemblance to the abstract syntax tree of the textual constraint that would be constructed by an OCL parser. For instance, the simplified tree representation for *PaperLength* constraint is illustrated in Fig. 3.7. Leaf nodes of the tree correspond to the constants (e.g. 2, *true*, "John") and variables (e.g. *self*, x) of the constraint. Each internal node corresponds to one atomic operation of the constraint, e.g. logical or arithmetic operation, access to an attribute, operation calls, iterator, etc. The root of the tree is the most external operation of the constraint. Packages like the Dresden OCL toolkit [44] can parse textual OCL constraints and build the corresponding trees.

As a preliminary step and to homogenize the translation procedure, we express all constraints in terms of the *allInstances*[4] operation using the following

[4]allInstances is a predefined OCL operation that returns the set of instances of the type.

context Paper **inv** PaperLength:
Paper::allInstances−>
forAll(x|x.wordCount < 10000)

(a)



(b)

```
% Position of class Paper
% within the list of
instances
index("Paper", 1).

% Position of attribute
wordCount
% within the list of
attributes
attIndex("Paper",
"wordCount", 2).

nodeConstant(_, _, Result):-
   Result = 10000.

nodeVariable(_, Vars, Result ):-          % x = var of the innermost iterator
   nth1(1, Vars, Result).                   % Result = Vars[1] = value of x

nodeAttrib(Instances, Vars, Result):-
   nodeVariable(Instances, Vars, Object),  % An object of class Paper
   attIndex("Paper", "wordCount", N),      % N = Index of field wordCount
   arg(N, Object, Result).                   % Result = Object[N] = wordCount
value

nodeAllInstances(Instances, Vars, Result) :-
   index("Paper", N),                      % N = Position of class Paper
   nth1(N, Instances, Result).               % Result = Instances[N] = Inst of
Paper

nodeLessThan(Instances, Vars, Result) :-
   nodeAttrib(Instances, Vars, Value1),    % 1st subexpression
   nodeConstant(Instances, Vars, Value2),  % 2nd subexpression
   #<(Value1, Value2, Result).                  % Result = (Value1 < Value2)?

nodeForAll(Instances, Vars, Result) :-
   nodeAllInstances(Instances, Vars, L),   % L = Result of allInstances
   ( foreach(Elem, L), foreach(Eval, Out), param(Instances,Vars) do
       % Eval = Result of evaluating nodeLessThan on an element of L
       nodeLessThan(Instances, [Elem|Vars], Eval) ),
   % Out = List of truth values.  Out[i]= Result of nodeLessThan(L[i])
   length(L, N),                           % N = length(L)
   #=(N, sum(Out), Result).                  % Result = (N = ΣOut[i])?

% Translation of the constraint PaperLength
paperLength(Instances) :-
   nodeForAll(Instances,[],Result),        % Evaluate the root node
   Result #= 1.                             % Result should be true
```

(c)

Figure 3.7: Translation of OCL constraints: (a) Class invariant after preprocessing, (b) OCL metamodel tree, (c) Constraint represented by means of Prolog rules in the CSP.

expansion rule:

context T **inv**: B ⇒ **context** T **inv**: T::allInstances()−>forAll(v|B')

where B' is obtained by replacing all occurrences of *self* in B with v.

Then, the translation procedure is defined as a post-order traversal of the corresponding OCL metamodel tree that translates all the children (subexpressions) of a node before translating the node (expression) itself. Each node of the tree is translated into an ECL^{i}PS^e Prolog compound term with an *unique functor name* that identifies the subexpression and *three arguments*, e.g. `nodeX(Instances, Vars, Result)`, with the following meaning:

1. *Instances* is a list with the set of all *instances* for each class and association. The i-th position of this list holds all the instances of class/association i, i.e. it holds the contents of the corresponding $Instances_i$ variable defined in Section 3.4.2. The order within this list is defined in auxiliary Prolog rules generated during the translation. This argument is required, for instance, to implement the OCL operation *allInstances* and navigation in associations.

2. *Vars* contains the list of the *quantified variables* available in the subexpression. The first position of this list holds the value of the quantified variable defined in the innermost iterator (e.g. *forAll* or *exists*). The second position holds the following variable in the next innermost iterator and so on. This argument will be used when evaluating attribute, operation or navigation expressions over variables defined in an iterator.

3. *Result* holds the *result* of the subexpression. The type of the result depends on the kind of operation applied in the node.

The behaviour of each node is formalised by means of a Prolog *rule*. This rule evaluates the subexpressions of the node and computes the result of the node (according to the semantics of the OCL operation represented by the node) in terms of the results of its subexpressions. Basic types (e.g. boolean, integer or real) and basic OCL operations (e.g. logical and arithmetic) have a direct counterpart implementation in the ECL^{i}PS^e constraint libraries. For more complex operations, such as iterators or operations on Collections, we have developed a new ECL^{i}PS^e library (see the next section) that implements the operations defined in the OCL Standard Library [83]. Nevertheless, for the sake of simplicity, in Fig. 3.7 we have directly added to each node the required computation without relying in our external library.

As a final step, once the translation for the body expression has been completed, we add to the CSP a new constraint representing the original OCL invariant. This constraint is defined as: `nameConstraint(Instances):-`

`rootNode(Instances,[],Result), Result#=1`, that is, the constraint is true when the `rootNode` evaluates to true. For example, see the *paperLength* constraint in Fig. 3.7(c).

3.4.3 UML/OCL Prolog Library

OCL provides a rich set of predefined types (e.g. collections: sequences, sets, bags,...), operations (arithmetic, logic, set,...) and iterator expressions (for all, exists, iterate,...) for the definition of complex constraints as part of the OCL standard library. In order to analyze OCL constraints, it is necessary to provide a translation for those operations in terms of the Prolog-based language used by the ECL^{i}PS^e solver. Rather than hard-coding these operations in the tool translation procedure, we have decided to group them in a separate library. When translating the OCL constraints, we will call the appropriate library predicate to implement the specific semantics of each constraint node. For example, the library predicate `ocl_int_equals` will be reused every time a constraint includes an integer equality comparison between two subexpressions. This separation provides additional flexibility as changes in the implementation of the OCL operators can be performed without modifying the translation method and thus, without changing the source code of UMLtoCSP[5].

The Prolog dialect used by ECL^{i}PS^e is an extension of pure Prolog. A full description of this notation is available in [5]. We will simply mention those extensions required for the comprehension of the implementation of our UML/OCL library:

- Support for higher-order predicates, i.e. the ability to pass predicate names as arguments.
- Syntactic flavor to facilitate the definition of constraints, e.g. iterator constructs like `for` or `foreach`. These constructs are specially useful to operate with Prolog lists, so they are used in collection operations, OCL iterators and navigations.
- Support for the definition of *suspended* constraints, i.e. delaying the execution of a predicate whose arguments have not been assigned yet.

The first extension, higher-order predicates, is provided by a library called `apply`. With this library, it is possible to pass predicate names as arguments of other predicates, and invoke those predicates later with a list of arguments that is constructed at run-time. Intuitively, it is possible to build generic predicates like "evaluate this predicate in each element of a collection", i.e. the "collect" function of collections. This function is computed by a predicate with the following signature: `ocl_set_collect(Instances, Vars, Set, Predicate, Result)`, where `Set` is the collection where the operation must be applied, `Predicate` is the name of the predicate to be applied and `Result` is the result of the operation. It is possible to implement iterators like "exists" or "forAll" in a similar way.

In the following sections, we will describe some design decision and implementation details of the UML/OCL Prolog library (available at [106] as a part of the download of the tool UMLtoCSP). We will discuss separately the implementation of basic types (Subsection 3.4.3), issues with suspensions (Subsection

[5] In fact, it is not even necessary to recompile the tool, as Prolog is an interpreted language.

3.4.3), the implementation of OCL collections (Subsection 3.4.3) and the OCL iterator expressions (Subsection 3.4.3).

OCL Basic Types

ECL^{i}PS^e provides several solver libraries, each one targeting a different type of constraints, e.g. linear constraints, graph constraints, ... In UMLtoCSP, we use the *finite domain/real number interval constraint solver* library (`ic`), which analyses constraints by considering the domain of each variable as one or more finite intervals of values. This library provides support for integers, floats and boolean[6] variables, and defines all the usual arithmetic ($+, *, /, \%$), relational ($=, \neq, <, >, \leq, \geq, \min, \max$) and boolean ($\wedge, \vee, \rightarrow$) operators. Furthermore, those operators which are not provided by the library itself, e.g. xor, can be simply defined using the Prolog language. Thanks to these predefined operators we can easily implement the support for OCL basic types in our library.

For example, let us consider the implementation of the relational operator "greater-than" for comparing two integers. This operator takes as parameters two predicate names and stores the result of the comparison in a variable called Result. The ECL^{i}PS^e code for this operation is the following:

```
ocl_int_greater_than(Instances, Vars, Pred1, Pred2, Result) :-
   apply(Pred1, [Instances, Vars, X]),  % X is the result of evaluating Pred1
   apply(Pred2, [Instances, Vars, Y]),  % Y is the result of evaluating Pred2
   Result::0..1,                        % Result is a boolean value
   #>(X, Y, Result).                    % Result is true iff X > Y
```

This predicate uses the predefined operator `#>` which is provided by the predefined `ic` library to define the integral "greater-than" constraint. All integer operators in the library have the "`#`" prefix while floating point operators have the "`$`". Whenever there is a potential ambiguity, e.g. between the default Prolog language operator and the operator redefined by the library, the name of the library is prepended to the operator. For example, the logical "and" operator from the `ic` library is invoked as `ic:and` to distinguish it from the default Prolog "and".

Note that in this predicate we are not forcing X to be greater than Y, we are just evaluating whether this is true and storing the result in the *Result* variable. This way of using boolean constraints is known as reified constraints.

A disadvantage of using the `ic` library is the lack of support for constraints on strings. In fact, there is no ECL^{i}PS^e library which provides efficient support for the type of string constraints that can be written in OCL, e.g. substrings or concatenations. In models with attributes of type string that are only compared among them with no substrings and concatenations, strings can be encoded as an integer or enumerated attribute. However, our current implementation does not provide support for other string operations in OCL constraints.

[6] Boolean values are represented as integers whose value is either zero (false) or one (true).

Suspensions

The result of an operation cannot be evaluated until the values of its arguments are known. In the constraint programming paradigm, the constraints of the CSP are defined in the beginning, and then we try to assign values to variables in such a way that all constraints are satisfied. It should be noted that constraints are defined before the values of variables are available, so we need the concept of *suspended* constraint, a constraint which has been defined (e.g. $a = b$) but cannot be evaluated until the values of variables are available (e.g. until we have tried to assign a or b).

A possible solution is delaying the evaluation of a constraint until all the complete assignment is available. However, this conservative approach is very inefficient. Instead, it might be possible to detect that an assignment violates a constraint without assigning values to all variables. For example, if we have variables a to z and a constraint $a = b$, the constraint can be evaluated as soon as a and b are assigned. Waiting for the other variables to become assigned means evaluating the constraint for all possible combinations of variables c to z, a number which grows exponentially with the number of variables involved.

To avoid inefficiencies due to the late evaluation of constraints, ECL^{i}PS^e provides a mechanism to *suspend* constraints when they are defined and *wake* them whenever one of its arguments is assigned. There is a reason why constraints are awakened when *one* argument is assigned and not *all*. The constraint propagation built in ECL^{i}PS^e can sometimes infer information about the result without knowing all the arguments of an operation. For example, if an argument of a product is 0, the result is automatically zero regardless of the other argument. The same type of early evaluation can be applied to boolean operations. Sometimes it is also possible to use the domain of an argument to infer information about the domain of other arguments. For example, in $a = b$ if we assign a the value 7, two situations can happen: if the value 7 is within the domain of b, then we know that the equality holds and $b = 7$; otherwise, we know that the equality does not hold.

In our library, we use two types of suspensions, that of the predefined predicates and that of the new rules that we define:

- In the predefined constraints of ECL^{i}PS^e libraries, like `$>`, or `ic:and`, suspension is already built in. This means that we can write `Value1 $> Value2` and ECL^{i}PS^e handles the suspension transparently.

- In the new rules that we define, it is necessary to specify conditions that restrict when it cannot be evaluated and must become suspended. This is achieved using the declarative suspension clause `delay-if` before the definition of a rule. For example, the clause `delay ocl_col_size(X) if var(X)` delays the execution of the rule `ocl_col_size` until its arguments ceases being an unassigned variable. The constraint will be automatically woken and evaluated by the ECL^{i}PS^e solver when variable `X` is given a value.

In this second group, it is very important to wake constraints as soon as possible: if the current solution is unfeasible, discovering it early will avoid unnecessary backtracking and greatly reduce the execution time. However, it is possible to define suspensions which are too conservative, for example, in collection operators. An operation like nonEmpty() can be evaluated as soon as we know that the list has no elements or more than one, it is not necessary to wait until the specific value of the elements of the collection is available. Selecting the right degree of suspension for each operation has been an important task in the design of the library.

OCL Collection Types

Like many Prolog dialects, lists are a key concept of the ECL^{i}PS^e notation. Prolog lists are represented as a sequence of elements separated by commas and enclosed between brackets, e.g. `[]` or `[2, 7, 25]`. These lists admit the repetition of elements and the order among elements matters. Formally, a list is defined recursively using the empty list (`[]`) and the constructor `|` which appends one element (*head*) to the beginning of an existing list (*tail*), e.g. `[head | tail]`. For example, the previous notation for lists is a shorthand of:

[2 | [7 | [25 | []]]]

Lists will be used as the backbone for the representation of OCL collections: all the elements of an OCL collection will be stored in a Prolog list. The Prolog semantics of lists matches that of OCL sequences, so the implementation of sequence operations will be straightforward. For example, the implementation of operation "size()" of collections, which returns the number of elements in a collection, relies on the "length" operation in Prolog lists:

```
ocl_col_size(Col, Size) :- length(Col, Size)
```

Similarly, other operations rely on Prolog operations on lists.

However, we needed to code additional predicates/constraints to manage other OCL collection types, i.e. to correctly represent their semantics when stored as Prolog lists, e.g. to enforce the uniqueness of elements in a set collection type. These additional predicates/constraints are the following:

Sets: Additional checks are required to ensure that the set contains no duplicates after the insertion of new elements (OCL operation "including") and the union of two sets or a set and a bag (union). To improve the efficiency of this representation, the elements in the Prolog list are kept ordered at all times.

Bags: As lists allow duplicate elements, bags can be represented directly as lists much like sequences. However, the performance of several operations like intersection or the equality check, can be improved in the elements of the bags are ordered. Rather than keeping the elements ordered at all times, the elements are ordered on demand when the equality check or intersection operations are invoked.

Ordered sets: The operations on ordered sets are equivalent to those of a sequence, except for the check to avoid duplicate elements. In this collection, elements cannot be stored in an ordered list as the insertion order must be preserved.

Operations on collections can be classified into two categories according to the degree of suspension that they require. In the first category, there are operations on collections which can be evaluated when the number of elements of the collections is known, even if the specific value of the elements in the collection is unknown, e.g. size(), isEmpty() and most operations on empty collections. Such operations must be delayed until the size of the collection is known. In a second category, other operations need that the values of the elements in the collection are known a priori, e.g. includes() in a non-empty list. In this case, it is not possible to evaluate the operation until all the elements of the collection have been given a value.

Iterators

OCL provides a set of iterator expressions over collections, e.g. existential (exists) and universal (forAll) quantification. As OCL collections are encoded in Prolog lists, the iterators must be translated in terms of lists.

The core operation of iterators is the ability to evaluate an OCL expression in each element of a collection. Talking in Prolog terms, this translates to the ability to apply a predicate (encoding the OCL expression) to each element of a list (encoding the OCL collection). Using the library `apply` it is possible to obtain a generic implementation of this operation by passing the predicate name to be evaluated as a parameter.

For example, let us consider the following existential quantification:

$$Col\ {-}{>}exists(x|Expr(x))$$

It is possible to evaluate this quantifier in the following way: evaluate $Expr$ in each element x of the collection Col and then count the number of $Expr(x)$ that evaluate to true. The quantifier evaluates to true if and only if that number is greater than zero. This implementation in terms of a CSP requires the following variables and constraints:

- **Variables**:
  - $Result$, a boolean variable which stores the result of the existential quantification.
  - An auxiliary variable N, which is an integer ranging from 0 to the number of elements in the collection.
  - A variable $Expr(x)$ in the CSP for each element x in the collection, i.e. a boolean variable which stores the result of evaluating the expression on x.
- **Constraints**:

- The necessary constraints to define the value of each $Expr(x)$.
- A new constraint: $N = \sum_{x \in Col} Expr(x)$.
- A new constraint: $Result = (N > 0)$.

Figure 3.8 illustrates the Prolog code required to compute the existential quantification. The auxiliary predicate `property_sat_count` is also used in the implementation of other operators such as *forAll* (universal quantification) or *one* (checking if a property holds in just one element of the collection). This code reuse is shown in Fig. 3.9. Notice how it is only necessary to change the number of elements which should evaluate to true: in "one" we need only one element, in "forAll" we need all the elements of the collection (we compute the number of elements of the collection using the predicate `ocl_col_size` from the collections).

Using these functions in our translation results in a more compact translation into Prolog. For example, the translation of the node `nodeForAll` in Fig. 3.7 would become the following:

```
nodeForAll(Instances, Vars, Result) :-
   nodeAllInstances(Instances, Vars, L),
   ocl_col_forAll(Instances, Vars, L, nodeLessThan, Result).
```

Finally, it should be noted that it is not necessary to manually add optimizations like "the result of existential quantifier is false when the collection is empty" as they are already considered by the constraint propagation process. If the collection is empty, N will be 0, so $N > 0$ will be false, i.e. the result of the iterator will be false.

3.4.4 Quality criteria for UML/OCL class diagrams

In addition to the elements of the model (classes, associations, constraints, ...), it is necessary to encode in the CSP an additional element: the goal of our analysis, i.e. the type of instance of the model that the solver should try to construct. Depending on the selected goal, it is possible to verify or validate different characteristics of our model. In the remainder of this section, we introduce several goals for the analysis of UML/OCL models.

Correctness properties for verification

A model is expected to satisfy several reasonable assumptions. For instance, it should be possible to instantiate the model in some way that does not violate any integrity constraint. Moreover, it may be desirable to avoid unnecessary constraints in the model. Failing to satisfy these criteria may be a symptom of an incomplete, over-constrained or incorrect model.

In our approach, correctness properties are represented as additional constraints in the CSP. If the CSP still has a solution once the new constraint is

```
ocl_col_exists(Instances, Vars, Collection, Predicate, Result ) :-
   % N = # of elements where Predicate evaluates to true
   property_sat_count(Instances, Vars, Collection, Predicate, N),
   #>(N, 0, Result).               % Result = (N > 0)

% Count the number of Predicates that evaluate to true in the Collection
property_sat_count(Instances, Vars, Collection, Predicate, Result ) :-
   % Apply Predicate to all elements of Collection
   % Store the results in the list TruthValues
   property_apply(Instances, Vars, Collection, Predicate, TruthValues),
   Result #= sum(TruthValues).      % Result is the sum of all the truth values

% Apply Predicate to each Element of Collection, store all outputs in the list Result
property_apply(Instances, Vars, Collection, Predicate, Result) :-
   ( foreach(Elem, Collection),   % One Value per Elem in the Collection
     foreach(Value, Result),      % Result is a list of those Values
     param(Predicate, Instances, Vars)
     do
       % Apply Predicate to Elem (Elem is added to the list
       % of visible variables within Predicate)
       apply(Predicate, [Instances, [Elem|Vars], Value]) ).
```

Figure 3.8: Code for the implementation of the existential quantification (code related to suspensions has been removed for clarity).

```
ocl_col_one(Instances, Vars, Collection, Predicate, Result ) :-
   property_sat_count(Instances, Vars, Collection, Predicate, N),
   #=(N, 1, Result).             % Result = ( N = 1 )

ocl_col_forAll(Instances, Vars, Collection, Predicate, Result ) :-
   property_sat_count(Instances, Vars, Collection, Predicate, N),
   ocl_col_size(Collection, S),  % S = Number of elements in the collection
   #=(N, S, Result).             % All elements should evaluate to true (N = S)
```

Figure 3.9: Code for the implementation of other existential quantifications (code related to suspensions has been removed for clarity).

added, we may conclude that the model satisfies the property. The set of correctness properties currently under consideration (and the additional constraint they impose on the CSP) is the following:

Strong satisfiability: The model must have a finite instantiation where the population of all classes and associations is at least one.
Formally: $\forall x \in \{Cl \cup As\} : Size_x > 0$.

Weak satisfiability: The model must have a finite instantiation where the population of at least one class is at least one.
Formally: $\sum_{x \in Cl} Size_x > 0$

Liveliness of a class c: The model must have a finite instantiation where the population of c is non-empty.
Formally: $Size_c > 0$

Lack of constraint subsumptions: Given two integrity constraints C_1 and C_2, the model must have a finite instantiation where C_1 is satisfied and C_2 is not. Otherwise, we say that C_1 *subsumes* C_2. C_2 could be removed.
Formally: $C_1 \wedge \neg C_2$

Lack of constraint redundancies: Given two integrity constraints C_1 and C_2, the model must have a finite instantiation where only one constraint is satisfied. Otherwise, constraints C_1 and C_2 are called *redundant*, e.g. both have always the same truth value. One of them should be removed.
Formally: $(C_1 \wedge \neg C_2) \vee (C_2 \wedge \neg C_1)$

Notice that there is a relationship among some of these correctness properties, e.g. strong satisfiability implies weak satisfiability and the lack of constraint subsumption among two constraints implies that none of them is redundant. Checking properties with different degrees of granularity improves the feedback provided to designers. For example, we are able to detect that two constraints are equivalent or that one is stronger than another one: both pieces of information can help designers in the revision of the constraints of the model.

Regarding the satisfiability properties, similar notions have been defined in the literature. The difference among each notion depends on whether (a) it refers to a specific class or all classes in a diagram, (b) it accepts empty instances and (c) it accepts infinite instances. For example, in [7] weak satisfiability is called *diagram satisfiability*, liveliness of a class is called *satisfiability of a specific class* and strong satisfiability is called *full satisfiability*. Other works such as [76] define the notions of *consistency* and *finite satisfiability*. A class diagram is called consistent if and only if it has a legal non-empty instance (possibly infinite), and it is called finite satisfiable if one of its legal instances is also finite. Notice that strong satisfiability as defined here and finite satisfiability are completely equivalent.

From a broader point of view, these correctness notions consider the *intra-model semantic consistency* of UML class diagrams, according to the classification of the surveys [73, 78]. This approach is not addressing inter-model

consistencies, i.e. the relationship among the different diagrams modeling the same system. We are also not considering other quality notions such as the completeness of the class diagram with respect to the domain or its comprehensibility, among others [78]. Furthermore, this section does not consider dynamic properties that may appear in the definition of OCL operation pre-conditions and post-conditions, such as the executability of operations or the reachability of a specific system state, e.g. [30, 61, 93].

Model Validation

Apart from verifying that the model satisfies the previous correctness properties, designers may be also interested in checking these properties over specific (partially defined) instantiations, e.g. checking satisfiability when a class c has an instance with a value v in an attribute a, to validate the behaviour of the model in those situations. The declarative nature of our approach allows the definition of additional constraints that characterise these desired states. For example, in our running example, a designer may want to check whether it is possible to find an instantiation where there is one paper with two student authors. An invariant describing this property is the following:

```
context Paper inv PaperByTwoStudents:
Paper::allInstances->exists( p | p.authors->size() = 2 and p.authors
->forAll(r | r.isStudent) )
```

After adding this invariant to the list of integrity constraints of the model, all instances generated by the solver will fulfill this partial specification. Therefore, this method can also be used to perform validation of specific scenarios for the model.

3.4.5 Generating the final CSP

The final CSP is obtained as a combination of the translation excerpts generated using the rules of section 3.4.2 (for the transformation of the UML/OCL diagram) and section 3.4.4 (for the definition of the quality properties to be verified). Remember that this generated CSP has a solution if and only if we can determine that the model satisfies the selected quality properties.

For efficiency reasons the CSP is organized in two subproblems:

1. The *Structural* subproblem. In this first subproblem we define the cardinality variables for the number of instances of each class and association (the $Size_x$ variables), their domains and all constraints restricting them plus the constraints corresponding to the correctness properties we want to check. In this phase, the goal is to find a legal assignment of values to these $Size_x$ variables. Each legal assignment represents the size dimensions (i.e. number of instances of each class and association) of a possible correct instantiation of the model. This subproblem helps to filter incorrect cardinality assignments that would always result in invalid instantiations, no matter the actual values given to the association and

attribute variables. Therefore, if no assignment is possible at this phase (e.g. due to design errors in the definition of multiplicity constraints), the CSP is directly unfeasible. Instead, if this first subproblem is indeed satisfiable we cannot yet guarantee the correctness of the model, it could happen that we find impossible to construct a possible legal instantiation of that size due to inconsistencies in the OCL constraints of the model. The definition of this subproblem is similar to the previous work from [36].

2. The *Global* subproblem. In this second subproblem, the valid values assigned to the $Size_x$ variables are used to instantiate the corresponding $Instances_x$ variables. Now the goal is to find legal values for properties (either attributes or roles) of all elements in the $Instances_x$ lists, i.e. the goal is trying to construct a valid instantiation with exactly $Size_i$ elements for each element i. Intuitively, the procedure tries to find a valid solution for this second subproblem for each assignment satisfying the first one. If there is no such solution, the CSP is determined as unfeasible.

Both phases follow the typical Constraint Programming outline: define the variables and their domains, define the constraints on the variables, and finally, find a legal assignment to these variables. In the *Structural* phase, we work on cardinality variables ($Size_x$), while in the *Global* subproblem we are interested in the set of instances ($Instances_x$) of classes and associations.

As an example, Fig. 3.10 depicts the CSP corresponding to a satisfiable version of our running example[7]. The colored areas highlight the two subproblems of the CSP. On the left of the figure, the organisation of several code excerpts (some of them taken from previous figures) is described. On the right, a possible search tree is depicted, where a dotted line shows the direction of the search. In this tree, after an initial attempt, a solution to the *Structural* subproblem is found, e.g. one paper, three researchers, one "writes" link and three "reviews" links. However, it is not possible to complete the *Global* subproblem using those values as cardinalities for the $Instances_x$ variables. Therefore, it is necessary to find another solution to the first subproblem, which can then be completed to find a valid solution to the CSP, e.g. the same solution with two "writes" links instead of one. The full definition of the CSP for the running example is available in the Appendix.

Although this two-step search strategy might penalise the overall efficiency of the process in certain cases (see Section 3.9), it has been chosen because of two reasons. First, some CSPs, as the one corresponding to our running example, can be immediately determined as unfeasible when considering the *Structural* subproblem. Second, in the *Global* subproblem we limit the search to cardinality values satisfying the first one, avoiding irrelevant verifications. This means that we can avoid instantiating classes and associations for scenarios in which we know for sure that the cardinality constraints do not hold. However, a disadvantage of this approach is that some constraints in the *Global* subproblem

[7] Fig.3.2 becomes satisfiable if the multiplicities of *manuscript* and *submission* are changed to 0..1. This version of the model is used to illustrate a successful search. There is more information about this example in Section 3.8.

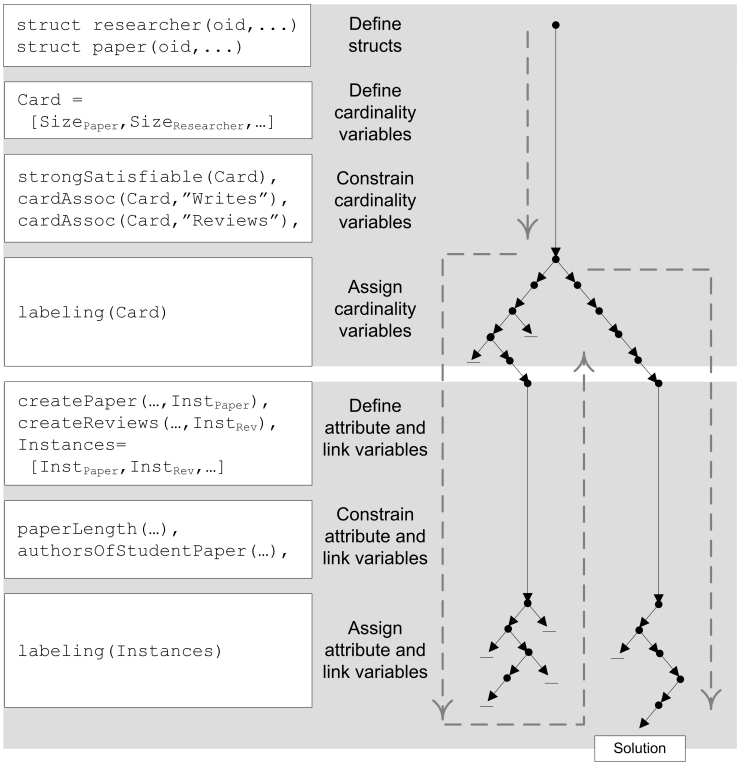


Figure 3.10: Definition of the CSP for the running example showing the two subproblems, the *Structural* subproblem (upper part) and the *Global* subproblem (bottom part)

may affect the cardinality of the classes and associations. For example, let us consider an OCL constraint of the form "T::allInstances()−>size() = 7", stating that there are 7 objects of type T. This constraint would appear in the second subproblem, even though it should be constraining the appropriate $Size_T$ variable. Other constraints on the size of classes and associations are not so trivial, like "T::allInstances()−>exists(...)", which requires there must be at least one object of type T. In these scenarios, the search will have unnecessary backtracks, as the solution provided by the *Structural* phase will always be unfeasible when it reaches the second phase. A way to avoid these redundant backtracks and therefore improve the search is to reveal these implicit size constraints appearing in OCL expressions using static analysis [113], e.g. $Size_T = 7$ and $Size_T \geq 1$ in the previous examples. These additional size constraints could be added to the first CSP to avoid getting unfeasible solutions caused by the OCL expressions.

3.5 From UML/OCL to Constraint Programming: dynamic aspects

Once we have checked that the static aspects of the UML/OCL model satisfy a minimum correctness level using the method presented in the previous section, we can focus on the verification of its dynamic aspects.

In particular, this section extends the previous method with support for the verification of the behavioural aspects of software models defined using the design by contract approach [112], where each operation is defined by means of a contract consisting of a *precondition* (set of conditions on the operation input) and a *postcondition* (conditions to be satisfied at the end of the operation). In conceptual modeling, this is also known as the declarative specification of an operation, in contrast to imperative specifications where the set of updates produced by the operation on the system state is explicitly defined (see our paper [89] for a lightweight method for the verification of imperative specifications). Our goal will be detecting defects in the definition of the operation (e.g. potential inconsistent interactions with integrity constraints) rather than checking whether an implementation fulfills the pre/postconditions. To do so, we present a set of "reasonable" correctness criteria that any operation should fulfill. For example, we will try to check if a precondition is so strong that it cannot be satisfied by any state that fulfills the integrity constraints (e.g. a precondition "a ≥ 5" when the model includes the constraint "a ≤ 3" is clearly unsatisfiable). Designers can select their preferred set of criteria among the predefined set of properties we propose.

As before, the verification will be driven by the discovery of examples/counterexamples. First, the designer selects the criteria to be checked. The model, the integrity constraints, the correctness criteria and the pre/postconditions will be transformed into a Constraint Satisfaction Problem (CSP) [5, 77] that can be solved by current Constraint Programming solvers. The solution of the CSP, if there is one, will be an example or counterexample that proves the criteria being an-

alyzed. The example is given to the designer as a valuable feedback in the form of an object diagram (so that he/she can understand it).

3.5.1 Declarative Operations in OCL: Basic Concepts

OCL is a formal high-level language used to describe properties on UML models. It admits several powerful constructs like quantified iterators (*forAll*, *exists*) and operations over collections of objects (*union*, *select*, *includes*, ...). The pattern for specifying a declarative operation *op* in OCL is the following:

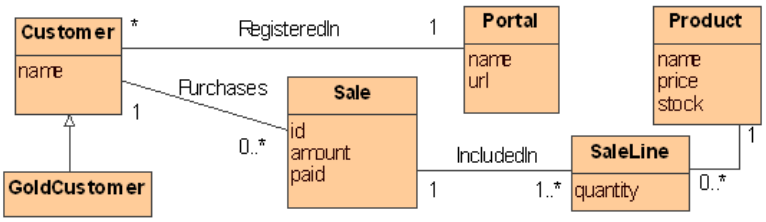
context TypeName::op(p1: Type1, ..., pN: TypeN): ResultType
pre Boolean expression (the precondition)
post Boolean expression (the postcondition)

Operations are always defined in the context of a specific type of the model. The *pre* and *post* clauses are used to express the preconditions and postconditions of the operation contract. In the boolean expressions, the implicit parameter *self* refers to the instance of the TypeName on which the operation is applied. Another predefined parameter, *result*, denotes the return value of the operation if there is one. The dot notation is used to access the attributes of an object or to navigate from that object to the associated objects in a related type. The value of an accessed attribute or role in a postcondition is the value upon completion of the operation. To refer to the value of that property at the start of the operation, one has to postfix the property name with the keyword *@pre*.

As an example consider the diagram of Fig. 3.11 aimed at representing a set of web portals for selling the products of a company to a group of registered customers, some of them classified as gold customers. The model includes two textual integrity constraints and three operations. The invariant *minStock* ensures that all products have a stock of at least five units, while *salesAmount* imposes that gold customers must have paid a minimum amount of 100000 euros in sales. Regarding the operations, *newCustomer* and *addSaleLine* create a new customer and a new sale line in a sale, respectively. In OCL, the creation of an object is indicated with the operation *oclIsNew*. Operation *addSaleLine* also updates the stock of the product and the total amount of the sale. The operator *@pre* in *p.stock=p.stock@pre - quantity* indicates that the stock of the product has been decreased by *quantity* units with respect to the previous value. *RemoveGoldCustomer* converts a gold customer with no sales into a plain one.

3.5.2 List of Correctness Properties for Dynamic models

Pre and postconditions of declarative operations must be defined accurately, taking into account the possible interactions with the integrity constraints. For instance, preconditions which are too strong may prohibit the execution of an operation altogether (since none of the valid states of the system can satisfy the precondition). This section presents a list of properties to determine whether pre and postconditions are correctly defined.



context Product **inv** minStock: self.stock $\geq$ 5

context GoldCustomer **inv** salesAmount:
self.sale −>select(s | s.paid).cost −>sum() $\geq$ 100000

context Customer::newCustomer(name:String, p: Portal): Customer
post result.oclIsNew() and result.name=name and result.portal=p

context Sale::addSaleLine(p: Product, quantity: Integer): SaleLine
pre p.stock > 0
post result.oclIsNew() and result.sale=self and result.product=p and
result.quantity=quantity and p.stock=p.stock@pre-quantity and
self.amount=self.amount@pre + quantity*p.price

context Portal::removeGoldCategory(c: Customer)
pre c.oclIsTypeOf(GoldCustomer) and c.sale−>isEmpty()
post not c.oclIsTypeOf(GoldCustomer)

Figure 3.11: Running example: class diagram, OCL constraints and operations.

In the definition of the correctness properties, we will use the following notation. Given a model M, let S denote a *snapshot* of M, i.e. a possible instantiation of the types defined in M. A snapshot S will be called *legal*, denoted as $\mathsf{Inv}[S]$, if it satisfies all integrity constraints of M, including all textual OCL constraints.

Given a declarative operation *op*, $\mathsf{Pre}_{op}[o, P, S]$ denotes that the precondition of *op* holds when it is invoked over an object o of an snapshot S using the values in P as argument values for the list of parameters of *op*. For the sake of clarity, we will assume that o is passed as an additional parameter in P, e.g. the first one, expressing then the preconditions simply as: $\mathsf{Pre}_{op}[P, S]$. S and P will be referred collectively as the *input* of the operation.

To evaluate the postcondition, we also need to consider the return value and the snapshot after executing the operation (considering new/deleted objects and links, updated attribute values, etc.). The final snapshot and the return value will be referred as the *output* of the operation. Then, $\mathsf{Post}_{op}[P, S \triangleright S', R]$ will denote that the postcondition of operation *op* holds when S is the snapshot before executing the operation, S' is the snapshot after executing it, P is the list of parameters and R is the return value.

According to this notation, the list of properties is defined as follows:

- **Applicability**: An operation *op* is *applicable* if the precondition is satisfiable, i.e. if there is an input where the precondition evaluates to true.

$$\exists S : \exists P : \mathsf{Inv}[S] \ \wedge \ \mathsf{Pre}_{op}[P, S]$$

- **Redundant precondition**: The precondition of an operation *op* is *redundant* if it is true for any legal input.

$$(\exists S : \mathsf{Inv}[S]) \ \wedge \ (\forall S : \forall P : \mathsf{Inv}[S] \ \rightarrow \ \mathsf{Pre}_{op}[P, S])$$

- **Weak executability**: An operation *op* is *weakly executable* if the postcondition is satisfiable, that is, if there is a legal input satisfying the precondition for which we can find a legal output satisfying the postcondition.

$$\exists S, S' : \exists P : \exists R : \mathsf{Inv}[S] \ \wedge \ \mathsf{Inv}[S'] \ \wedge \ \mathsf{Pre}_{op}[P, S] \ \wedge \ \mathsf{Post}_{op}[P, S \triangleright S', R]$$

- **Strong executability**: An operation *op* is *strongly executable* if, for every legal input satisfying the precondition, there is a legal output that satisfies the postcondition.

$$\forall S : \forall P : \exists S' : \exists R : (\mathsf{Inv}[S] \wedge \mathsf{Pre}_{op}[P, S]) \ \rightarrow \ (\mathsf{Inv}[S'] \wedge \mathsf{Post}_{op}[P, S \triangleright S', R])$$

- **Correctness preserving**: An operation *op* is *correctness preserving* if, given a legal input, each possible output satisfying the postcondition is also legal.

$$\forall S, S' : \forall P : \forall R : (\mathsf{Inv}[S] \wedge \mathsf{Pre}_{op}[P, S]) \ \rightarrow \ (\mathsf{Post}_{op}[P, S \triangleright S', R] \ \rightarrow \ \mathsf{Inv}[S'])$$

- **Immutability**: An operation *op* is *immutable* if, for some input, it is possible to execute the operation without modifying the initial snapshot.

$$\exists S : \exists P : \exists R : \mathsf{Inv}[S] \ \wedge \ \mathsf{Pre}_{op}[P, S] \ \wedge \ \mathsf{Post}_{op}[P, S \triangleright S, R]$$

- **Determinism**: An operation *op* is *non-deterministic* if there is a legal input that can produce two different legal outputs, e.g. different result values or different final snapshots.

$$\begin{aligned} \exists S, S'_1, S'_2 : \exists P : \exists R_1, R_2 : \ & \mathsf{Inv}[S] \ \wedge \ \mathsf{Inv}[S'_1] \ \wedge \ \mathsf{Inv}[S'_2] \ \wedge \ \mathsf{Pre}_{op}[P, S] \ \wedge \\ & \mathsf{Post}_{op}[P, S \triangleright S'_1, R_1] \ \wedge \ \mathsf{Post}_{op}[P, S \triangleright S'_2, R_2] \ \wedge \\ & (\ (S'_1 \neq S'_2) \ \vee \ (R_1 \neq R_2) \ ) \end{aligned}$$

Studying these properties in the running example, we have, for instance, that the precondition of *addSaleLine* is redundant since it is subsumed by the integrity constraint *minStock*. Also, *addSaleLine* is weakly executable but not strongly executable: for those states where *p.stock-quantity*<5 the final state will violate the invariant *minStock*. The precondition of *removeGoldCategory*

is not applicable since constraint *salesAmount* forces all gold customers to be related to at least a sale. Finally, *newCustomer* is strongly executable but not correctness preserving as it might create a gold customer (instead of a plain one) with no sales, violating *salesAmount*.

It is important to remark that, in general, designers define underspecified postconditions [112]. This means that, given an operation contract, there are usually several final states that satisfy its postcondition. Therefore, most operation contracts will be flagged by our analysis as non-deterministic. To improve the accuracy of the results, designers may want to provide postconditions which are precise enough to characterize the exact set of desired final states. For basic postcondition expressions, an educated guess of the designer's intention can be inferred by analyzing the initial ambiguous postcondition [19, 26], and thus, it would be possible to automatically generate a set of additional conditions to define more precisely the desired final state. This is left as further work.

3.5.3 Verifying Operations with Constraint Programming

This subsection presents a systematic and automatic procedure to verify correctness properties of operation contracts using the constraint programming paradigm.

As before, the key idea of our approach is to translate the model, together with its integrity constraints, the desired correctness property and the operation to verify, into a CSP such that by checking whether the generated CSP has a solution we can determine if the operation satisfies the property. Both the translation procedure and the search of a solution for the CSP (performed using existing CSP solvers) are completely automatic and, therefore, all the verification process is transparent to the designer.

In short, with our translation procedure, the set of variables in the generated CSP characterize a possible snapshot of the model, i.e. the variable values represent the objects of the snapshot, their attributes values, their relations, etc. Its constraints ensure that the variable values (i.e. the snapshot) are consistent with the implicit structural UML constraints (e.g. all objects in a subtype must be also instance of its supertype), graphical constraints (e.g. multiplicities) and textual OCL constraints. Pre and postconditions of operations and correctness properties are translated as additional constraints.

Given this set of variables, domains and constraints, the final CSP is organized as a sequence of subproblems to be solved by the constraint solver in order to find a solution for the CSP, and thus, prove the desired correctness property. The exact combination of these subproblems in the CSP depends on the chosen property. For properties regarding the operation precondition, the resolution of the CSP first searches for a legal snapshot which satisfies the operation precondition (this, for instance, proves the *applicability* of the operation). If no solution is found, the solver concludes that the property is not satisfied. For properties involving postconditions, once we have a legal instance that satisfies the precondition, the solver must search for a second legal snapshot that satisfies the postcondition (see Figure 3.12). As we will see, for some properties we

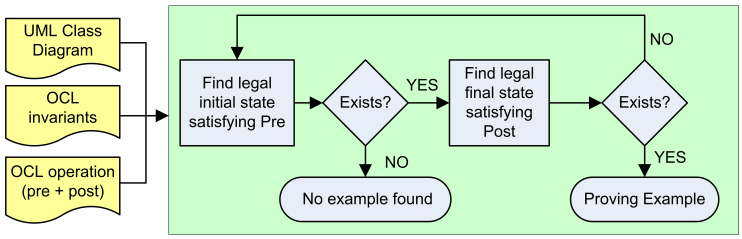


Figure 3.12: Analysis of the weak executabiliy property.

```
invariantMinStock(Snapshot) :-
    % Get the list of Objects in Snapshot of type Product
    getObjects(Snapshot, ''Product'', Objects),
    ( foreach(Object, Objects) do   % Iterate over all objects
      % Evaluate the invariant expression using this object as ''self''
      evalRootMinStock(Snapshot, [Object], Result),
      % The invariant must evaluate to true
      Result #=1).
```

```
evalRootMinStock( Snapshot, Vars, Result ) :-
    attribStock( Snapshot, Vars, X ), % X = attrib value
    const5( Snapshot, Vars, Y ),      % Y = constant
    #>=(X, Y, Result).                % Result = X >= Y
const5( _, _, Result ):- Result #= 5.
```

Figure 3.13: Translation of the invariant *minStock* (top) and some subexpressions (bottom).

will search for solutions that falsify the pre/postcondition expressions instead.

The encoding of the UML class diagram and the OCL constraints in the CSP is performed as shown in the previous section. Here we show how to encode as well the operations' pre and postconditions in the CSP and how they are combined, depending on the selected correctness property, to generate the final CSP that will be used to prove the property.

Translation of OCL operation contracts

Operations introduce new challenges in this translation: the list of parameters of the operation, the result value, and the complexity of studying two snapshots at once when analyzing postconditions.

Translation of preconditions

The boolean OCL expression of a precondition is basically translated following the same procedure explained above for the translation of invariant bodies. However, there are two differences regarding how and when the precondition expression is evaluated: the *parameters* and the *quantification*.

In the analysis of a precondition, it is necessary to consider the possible value of the operation parameters. For parameters of a basic type (integer,

float, boolean, string) designers must define their possible finite domain, for instance defining a lower and upper bound. Parameters whose type is one of the classes of the model (as the *self* parameter) can only refer to an object existing in the snapshot, so their value is already constrained by the valid instances of the snapshot where the operation is invoked. When evaluating a precondition, parameters become additional variables of the CSP, and their values are discovered by the solver as a part of the search for a solution to the CSP. For instance, when checking the applicability of an operation, the solver will automatically try several possible combinations of parameter values until it finds a combination (if any) that satisfies the Prolog rule generated for the precondition.

Contrary to invariants, properties on preconditions only require to find a single combination of a valid state and a possible assignment for the operation parameters that satisfy the precondition. Therefore, preconditions will be translated into a rule which simply evaluates the precondition body, invoking the rule for the topmost operator. To ensure that the rules for the precondition body have access to all parameter values during the rule evaluation, the list of visible variables for these rules (second argument of the Prolog rule) is initialized with the list of parameter values. In this way, accessing a parameter within the expression is equivalent to accessing any other variable: the rule only needs to be aware of the position of each parameter in the variables list. As an example, the precondition rule for *addSaleLine* will be defined as follows:

```
preconditionAddSaleLine(Snapshot, Parameters, Result) :-
    % Result = truth value of evaluating the precondition
    evalRootExpr(Snapshot, Parameters, Result).
```

where *evalRootExpr* represents the rule for the root node of the precondition expression. The output *Result* value, reporting whether the given input (i.e. the *self* object plus the other parameters) satisfies the precondition, will be used later on to determine the satisfaction of correctness properties for the operation.

Translation of postconditions

Two new factors in the translation of postconditions are the return value and the relationship between the two snapshots representing the initial and final states.

In our translation, the return value will simply become another variable in the list of visible variables, just like *self* or the other parameters in the precondition.

Relationships between the initial and the final state are expressed by means of the *oclIsNew* and, specially, the *@pre* OCL operators. *OclIsNew* highlights that an object should exist in the final state but not in the initial one; and *@pre* is used to retrieve the value of a subexpression in the initial state. Thus, the Prolog implementation of these two operators needs to receive an additional argument: the snapshot for the initial state. To avoid changing the general rule pattern due to this extra argument, this initial state is stored in the global variable `initialstate`. This variable will be conveniently accessed within the subrules for these two operators. Translation of all other OCL operators in the postcondition expression is not changed from previous translations steps. They

```
:- local reference(initialstate).
postconditionAddSaleLine(InitialState, FinalState,
                         Parameters, RetValue, Result) :-
    % Add the return value and parameters to the list of visible vars
    append([RetValue], Parameters, Variables),
    % Store the initial state, needed in oclIsNew and @pre nodes
    setval(initialstate, InitialState),
    % Result = truth value of evaluating the postcondition
    evalRootExpr(FinalState, Variables, Result).
```

Figure 3.14: Translation of the OCL postcondition of operation *addSaleLine*.

```
ocl_isNew(FinalState, Oid, TypeName, Result) :-
    % Recover the initial state from the global variable
    getval(initialstate, InitState),
    % Get the list of objects before and after the operation
    getObjects(InitState, TypeName, ObjectsBefore),
    getObjects(FinalState, TypeName, ObjectsAfter),
    % Check if Oid exists before/after the operation
    existsObjectWithOid(ObjectsBefore, Oid, ExistsBefore),
    existsObjectWithOid(ObjectsAfter, Oid, ExistsAfter),
    % Result = ExistsAfter and not ExistsBefore
    and( ExistsAfter, neg ExistsBefore, Result).
```

Figure 3.15: Translation of the OCL operator *oclIsNew*.

are just evaluated on the particular snapshot given as argument to their Prolog rule, it does not matter if it represents the initial or the final state.

To sum up, the definition of the rule for the postcondition of the operation *addSaleLine* is shown in Figure 3.14. The *initialstate* variable will then be used in the rules evaluating *oclIsNew* and *@pre* nodes appearing in postcondition expressions. We provide the rule for *oclIsNew* as an example in Figure 3.15. It determines if the object with the *Oid* value given as an argument is an object that did not exist before executing the operation.

Translation of correctness properties

As a last step, each correctness property (or its negation) is translated as a new CSP constraint restricting the *result* values returned by the pre and postcondition rules such that finding a solution to the CSP with this new constraint suffices to prove the property.

Whether to use the property or its negation depends on the quantification used in the property formalization, *existential* or *universal* (see Section 3.5.2). Existentially quantified properties can be *proved* by finding an *example*, i.e. a case where the property is satisfied. For example, applicability can be proved by finding a legal input that satisfies the precondition. Universally quantified properties can be *disproved* by finding a *counterexample*. For instance, redundancy can be disproved by finding a legal snapshot that does not satisfy the

```
weakExecutabilityAddSaleLine(Example) :-
    Example = [InitState, FinalState, Parameters, RetValue],
    findInitialState(InitState, Parameters),
    findFinalState(InitState, FinalState, Parameters, RetValue).
findInitialState(InitState, Parameters) :-
    % Definition of variables, domains, graphical integrity constraints
    % Textual integrity constraints
    invariantMinStock(InitState), invariantSalesAmount(InitState),
    % Precondition
    preconditionAddSaleLine(InitState, Parameters, ResultOfPre),
    ResultOfPre #= 1, % Weak executability
    % Now find a solution satisfying all these constraints
    labeling([InitState, Parameters]).
findFinalState(InitState, FinalState, Parameters, RetValue) :-
    % Definition of variables, domains, graphical integrity constraints
    % Textual integrity constraints
    invariantMinStock(FinalState), invariantSalesAmount(FinalState),
    % Postcondition
    postconditionAddSaleLine(InitState, FinalState, Parameters,
                             RetValue, ResultOfPost),
    ResultOfPost #= 1, % Weak executability
    % Now find a solution satisfying all these constraints
    labeling([FinalState, RetValue]).
```

Figure 3.16: CSP generated for checking weak satisfiability of *addSaleLine*. The *labeling* operator is a possible backtracking implementation offered by the constraint solver that attempts to assign values to the given list of input variables. If the assignment does not satisfy all the stated CSP constraints preceding the labeling, a new assignment is tried until the solver finds a solution or determines that no solution exists.

precondition Similarly, the lack of (counter)examples can be used to (dis)prove the property.

The selected property also influences how the final CSP is organized as a combination of the rule excerpts generated during the previuos translation steps. For properties on preconditions, postcondition rules are not included. For properties on postconditions, the CSP is split up into two subproblems (see Figure 3.12). The first one (*findInitialState*) tries to find a legal snapshot that satisfies the precondition rule. This initial snapshot is then given as an argument to the second subproblem (*findFinalState*), in charge of finding a second legal snapshot satisfying (or not) the postcondition to prove the property. As an example, Figure 3.16 sketches the final CSP to determine whether *addSaleLine* is weakly executable. Other properties imply adding new constraints/subproblems to the CSP. For instance, immutability requires a new constraint imposing the equality between the initial and final states.

3.6 Verification and Validation of Declarative Model-to-Model Transformations Through Invariants

The same principles developed for the verification of UML/OCL models can be adapted to the verification of model transformations, the other key elemenet in any MDE approach.

There are two main approaches to M2M transformation: *operational* and *declarative*. The former is based on rules or instructions that explicitly state how and when creating the elements of the target model from elements of the source one. Instead, in declarative approaches, some kind of visual or textual patterns describing the relations between the source and target models are provided, from which operational mechanisms are derived e.g. to perform forward and backward transformations. These declarative patterns are complemented with additional information to express relations between attributes in source and target elements, as well as to constrain when a certain relation should hold. The Object Constraint Language (OCL) standard [83] is frequently used for this purpose [85].

The increasing complexity of modelling languages, models and transformations makes urgent the development of techniques and tools that help designers to assure transformation correctness. Whereas several notations have been proposed for specifying M2M transformations in a declarative way [1, 65, 85, 96], there is a lack of methods for analysing their correctness in an integral way, taking into account the relations expressed by the transformation, as well as the meta-models and their well-formedness rules.

In this section we propose verification and validation techniques for M2M transformations based on the analysis of a set of OCL invariants automatically derived from the declarative description of the transformations. These invariants state the conditions that must hold between a source and a target model in order to satisfy the transformation definition, i.e. in order to represent a valid mapping. We call these invariants, together with the source and target meta-models, a *transformation model* [17]. To show the wide applicability of the technique, we have studied how to create this transformation model from two prominent M2M transformation languages: Triple Graph Grammars (TGGs) [96] and QVT [85]. In this section we will limit the description to the TGG case. See [28] for details on the QVT part.

Once the transformation model is synthesized, we can determine several correctness properties of the transformation by analysing the generated transformation model with any available tool for the verification of static UML/OCL class diagrams (see [4, 23, 91, 99]) though, obviously, we will use our own UMLtoCSP approach for that. In particular, we have predefined a number of verification properties in terms of the extracted invariants, which provide increasing confidence on the transformation correctness. For example, we can check whether a relation or the whole transformation is applicable in the forward direction (i.e., whether there is a source model enabling a relation), forward weak executable

(if we can find a pair of source and target models satisfying the relation and the meta-model constraints), forward strong executable (if a relation is satisfied whenever it is enabled), or total (whether all valid source models can be transformed).

The transformation model can also be used for validation purposes. Given the transformation model, tools like UMLtoCSP can be used to automatically generate valid pairs of source and target models, or a valid target model for a given or partially specified source model. These generated pairs help designers in deciding whether the defined transformation reflects their intention, thus helping to uncover transformation defects. Additionally, we have devised heuristics to partially automate the validation process by means of generating potentially relevant scenarios (representing corner cases of the transformation) that the designer may be specially interested in reviewing. Refer to [28] for details on our validation strategies.

3.6.1 Triple Graph Grammars

Triple Graph Grammars (TGGs) [96] were proposed by A. Schürr as a formal means to specify transformations between two languages in a declarative way. TGGs are founded on the notion of graph grammar [95]. A graph grammar is made of rules having graphs in their left and right hand sides (LHS and RHS), plus the initial graph to be transformed. Applying a rule to a graph is only possible if an occurrence of the LHS (a *match*) is found in it. Once such occurrence is found, it is replaced by the RHS graph. This is called *direct derivation*. It may be possible to find several matches for a rule, and then one is chosen at random. The execution of a grammar is also non-deterministic: at each step, one rule is randomly chosen and its application is tried. The execution ends when no rule can be applied.

Even though graph grammar rules rely on pre- and post-conditions, and on pattern matching, when used for model-to-model transformation, they have an operational, unidirectional style, as the rules specify how to build the target model assuming the source already exists. On the contrary TGGs are declarative and bidirectional since, starting from a unique TGG specifying the synchronized evolution of two graphs, it is possible to generate forward and backward transformations as well as operational mechanisms for other scenarios [69].

TGGs are made of rules working on triple graphs. These are made of two graphs called *source* and *target*, related through a *correspondence* graph. Any kind of graph can be used for these three components, from standard unattributed graphs ($V; E; s, t\colon E \rightarrow V$) to more complex attributed graphs (e.g., E-graphs [47]). The nodes in the correspondence graph (the *mappings*) have morphisms[8] to the nodes in the source and target graphs. Triple morphisms are defined as three graph morphisms that preserve the correspondence functions. They are used to relate the LHS and RHS of a TGG rule, to identify a match of the LHS in a graph, and to type a triple graph.

[8] A morphism corresponds to the mathematical notion of total function between two sets, or in general between two structures (graphs, triple graphs, etc.)

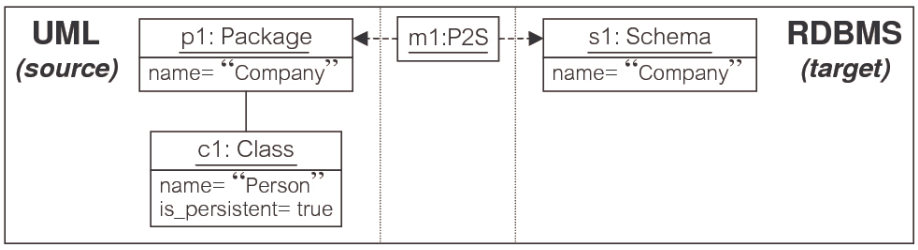


Figure 3.17: A triple graph example

Definition 1 (Triple Graph and Morphism) *A triple graph* $TrG = (G_s, G_c, G_t, cs\colon V_{G_c} \to V_{G_s}, ct\colon V_{G_c} \to V_{G_t})$ *is made of two graphs* G_s *and* G_t *called source and target, related through the nodes of the correspondence graph* G_c.

A triple graph morphism $f = (f_s, f_c, f_t) : TrG^1 \to TrG^2$ *is made of three graph morphisms* $f_x\colon G_x^1 \to G_x^2$ *(with* $x = \{s, c, t\}$*) such that the correspondence functions are preserved.*

In the previous definition, V_{G_x} is the set of nodes of graph G_x. Morphisms cs and ct relate two nodes x and y in the source and target graphs iff $\exists n \in V_{G_c}$ with $cs(n) = x$ and $ct(n) = y$. We often depict a triple graph by $\langle G_s, G_c, G_t \rangle$, and use TrG_x (for $x = \{s, c, t\}$) to refer to the x component of TrG. In this way, $\langle G_s, G_c, G_t \rangle_s = G_s$.

Fig. 3.17 shows a triple graph, taken from the class-to-relational transformation [85], which we use as a running example. The source graph is a class diagram with a package and a class, the target one is a relational schema model with one schema node, and the correspondence includes a mapping between the package and the schema. Note that "source" and "target" are relative terms, as we could also use source for the relational schema and target for the class diagram.

A triple graph is typed by a *meta-model triple* [56] or TGG schema, which contains the source and target meta-models and declares allowed mappings between both. Fig. 3.18 shows the meta-model triple for our running example. The correspondence meta-model declares five classes: `P2S` maps packages and schemas, `A2Co` maps attributes and columns, and `CT` and its specializations `C2T` and `C2TCh` relate classes and tables. In particular `C2TCh` is used to relate a children class with the table associated to its parent class. The dotted arrows specify the allowed morphisms from the correspondence to the source and target models, and can be treated as normal associations with cardinality 1 on the side of the source/target class. The meta-model includes OCL constraints ensuring uniqueness of attribute names for each class and table, as well as same persistence for a class and its children. As an example, the triple graph in Fig. 3.17 conforms to the meta-model in Fig. 3.18.

A typed triple graph is formally represented as $(TrG, type\colon TrG \to MM)$, where the first element is a triple graph and the second a morphism to the meta-model triple. Morphisms between typed triple graphs must respect the typing

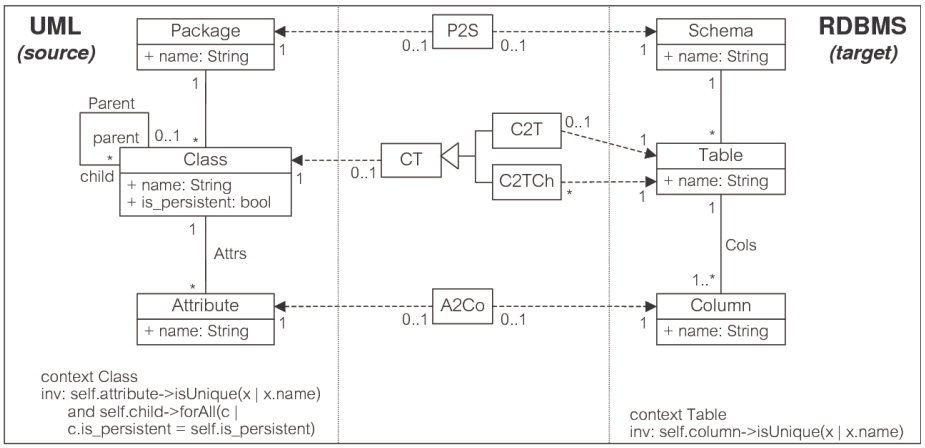


Figure 3.18: Example meta-model triple

morphism and can take inheritance into account, as in [56]. For simplicity of presentation, we omit the typing in the following definitions.

Besides a meta-model triple, a M2M transformation by TGGs consists of a set of declarative rules that describe the synchronized evolution of two models. Rules have triple graphs in their LHS and RHS and may include OCL attribute conditions. This contrasts with the usual approach of using attribute computations in the rules instead of conditions [47]. We use the latter as it poses some benefits that will be shown later on when operationalising the rules. Declarative rules are non-deleting because they describe how models are created, hence they are defined by an injective triple morphism.

Definition 2 (Declarative TGG Rule) *A declarative TGG rule* $p = (r\colon L \rightarrow R, ATT_{COND})$ *is made of two triple graphs,* $L = \langle L_s, L_c, L_t \rangle$ *and* $R = \langle R_s, R_c, R_t \rangle$*, an injective triple morphism* r *between* L *and* R*, and a set* ATT_{COND} *of OCL constraints over* R*, expressing attribute conditions.*

Fig. 3.19 shows four example TGG declarative rules using a compact notation that presents together L and R. The elements created by the rules (R-L) are marked as {`new`}, and the preserved elements are untagged. As an example, rule `Class-Table` is shown in the upper row first in extended and then in compact notation.

Rule `Package-Schema` declares that every time a package is created, a schema with the same name is created simultaneously, and vice versa. Rule `Class-Table` specifies that creating a persistent class in a package already related to a schema should create a table with the same name in that schema, and vice versa. In this case the attribute condition demands the class to have no parent. Note that we do not demand the LHS/RHS of rules to satisfy the integrity constraints of the meta-model. For example the RHS of the rule `Class-Table` is not a valid model because, according to the meta-model in Fig. 3.18, each table should

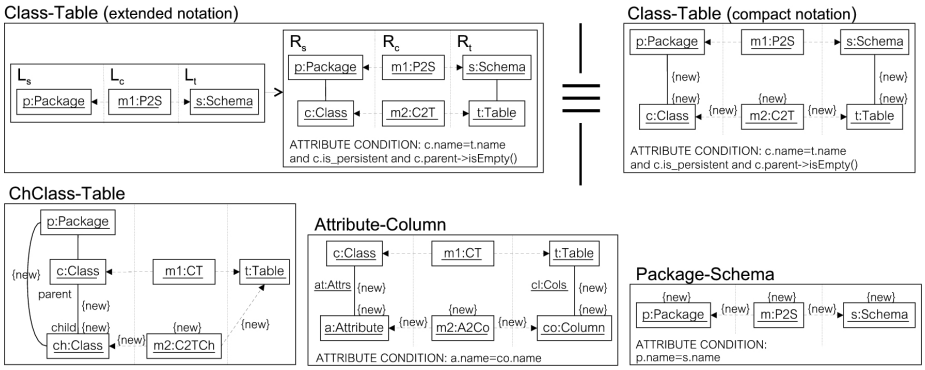


Figure 3.19: Some declarative TGG rules for the class-to-relational transformation.

be connected to at least one column. When executing a transformation it is acceptable to go through some intermediate models that are inconsistent with respect to the meta-model's constraints. What is important is that the final models are consistent.

For non top-level classes, the rule `ChClass-Table` is used instead of rule `Class-Table`. This rule specifies that creating a child class of a class already related to a table should map the child class to the same table. Finally, `Attribute-Column` synchronously creates attributes and columns for classes related to tables.

A TGG is bidirectional as rules do not specify any direction, but synchronously create and relate source and target elements. A TGG defines the language of all triple graphs that satisfy the meta-model constraints and that can be derived using zero or more applications of the grammar rules. Please note that some derived graphs may not conform to the meta-model, and hence are not part of the language.

In practice, one does not use declarative TGG rules to create source and target models at the same time, as it would require a synchronous coupling of both models. Instead, so-called operational rules are derived for different tasks, e.g. to perform forward (source-to-target) and backward (target-to-source) transformations. A forward transformation creates a set of target elements that correspond to a given set of initial source model elements, and conversely with a backward transformation. The algorithm to derive such rules was proposed in [96] (see also [69] for the description of operational rules for other purposes). Next we present an extension of the algorithm that handles OCL attribute conditions. We will use this definition in order to derive the OCL invariants in the next section.

Definition 3 (Operational TGG Rule) *Given the declarative TGG rule* $p = (r\colon \langle L_s, L_c, L_t\rangle \rightarrow \langle R_s, R_c, R_t\rangle, ATT_{COND})$*, the following operational rules can be derived:*

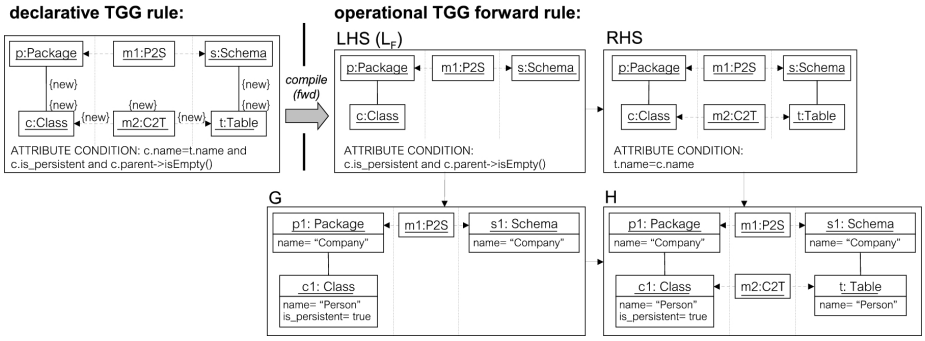


Figure 3.20: Derivation by operational TGG forward rule

- *Forward:* $\overrightarrow{p} = (r' \colon \langle R_s, L_c, L_t\rangle \rightarrow \langle R_s, R_c, R_t\rangle, \overrightarrow{ATT}_{LHS}, \overrightarrow{ATT}_{RHS})$.
- *Backward:* $\overleftarrow{p} = (r' \colon \langle L_s, L_c, R_t\rangle \rightarrow \langle R_s, R_c, R_t\rangle, \overleftarrow{ATT}_{LHS}, \overleftarrow{ATT}_{RHS})$.

where $\overrightarrow{ATT}_{LHS}$ *(resp.* $\overleftarrow{ATT}_{LHS}$*) contains the part of the* ATT_{COND} *OCL expression concerning elements of the LHS of the forward (resp. backwards) operational rule only.* $\overrightarrow{ATT}_{RHS}$ *(resp.* $\overleftarrow{ATT}_{RHS}$*) contains the part of* ATT_{COND} *not included in* $\overrightarrow{ATT}_{LHS}$ *(resp.* $\overleftarrow{ATT}_{LHS}$*).*

The operational rules enforce the pattern given by the declarative rule, thus their RHS is equal to the RHS of the declarative rule. In the forward case, the LHS assumes that the source graph already exists, whereas in the backward case the existence of the target graph is assumed. In the rest of the section, we use L_F and L_B to refer to the LHS of the forward and backward rules. The conditions in ATT_{COND} are split in those to be checked on the LHS before rule application ($\overleftarrow{ATT}_{LHS}$ and $\overrightarrow{ATT}_{LHS}$) and those to be checked after rule application ($\overleftarrow{ATT}_{RHS}$ and $\overrightarrow{ATT}_{RHS}$).

As an example, the upper row of Fig. 3.20 shows the operational forward rule derived from the declarative rule `Class-Table`. The forward rule assumes that the package `p` has a class `c` and is related to a schema `s`, and then creates a new table. The figure shows the application of the rule to the graph of Fig. 3.17 resulting in a triple graph H that contains a newly created table in its target. Note that this resulting target graph does not satisfy the meta-model constraints because each table needs to have at least one column. Thus, one can infer that the transformation is not total, as classes without attributes cannot be transformed into a valid model. This simple example shows the necessity of providing automatic means to check properties of rules and transformations.

Our definition of declarative rule does not use attribute computations as usual in the literature [47], but declarative attribute conditions ATT_{COND}. The advantage is that no algebraic manipulation is needed when generating the operational rules, but just to split the original conditions in those to be

checked in the LHS ($\overleftarrow{ATT}_{LHS}$, $\overrightarrow{ATT}_{LHS}$) and the RHS ($\overleftarrow{ATT}_{RHS}$, $\overrightarrow{ATT}_{RHS}$). When this is implemented in practice, we can use a constraint solver to resolve attribute values. Previous approaches perform algebraic manipulation so that the attribute values for the created objects could be calculated from the ones in the LHS. For instance, in the presented example, we would have had to assign the name of the class to the name of the newly created table. Although in this case it is just an assignment, in general such algebraic manipulations present practical problems because they are difficult to automate. Note however that relying purely on constraint solving at the operational level may present computational efficiency problems in some cases. Of course, there are several tools and approaches that allow embedding OCL in normal graph grammar rules (i.e., not in TGG rules), like VMTS [72] and Fujaba [98], to express attribute conditions and computations.

Next section shows how we avoid algebraic manipulation of attribute expressions by compiling the declarative TGG rules into OCL invariants (instead of into operational rules) and using a constraint solver to actually perform and analyse the transformation. For this purpose, rules are interpreted as constraints (similar to [42]) or invariants that a pair of models should satisfy.

3.6.2 Extracting OCL Invariants from Declarative TGG Rules

Our verification approach for M2M declarative transformations is based on the analysis of the transformation model [17] derived from the transformation specification. The transformation model is made of the source and target metamodels (that can be easily expressed as UML class diagrams) plus the set of invariants that must hold between the source and target models in order to satisfy the transformation definition. These invariants must guarantee that the target model is a valid transformation of the source according to the set of TGG rules, and similar for the target.

In this section we present a procedure that creates invariants capturing the semantics of the TGG rules. This procedure can be regarded as a M2M transformation itself between the TGG and UML/OCL metamodels.

The invariants must ensure that each rule p is satisfied in the model. Hence, we introduce two concepts: rule *enabledness* and rule *satisfaction*. Intuitively, a declarative rule is source-enabled (resp. target-enabled) in a given graph if there exists some match of the LHS of its associated forward operational rule (resp. backward rule) in the graph.

Definition 4 (Enabledness of Rule) *Given the declarative TGG rule* $p = (r\colon \langle L_s, L_c, L_t\rangle \rightarrow \langle R_s, R_c, R_t\rangle, ATT_{COND})$ *and a triple graph* G*:*

- p *is source-enabled if* $\exists m\colon L_F \rightarrow G$*, and* $m(L_F)$ *satisfies* $\overrightarrow{ATT}_{LHS}$ *assuming the identification of objects and links induced by* m *and using* G *as context. Given a morphism* m*, we write* $G \vdash_{m,F} p$ *if* m *enables* p *source-to-target in* G*.*

- *p is target-enabled if $\exists m\colon L_B \to G$, and $m(L_B)$ satisfies $\overleftarrow{ATT}_{LHS}$ assuming the identification of objects and links induced by m and using G as context. Given a morphism m, we write $G \vdash_{m,B} p$ if m enables p target-to-source in G.*

As an example, the declarative rule `Class-Table` in Fig. 3.20 is source-enabled in triple graph G because there is an occurrence of the LHS of its operational forward rule in G. On the contrary, the rule is not target-enabled because the LHS of the operational backward rule would have needed a table in the target graph to be matched. Hence we have $G \vdash_{m,F}$ `Class-Table` and $G \nvdash_{m,B}$ `Class-Table`. However the rule is source- and target-enabled in H.

Satisfaction of a rule involves checking both forward and backward satisfaction and is useful to ensure that two models are actually synchronized according to the rule. Intuitively, forward (resp. backward) satisfaction requires that the target (source) model satisfies the RHS of the rule for each match where the rule is source- (target-) enabled. We will use the notion of satisfaction in our algorithm to generate invariants.

Definition 5 (Satisfaction of Rule) *Given the declarative TGG rule $p = (r\colon \langle L_s, L_c, L_t\rangle \to \langle R_s, R_c, R_t\rangle, ATT_{COND})$ and a triple graph G:*

- *p is forward-satisfied in G, written $G \models_F p$, if $\forall m\colon L_F \to G$ s.t. $G \vdash_{m,F} p$, then $\exists m'\colon R \to G$ with $m = m' \circ r$ s.t. $m'(G)$ satisfies ATT_{COND} assuming the identification of objects and links induced by m and using G as context.*

- *p is backward-satisfied in G, written $G \models_B p$, if $\forall m\colon L_B \to G$ s.t. $G \vdash_{m,B} p$, then $\exists m'\colon R \to G$ with $m = m' \circ r$ s.t. $m'(G)$ satisfies ATT_{COND} assuming the identification of objects and links induced by m and using G as context.*

- *p is satisfied in G, written $G \models p$, if $G \models_F p \wedge G \models_B p$*

Thus, a rule is forward-satisfied, if for each morphism where the rule is source-enabled, there is an occurrence m' of the RHS which preserves the identification of objects and links ($m = m' \circ r$) and satisfies the OCL constraints in ATT_{COND}. As an example, rule `Class-Table` in Fig. 3.20 is not forward-satisfied in G, but it is in H. The rule is backward-satisfied in both G and H, in the first case *trivially* because the rule is not target-enabled. As we have that $H \models_F$ `Class-Table` and $H \models_B$ `Class-Table` then we have $H \models$ `Class-Table`, or in other words, H contains two synchronized models according to `Class-Table`.

In terms of the previous definitions, the invariants to be extracted for each rule p are responsible for:

a) Locating each occurrence where p is source-enabled (see Definition 4).

b) Locating each occurrence where p is target-enabled (see Definition 4).

c) Ensuring that the elements of each occurrence found in a) and b) are connected to mapping objects according to the RHS of p.

d) Ensuring that the mapping objects connect elements that satisfy p (see Definition 5).

Next we describe our extraction procedure and the structure of the generated invariants. Our procedure makes two assumptions: (i) all rules create at least one element in the correspondence graph and (ii) each type of mapping is created by at most one rule. Given a rule, the first two steps in the procedure (see next definition) add invariants to every node n in the source or target graphs of the RHS that is connected to a newly created correspondence node m. This corresponds to the items a), b) and c) in the previous list. Step 3 in the procedure adds the invariant to every correspondence node m created by the rule (item d)).

Definition 6 (Invariant Extraction) *Given a declarative TGG rule* $p = (r\colon \langle L_s, L_c, L_t\rangle \rightarrow \langle R_s, R_c, R_t\rangle, ATT_{COND})$*:*

1. $\forall n \in V_{R_s}$ *s.t.* $\exists m \in V_{R_c} - r(V_{L_c})$ *with* $cs(m) = n$*, add an invariant named* `p` *to* $type(n)$*.*
2. $\forall n \in V_{R_t}$ *s.t.* $\exists m \in V_{R_c} - r(V_{L_c})$ *with* $ct(m) = n$*, add an invariant named* `p` *to* $type(n)$*.*
3. $\forall m \in V_{R_c} - r(V_{L_c})$*, add an invariant named* `p` *to* $type(m)$*.*

Invariant `p` in the source checks that for each occurrence where the rule is source-enabled, the rule is satisfied. According to Definition 4, the rule is source-enabled it there exists an occurrence of the LHS of the forward rule L_F satisfying the terms in $\overrightarrow{ATT}_{LHS}$. This is actually checked by a helper query operation named `p-enabled(...)`. If such query operation returns true, then the invariant `p` ensures that this occurrence is connected to all required objects needed to satisfy the rule. This is performed by a helper operation `p-mapping(...)` placed in the created correspondence node. This operation checks the object graph satisfies the structure of the RHS and the OCL constraints in ATT_{COND}, similar to Definition 5. Symmetrically, the invariant in the target ensures that each occurrence where `p` is target-enabled satisfies the rule. As both invariants are similar, we only show the structure of `p` for the source elements.

Definition 7 (Invariant for Source Elements) *Given a declarative TGG rule* $p = (r\colon \langle L_s, L_c, L_t\rangle \rightarrow \langle R_s, R_c, R_t\rangle, ATT_{COND})$*, then* $\forall n \in V_{R_s}$ *s.t.* $\exists m \in V_{R_c} - r(V_{L_c})$ *with* $cs(m) = n$*, the following invariant is generated:*

context *type(n)* ***inv*** *p:*
$type(n_i) :: allInstances()->forAll(n_i|$
$type(n_j) :: allInstances()->forAll(n_j|...$ $\Big\}\ \forall n_k \in V_{L_F} - \{n\}$
if self.p-enabled$(n_i, n_j, ...)$ *then*
$type(n_u) :: allInstances() -> exists(n_u|$
$type(n_v) :: allInstances() -> exists(n_v|...$ $\Big\}\ \forall n_w \in V_R - r(V_{L_F})$
$type(m) :: allInstances() -> exists(m|...$
$m.p\text{-}mapping(n_i, n_j, ..., n_u, n_v, ...)$ *endif...*))...))

context *type(n)::p-enabled*$(n_i :\ type(n_i),\ n_j :\ type(n_j), \ldots)$
body $n_i.role_j->includes(n_j)$
and...
...and $\overrightarrow{ATT}_{LHS}$ $\Big\}\ \forall e \in E_{L_F}\ s.t.\ n_i \overset{s}{\leftarrow} e \overset{t}{\rightarrow} n_j$

where E_{L_F} *is the set of edges in* L_F*,* $role_j$ *is the role in the meta-model that allows navigating from* n_i *to* n_j*. If some edge has* n *as source or target we use the reserved word* $self$ *to refer to* n *in the expression.*

Note that the query operation `p-enabled` receives as parameters the objects in L_F, and then checks that they are connected according to L_F and that they satisfy $\overrightarrow{ATT}_{LHS}$. If association end $role_j$ has cardinality 1, then we do not use $n_i.role_j->includes(n_j)$ but simply $n_i.role_j = n_j$. The invariant for the target elements is generated in the same way, but considering nodes $n \in V_{R_t}$ and then traversing the graph L_B. For nodes created in the correspondence graph, invariants are generated as follows.

Definition 8 (Invariant for Mappings) *Given* $p = (r\colon \langle L_s, L_c, L_t\rangle \rightarrow \langle R_s, R_c, R_t\rangle, ATT_{COND})$*, then* $\forall n \in V_{R_c} - r(V_{L_c})$ *the following invariant is generated:*

context *type(n)* ***inv*** *p:*
$type(n_i) :: allInstances()->exists(n_i|$
$type(n_j) :: allInstances()->exists(n_j|...$ $\Big\}\ \forall n_k \in V_R - \{n\}$
$n_i.role_j->includes(n_j)$ *and...*
...and $self.p\text{-}mapping(n_i, n_j, ...)...)...)$ $\Big\}\ \forall e \in r(E_L)\ s.t.\ n_i \overset{s}{\leftarrow} e \overset{t}{\rightarrow} n_j$

---

context *type(n)::p-mapping*$(n_i :\ type(n_i),\ n_j :\ type(n_j), \ldots)$
body $n_i.role_j->includes(n_j)$
and...
...and ATT_{COND} $\Big\}\ \forall e \in E_R - r(E_L)\ s.t.\ n_i \overset{s}{\leftarrow} e \overset{t}{\rightarrow} n_j$

Note that the main body of the invariant checks the existence of the node in the RHS and the edges in the LHS. Then, the query operation checks the existence of the remaining edges in the RHS and the conditions in ATT_{COND}.

Let us consider the example rules in Fig. 3.19. From rule `Class-Table` we generate invariants for the class, the table and the `C2T` mapping, because the latter is created and connected to the class and the table. Source-enabledness is checked by the class's `Class-Table-enabled` operation, whereas actual satisfaction is checked by the `Class-Table-mapping` operation in the correspondence

node. A similar invariant for the table ensures that whenever the rule is target-enabled, it is actually satisfied.

context Class **inv** Class-Table:
$Package :: allInstances()->forAll(p|$
$\quad P2S :: allInstances()->forAll(m1|$
$\quad\quad Schema :: allInstances()->forAll(s|$
$\quad\quad\quad if\ self.Class\text{-}Table\text{-}enabled(p, m1, s)\ then$
$\quad\quad\quad\quad Table :: allInstances()->exists(t|$
$\quad\quad\quad\quad\quad C2T :: allInstances()->exists(m2|$
$\quad\quad\quad\quad\quad\quad m2.Class\text{-}Table\text{-}mapping(p, m1, s, self, t)))\ endif)))$

---

context Class::Class-Table-enabled(p:Package, m1:P2S, s:Schema)
body $self.package = p\ and\ m1.package = p\ and\ m1.schema = s$
$\quad and\ self.is\_persistent\ and\ self.parent->isEmpty()$

---

context C2T **inv** Class-Table:
$Package :: allInstances()->exists(p|$
$\quad P2S :: allInstances()->exists(m1|$
$\quad\quad Schema :: allInstances()->exists(s|$
$\quad\quad\quad Table :: allInstances()->exists(t|$
$\quad\quad\quad\quad Class :: allInstances()->exists(c|$
$\quad\quad\quad\quad m1.package = p\ and\ m1.schema = s$
$\quad\quad\quad\quad and\ self.Class\text{-}Table\text{-}mapping(p, m1, s, c, t))))))$

---

context C2T::Class-Table-mapping(p:Package, m1:P2S, s:Schema,
c:Class, t:Table)
body $c.name = t.name\ and\ t.schema = s\ and\ c.package = p\ and\ self.class = c$
$\quad and\ self.table = t\ and\ c.is\_persistent\ and\ c.parent->isEmpty()$

---

context Table **inv** Class-Table:
$Package :: allInstances()->forAll(p|$
$\quad P2S :: allInstances()->forAll(m1|$
$\quad\quad Schema :: allInstances()->forAll(s|$
$\quad\quad\quad if\ self.Class\text{-}Table\text{-}enabled(p, m1, s)\ then$
$\quad\quad\quad\quad Class :: allInstances()->exists(c|$
$\quad\quad\quad\quad\quad C2T :: allInstances()->exists(m2|$
$\quad\quad\quad\quad\quad\quad m2.Class\text{-}Table\text{-}mapping(p, m1, s, c, self)))\ endif)))$

---

context Table::Class-Table-enabled(p:Package, m1:P2S, s:Schema)
body $self.schema = s\ and\ m1.package = p\ and\ m1.schema = s$

The generated invariant in the `Class` checks that if there is an occurrence of L_F, then there must exist a table such that the rule conditions are satisfied. The occurrence of L_F is sought by the three first nested *forAll*, which iterate to look for a package `p`, a schema `s` and a correspondence node `m1`. These elements should be connected according to L_F, and satisfy the constraints in $\overrightarrow{ATT}_{LHS}$, what is checked by the operation *Class-Table-enabled*. If such operation returns true, the invariant looks for a table `t` and a mapping `m2` connected as specified

by the RHS of the rule and satisfying ATT_{COND}, what is checked by operation *Class-Table-mapping* in the mapping class `C2T`. Symmetrically, the invariant in the `Table` checks that if there is an occurrence of L_B, there is a class satisfying the rule. The invariants extracted from the other rules are shown in the Appendix.

Note that, using invariants, it is difficult to express the fact that a new table has to be created for each occurrence of L_F. Instead, we just say that a table should exist, and rely on the mapping cardinalities to ensure that indeed one is created for each class. Should we have assigned a wrong cardinality $*$ (instead of 0..1) between `C2T` and `Table`, the generated invariants would allow two classes with the same name to be related to the same table. However this is not what rule `Class-Table` expresses, which demands two different tables. By defining the right cardinality 0..1 we implement a correct translation, as each table is related to at most one `C2T` mapping, and this to exactly one class, thus ensuring that each class is related to at most one table.

We would like to remark that, both, the number of extracted invariants and the internal complexity of each invariant, are linear with respect to the number of TGG rules. Therefore, the extraction process can deal with TGGs of any size.

3.6.3 Verification of Model-to-Model Transformations

The verification of transformations answers the question "*is the transformation right?*", i.e. are there any defects in the transformation? This verification problem can be expressed in terms of the transformation model because, like any other model, it is expected to satisfy several reasonable assumptions. For instance, it should be possible to instantiate the model in some way that does not violate any integrity constraint, including the OCL invariants of the metamodels and the transformation rules. Failing to satisfy these criteria may be a symptom of an incomplete, over-constrained or incorrect model, reflecting potential defects in the original M2M transformation.

In this section we formalize some properties that can be used to study quality notions of M2M transformations. These quality notions capture static properties of the M2M transformation, that is, they consider the application of the transformation to specific source and target models rather than studying the evolution of the model (e.g. incremental transformation or model synchronization). We introduce a particular notation in order to keep the formalization independent of the language employed for the transformation specification and the approach used for analysis. However, the predicates that we will define have a direct correspondence with the invariants extracted for TGGs and QVT.

All these properties can be encoded as UML/OCL consistency problems on the transformation model and verified using our UML/OCL to CSP approach introduced before. For example, executability of the transformation is directly equivalent to the consistency problem, i.e. a transformation is executable iff its transformation model is satisfiable (i.e. if there is a pair of source and target models satisfying the transformation model). Other verification properties have

to be decomposed into two or more consistency problems affecting either only the source model, only the target model, or the entire transformation model. For example, we can prove that a transformation is not total if we find a counterexample, i.e. a legal instance of the source model with no corresponding instance in the target model. To find the counterexample, first we generate a legal instance x of the source model. Then, we check if the entire transformation model is consistent when an additional invariant is added: the source model must be instantiated to x. If it is inconsistent, we have found our counterexample, otherwise, we keep generating new instances for x until we find our counterexample or we conclude no counterexample exists. A similar procedure can be used to check the other properties.

The notation used in the formalization is the following:

- S and T denote a source and a target model respectively.
- $\langle S, T \rangle$ is used for a pair of related source and target models.
- r denotes a rule or relation[9], where we write $\mathsf{PRE}_{\mathsf{r}}^{\mathsf{Fwd}}$, $\mathsf{PRE}_{\mathsf{r}}^{\mathsf{Bwd}}$ to denote its forward and backward pre-conditions, and $\mathsf{POST}_{\mathsf{r}}$ its post-conditions. In our transformation model, $\mathsf{PRE}_{\mathsf{r}}^{\mathsf{Fwd}}$ and $\mathsf{PRE}_{\mathsf{r}}^{\mathsf{Bwd}}$ correspond to the generated OCL queries `p-enabled` presented in previous definitions for TGGs and QVT respectively, whereas $\mathsf{POST}_{\mathsf{r}}$ corresponds to the complete generated invariant.
- TS denotes a M2M specification made of a set of rules or relations.

We also use the auxiliary function $\mathsf{OCC}(\_, \_)$ that returns all occurrences of the first argument (a pair of related models with a set of constraints) into the second (a pair of related models). The following predicates will be used to define verification properties, where graphs G and H used as examples can be found in Fig. 3.21:

- $\mathsf{Inv}[S]$ holds if S is conformant to its meta-model. Similarly, $\mathsf{Inv}[T]$ holds if T is conformant to its meta-model.
- $\mathsf{Inv}[\langle S, T \rangle] \stackrel{def}{=} \mathsf{Inv}[S] \wedge \mathsf{Inv}[T]$.
- $\langle S, T \rangle \subseteq \langle S', T' \rangle$ holds if $\langle S, T \rangle$ is a submodel of $\langle S', T' \rangle$.
- $\langle S, T \rangle = \langle S', T' \rangle$ holds if $\langle S, T \rangle$ is isomorphic to $\langle S', T' \rangle$, i.e. both models are equal up to equality of object identifiers.
- $\mathsf{EN}_{\mathsf{r}}^{\mathsf{Fwd}}[\langle S, T \rangle] \stackrel{def}{=} \mathsf{OCC}(\mathsf{PRE}_{\mathsf{r}}^{\mathsf{Fwd}}, \langle S, T \rangle) \neq \emptyset$, i.e. r is source-enabled if there is some occurrence of its forward pre-condition. For TGG rules, this predicate corresponds to Definition 4. For example, $\mathsf{EN}_{\mathsf{Class-Table}}^{\mathsf{Fwd}}[G]$ holds because there is one occurrence of $\mathsf{PRE}_{\mathsf{Class-Table}}^{\mathsf{Fwd}}$ in G.

[9] In the following, we use rule and relation interchangeably.

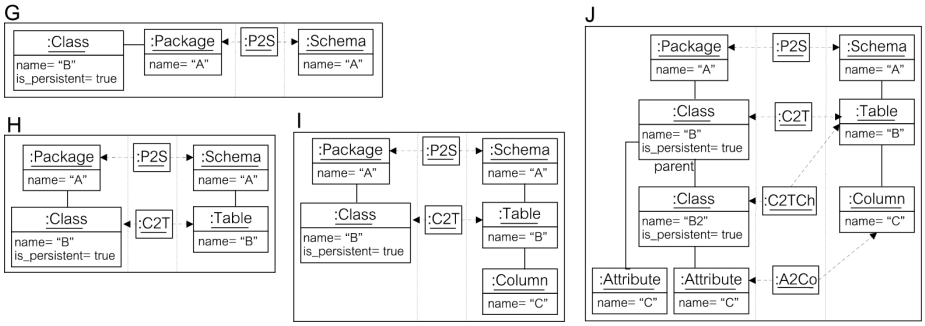


Figure 3.21: Example triple graphs for the verification of rule properties: (1) G is an example of forward applicability for rule `Class-Table`; (2) H is an example of forward weak executability for rule `Class-Table`; (3) I is an example of executability for rule `Class-Table`; (4) J is a counterexample of strong executability for rule `Attribute-Column`

- $\mathsf{EN}^{\mathsf{Bwd}}_{\mathsf{r}}[\langle S,T\rangle] \stackrel{def}{=} \mathsf{OCC}(\mathsf{PRE}^{\mathsf{Bwd}}_{\mathsf{r}}, \langle S,T\rangle) \neq \emptyset$. For example, $\mathsf{EN}^{\mathsf{Bwd}}_{\mathsf{Class-Table}}[G]$ does not hold because there is no occurrence of $\mathsf{PRE}^{\mathsf{Bwd}}_{\mathsf{Class-Table}}$ in G, but $\mathsf{EN}^{\mathsf{Bwd}}_{\mathsf{Class-Table}}[H]$ holds.

- $\mathsf{SAT}^{*}_{\mathsf{r}}[\langle S,T\rangle] \stackrel{def}{=} (\forall \langle S',T'\rangle \in \mathsf{OCC}(\mathsf{PRE}^{\mathsf{Fwd}}_{\mathsf{r}}, \langle S,T\rangle) \cup \mathsf{OCC}(\mathsf{PRE}^{\mathsf{Bwd}}_{\mathsf{r}}, \langle S,T\rangle) : \exists \langle S'',T''\rangle \in \mathsf{OCC}(\mathsf{POST}_{\mathsf{r}}, \langle S,T\rangle) : \langle S',T'\rangle \subseteq \langle S'',T''\rangle)$. This predicate holds when a pair of models satisfies the post-conditions of a rule in all occurrences of its pre-conditions, which may be zero (trivial satisfaction). In such a case we say that the models satisfy the rule, which corresponds to Definition 5 for TGG rules. For example, $\mathsf{SAT}^{*}_{\mathsf{Class-Table}}[H]$ holds because the only occurrences of $\mathsf{PRE}^{\mathsf{Fwd}}_{\mathsf{Class-Table}}$ and $\mathsf{PRE}^{\mathsf{Bwd}}_{\mathsf{Class-Table}}$ are satisfied (i.e. included in an occurrence of $\mathsf{POST}_{\mathsf{Class-Table}}$). For QVT it is similar, but in addition the *when* and *where* clauses may imply the satisfaction of other relations.

- $\mathsf{SAT}^{*}_{\mathsf{TS}}[\langle S,T\rangle] \stackrel{def}{=} (\forall r \in TS : \mathsf{SAT}^{*}_{\mathsf{r}}[\langle S,T\rangle])$. This predicate holds if $\langle S,T\rangle$ satisfies (even trivially) all rules in the specification TS.

- $\mathsf{SAT}_{\mathsf{r}}[\langle S,T\rangle] \stackrel{def}{=} \mathsf{SAT}^{*}_{\mathsf{r}}[\langle S,T\rangle] \wedge (\mathsf{EN}^{\mathsf{Fwd}}_{\mathsf{r}}[\langle S,T\rangle] \vee \mathsf{EN}^{\mathsf{Bwd}}_{\mathsf{r}}[\langle S,T\rangle])$. This predicate holds when a pair of models satisfies r's post-conditions, but not trivially (i.e. at least one occurrence exists). For example, $\mathsf{SAT}_{\mathsf{Class-Table}}[H]$ holds.

- $\mathsf{SAT}_{\mathsf{TS}}[\langle S,T\rangle] \stackrel{def}{=} (\forall r \in TS : \mathsf{SAT}_{\mathsf{r}}[\langle S,T\rangle])$.

Once established the notation and necessary predicates, we are ready to define the list of quality properties of M2M transformations at two levels: considering the role of individual rules within a transformation, or considering the

transformation model as a whole. In addition, some properties can be studied at both levels. We start with properties applicable to the level of rules. We assume the forward direction, but it should be clear that the same properties can be easily defined for the backward direction. Each property contains a description, its formula in terms of the previous notation, and an example. The graphs used as examples can be found in Fig. 3.21 (for QVT the examples would be similar but assuming an empty correspondence graph).

Applicable: r is *forward applicable* if there is a pair of models where r is source-enabled and the source model satisfies its meta-model constraints. We do not ask the target model to satisfy its meta-model constraints, as they may be violated during the transformation (e.g. lower cardinality constraints in associations).

Formula: $\exists\langle S,T\rangle : \mathsf{Inv}[S] \wedge \mathsf{EN}_{\mathsf{r}}^{\mathsf{Fwd}}[\langle S,T\rangle]$.

Example. Rule `Class-Table` is forward applicable in G because there is one occurrence of $\mathsf{PRE}^{\mathsf{Fwd}}_{\mathsf{Class-Table}}$ and the source graph is a valid model.

Weak Executable: r is *forward weak executable* if there exists a pair of models that satisfy r, and the source is a valid model.

Formula: $\exists\langle S,T\rangle : \mathsf{Inv}[S] \wedge \mathsf{SAT}_{\mathsf{r}}[\langle S,T\rangle]$.

Example. Rule `Class-Table` is forward weak executable because H contains one occurrence of $\mathsf{POST}_{\mathsf{Class-Table}}$. Please note that the target graph of H is not a valid model, as tables need at least one column. However this condition is not demanded by the property.

Executable: r is *executable* if there exists a valid pair of models that satisfy it. Note that this property is independent of the direction.

Formula: $\exists\langle S,T\rangle : \mathsf{Inv}[\langle S,T\rangle] \wedge \mathsf{SAT}_{\mathsf{r}}[\langle S,T\rangle]$.

Example. Rule `Class-Table` is executable because graph I contains one occurrence of $\mathsf{POST}_{\mathsf{Class-Table}}$ and its source and target graphs are valid models.

Strong Executable: r is *forward strong executable* if the target of every source model where r is source-enabled can be completed to satisfy r.

Formula: $\forall\langle S,T\rangle : \mathsf{Inv}[\langle S,T\rangle] \wedge \mathsf{EN}_{\mathsf{r}}^{\mathsf{Fwd}}[\langle S,T\rangle] \Rightarrow \exists T' : (\mathsf{SAT}_{\mathsf{r}}[\langle S,T\rangle] \vee (\mathsf{Inv}[\langle S,T'\rangle] \wedge \mathsf{SAT}_{\mathsf{r}}[\langle S,T'\rangle] \wedge \langle S,T\rangle \subseteq \langle S,T'\rangle))$.

Example. Rule `Attribute-Column` is not forward strong executable because, as the counterexample triple graph J shows, a class diagram where two classes related through inheritance define two attributes with same name cannot be translated into a valid target model. On the other hand `Package-Schema` is strong executable.

Remark. These three forms of executability demand increasing levels of satisfiability for a given rule. While for weak executability and executability r has to be existentially satisfied (in the latter by some valid target

model), in the strong version it must be universally satisfied in all cases where it can be source-enabled. Note that a rule is executable if and only if it is weak executable both forwards and backwards.

Total: r is *total* if it is not trivially satisfiable in every valid source model. This is equivalent to ask r to be forward weak executable in each valid source model.

Formula: $\forall S : \mathsf{Inv}[S] \Rightarrow \exists T : \mathsf{SAT}_{\mathsf{r}}[\langle S, T\rangle]$.

Example. Rule `Package-Schema` is not total, because the empty model satisfies the source meta-model constraints, but there is no target model that together with it satisfies the rule. The rule would be total should we add to the meta-model a constraint asking each model to have at least one package.

Deterministic: r is *deterministic* if each valid source model can be correctly transformed in a unique way using r.

Formula: $\forall S, T, T' : \mathsf{Inv}[S] \Rightarrow (\mathsf{EN}_{\mathsf{r}}^{\mathsf{Fwd}}[\langle S, T\rangle] \wedge \mathsf{EN}_{\mathsf{r}}^{\mathsf{Fwd}}[\langle S, T'\rangle] \wedge \mathsf{SAT^*}_{\mathsf{TS}}[\langle S, T\rangle] \wedge \mathsf{SAT^*}_{\mathsf{TS}}[\langle S, T'\rangle] \Rightarrow T = T')$.

Example. All rules in our example are deterministic. In general, this property can be used to detect under-constrained transformation models. For example, should we change the cardinality of the mapping between `C2T` and `Table` from 0..1 to $*$, then rule `Class-Table` would be non-deterministic because two classes with the same name could be mapped to the same table or to two tables with the same name. A rule could also fail to be deterministic due to attribute computation (e.g. if an attribute value is set to be the square root of another), or because the target model contains elements not mentioned in the transformation. For instance, if the relational schema meta-model had e.g. foreign key nodes, as these are not considered by any rule, there could be target models with and without foreign keys associated to a unique source model.

Finally note that the formula demands $\langle S, T\rangle$ and $\langle S, T'\rangle$ to satisfy (even trivially) the whole transformation specification TS. Otherwise no rule in our example would be deterministic. See for example H and I which satisfy `Package-Schema` and have the same source model.

Functional: r is *functional* if it is total and deterministic.

Formula: $Total(r) \wedge Deterministic(r)$.

Example. No rule in our example is functional. Rule `Package-Schema` would be functional if we demand at least one package in each UML model.

Exhaustive: r is *exhaustive* if it is satisfiable in each target model. This property is the dual of property *total*, and is equivalent to ask r to be weak executable in each valid target model.

Formula: $\forall T : \mathsf{Inv}[T] \Rightarrow \exists S : \mathsf{SAT}_{\mathsf{r}}[\langle S, T\rangle]$.

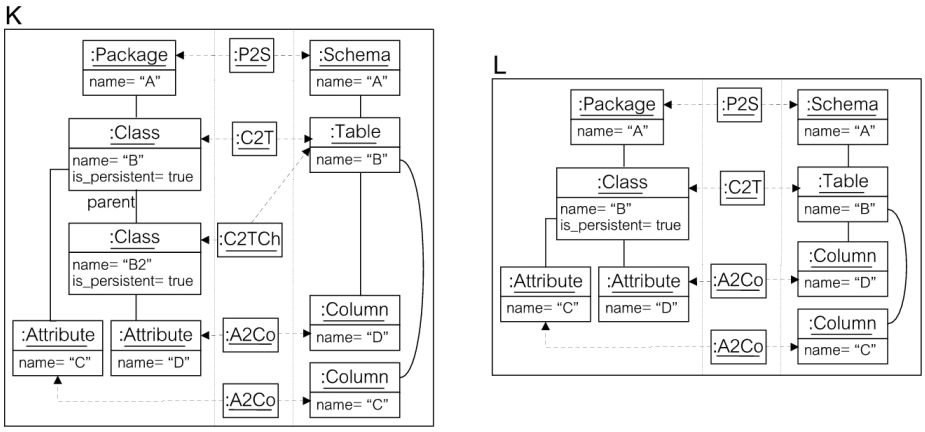


Figure 3.22: Example triple graphs for the verification of transformation properties: our TGG is executable but non-injective (we find two source models for the same target model)

Example: No rule in the example is exhaustive. Rule `Package-Schema` is not exhaustive because there is no source model that together with an empty target model satisfies the rule. The rule would be exhaustive should we add a constraint to the target meta-model asking for at least one schema in each RDBMS model.

Injective: r is *injective* if each valid target model is a correct transformation of a unique source model. This property is the dual of *deterministic* for the source model.

Formula: $\forall T, S, S' : \mathsf{Inv}[T] \Rightarrow (\mathsf{EN}_{\mathsf{r}}^{\mathsf{Fwd}}[\langle S, T\rangle] \wedge \mathsf{EN}_{\mathsf{r}}^{\mathsf{Fwd}}[\langle S', T\rangle] \wedge \mathsf{SAT}^{*}{}_{\mathsf{TS}}[\langle S, T\rangle] \wedge \mathsf{SAT}^{*}{}_{\mathsf{TS}}[\langle S', T\rangle] \Rightarrow S = S')$.

Example: Rule `Class-Table` is not injective. We can find the counterexample graphs K and L shown in Fig. 3.22 which have the same target, and one is produced from a class with two attributes, and the other from two classes related through inheritance with one attribute each. On the contrary, rule `Package-Schema` is injective.

Bijective: r is *bijective* if it is exhaustive and injective.

Formula: $Exhaustive(r) \wedge Injective(r)$.

Example. No rule in the example is bijective. Rule `Package-Schema` would be bijective should we forbid empty source and target models.

Redundant: r is *redundant* in a specification TS if the set of pairs of models satisfying TS is exactly the same as those satisfying $TS \setminus \{r\}$.

Formula: $\forall \langle S, T\rangle : \mathsf{Inv}[\langle S, T\rangle] \Rightarrow (\mathsf{SAT}^{*}{}_{\mathsf{TS}\setminus\{\mathsf{r}\}}[\langle S, T\rangle] \Leftrightarrow \mathsf{SAT}^{*}{}_{\mathsf{TS}}[\langle S, T\rangle])$.

Example. None of the rules in the example specification are redundant. An example of redundant rule would be one like `Attribute-Column`, but applicable only to persistent classes.

Enabledness Subsumption: Given two rules r_1 and r_2, r_1 *forward subsumes* r_2, written $r_1 \leq_F r_2$, if whenever r_2 is source-enabled so is r_1. That is, the forward pre-conditions of r_1 are weaker than those of r_2.

Formula: $\forall \langle S, T\rangle : \mathsf{Inv}[S] \wedge \mathsf{EN}^{\mathsf{Fwd}}_{\mathsf{r2}}[\langle S, T\rangle] \Rightarrow \mathsf{EN}^{\mathsf{Fwd}}_{\mathsf{r1}}[\langle S, T\rangle]$.

Example. We have that `Package-Schema` $\leq_F$ `Class-Table`, as whenever the latter is source-enabled, so is the former. Note that if a QVT relation r calls relations $r_1, ..., r_n$ in the where clause, we should have $r \leq_F r_1 \wedge ... \wedge r \leq_F r_n$ if the relation is to be enforced in the forward direction. Similarly, if relation r calls relations $r_1, ..., r_n$ in the when clause, we should have $r_1 \leq_F r \wedge ... \wedge r_n \leq_F r$ if the relation is to be enforced in the forward direction. Note that if the relation is meant to be bi-directional, we would have backward subsumption too.

Next, we generalize some of the presented properties to the level of transformation. In this case all properties are independent of the direction.

Executable: TS is *executable* if there is a valid pair of models satisfying it.

Formula: $\exists \langle S, T\rangle : \mathsf{Inv}[\langle S, T\rangle] \wedge \mathsf{SAT}^*{}_{\mathsf{TS}}[\langle S, T\rangle]$.

Example. Our example TGG transformation is executable because, e.g. graphs K and L satisfy (trivially or not) all rules. Note that we use the $\mathsf{SAT}^*{}_{\mathsf{TS}}[\ldots]$ predicate instead of $\mathsf{SAT}_{\mathsf{TS}}[\ldots]$ because otherwise we would be requiring at least one explicit occurrence of each rule.

Total: TS is *total* if for each valid source model there is a valid target model satisfying it.

Formula: $\forall S : \mathsf{Inv}[S] \Rightarrow \exists T : \mathsf{SAT}^*{}_{\mathsf{TS}}[\langle S, T\rangle] \wedge \mathsf{Inv}[T]$.

Example. Our TGG is not total as e.g. there is no valid target model such that together with the source model of graph G satisfies the specification (tables without columns are not allowed).

Deterministic: TS is *deterministic* if each valid source model can be correctly transformed in a unique way.

Formula: $\forall S, T, T' : \mathsf{Inv}[S] \Rightarrow (\mathsf{SAT}^*{}_{\mathsf{TS}}[\langle S, T\rangle] \wedge \mathsf{SAT}^*{}_{\mathsf{TS}}[\langle S, T'\rangle] \wedge \mathsf{Inv}[T] \wedge \mathsf{Inv}[T'] \Rightarrow T = T')$.

Example. Our TGG is deterministic because its rules are deterministic.

Functional: TS is *functional* if it is total and deterministic.

Formula: $Total(TS) \wedge Deterministic(TS)$.

Example. Our specification is not functional because it is not total.

Exhaustive: TS is *exhaustive* if each target model can be produced from some source model. This property is the reciprocal of property *total*.

Formula: $\forall T : \mathsf{Inv}[T] \Rightarrow \exists S : \mathsf{SAT}^*{}_{\mathsf{TS}}[\langle S, T\rangle] \wedge \mathsf{Inv}[S]$.

Example: Our TGG is exhaustive as each valid relational schema can be generated from some class diagram.

Injective: TS is *injective* if each target model satisfies TS together with just one single source model. This property is the reciprocal of *deterministic*.

Formula: $\forall T, S, S' : \mathsf{Inv}[T] \Rightarrow (\mathsf{SAT}^*{}_{\mathsf{TS}}[\langle S, T\rangle] \wedge \mathsf{SAT}^*{}_{\mathsf{TS}}[\langle S', T\rangle]$
$\mathsf{Inv}[S] \wedge \mathsf{Inv}[S'] \Rightarrow S = S')$.

Example: Our TGG is not injective as graphs K and L show.

Bijective: TS is *bijective* if it is exhaustive and injective.

Formula: $Exhaustive(TS) \wedge Injective(TS)$.

Example. Our specification is not bijective because it is not injective.

3.7 A UML/OCL Framework for the Analysis of Graph Transformation Rules

Graph Transformation [47, 95] is a rule-based technique for expressing model transformations. It has been used for specifying in-place transformations like animations, simulations, optimizations and redesigns. It is now gaining increasing popularity due to its visual form (making rules intuitive) and formal nature (making rules and grammars amenable to analysis). For example, it has been used to describe the operational semantics of Domain Specific Visual Languages (DSVLs) [43], taking the advantage that it is possible to use the concrete syntax of the DSVL in the rules, which then become more intuitive to the designer. As models and meta-models can be expressed as graphs (with typed, attributed nodes and edges), graph transformation can be used for model manipulation in the MDD approach.

Clearly, the important role of (graph) transformations in MDD needs to be supported by analysis techniques that help designers in determining the quality of such transformations. So far, the main formalization of graph transformation is the so called algebraic approach [47], which uses category theory in order to express the rewriting. Prominent examples of this approach are the double [47] and the single [95] pushout (DPO and SPO), which have developed interesting analysis techniques, e.g. to check independence between pairs of derivations [47, 95], or to calculate critical pairs (minimal context of pairs of conflicting rules) [57]. However, graph grammar analysis techniques work with simplified meta-models (so called type graphs), which lack OCL-like constraints for expressing the well-formedness rules of the meta-model. By not considering the meta-model constraints, one could design incorrect rules violating such well-formedness rules. We believe that integrating OCL constraints and graph

transformation is crucial for the applicability of the latter in real software MDD projects.

In this section, we show how to integrate OCL and graph transformation, and provide more advanced analysis techniques for graph transformations by using OCL as an intermediate representation to express the semantics of graph transformation rules. Then, this OCL representation is used to automatically verify the analysis properties of interest using our UMLtoCSP approach. Note that the nature of graph transformation rules is more operational than the declarative M2M languages seen in the previous section. That's why, while for the previous transformation languages we were using UML/OCL static models as intermediate representation for the verification process, in this case we will use dynamic UML/OCL models (i.e. models with operation contracts) for that.

Representing rules with OCL, concepts like attribute computation and attribute conditions in rules can be seamlessly integrated with the meta-model and its OCL constraints during the rule analysis. Moreover, it makes available a plethora of tools able to analyze this kind of specifications. A secondary benefit of our approach is that graph transformation is made available to the increasing number of MDA tools that the community is building and vice-versa. For example, by using such MDA tools, it could be possible to (partially) generate code for the transformations, or apply metrics and redesigns to the rules. In addition, the OCL specification derived from graph transformation rules could be used as a way to add behaviour to meta-models, and contracts for methods. Finally, an intermediate OCL representation serves as a neutral language for the integration of different transformation languages, approaches and tools.

More in detail, we use OCL to represent fully expressive DPO and SPO rules with negative application conditions and attribute conditions. In addition, we have represented a number of analysis properties with OCL, taking into account both the rule structure and the rule and meta-model constraints. These properties include rule applicability (whether there is a model satisfying the rule and the meta-model constraints), weak executability (whether the rule's post-condition and the meta-model constraints are satisfiable by some model) and strong executability (if a rule applied to a legal model – which conforms to the meta-model and its well-formedness constraints – always yields a legal model) among others.

3.7.1 Introduction to graph transformations

In this section we give an intuition on graph transformation by presenting some rules that belong to a simulator of a DSVL for production systems. Fig. 3.23 shows the DSVL meta-model.

The meta-model defines different kinds of machines (concrete subclasses of *Machine*), which can be connected through conveyors. These can be interconnected and contain pieces (the number of pieces they actually hold is stored in attribute *nelems*), up to its maximum capacity (attribute *capacity*). The OCL invariants on class *Conveyor* guarantee that the number of elements of a conveyor is equal to the number of pieces connected to it and never exceeds its

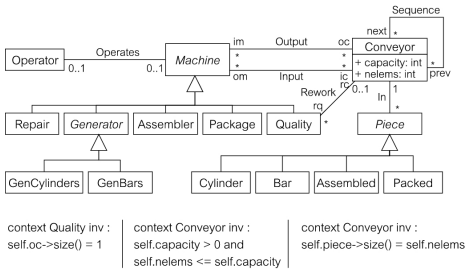


Figure 3.23: Meta-model of a DSVL for production systems

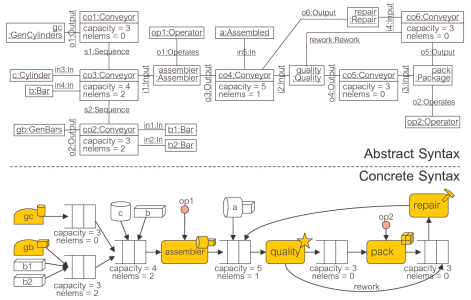


Figure 3.24: Example production system model

capacity. Human operators are needed to operate the machines, which consume and produce different types of pieces from/to conveyors.

Fig. 3.24 shows a production model example conformant to the previous meta-model, expressed using abstract syntax on top, and a visual concrete syntax at the bottom. It contains six machines (one of each type), two operators, six conveyors and five pieces. In concrete syntax, machines are represented as decorated boxes, except generators, which are depicted as semi-circles with an icon representing the kind of piece they generate. Operators are shown as circles, conveyors as lattice boxes, and each kind of piece has its own shape. In the model, the two operators are currently operating an assembler and a package machine respectively. Even though all associations in the meta-model are bidirectional, we have assigned arrows in the concrete syntax, but this does not affect navigability. For the case of the *Input* and *Output* associations, the arrow in the concrete syntax helps identifying the input and output machines.

We use graph transformation [47] for the specification of the DSVL operational semantics. A graph grammar is made of a set of rules and an initial graph (*host graph*) to which the rules are applied. Each rule is made of a left and a right hand side (LHS and RHS) graph. The LHS expresses pre-conditions for the rule to be applied, whereas the RHS contains the rule's post-conditions. In order to apply a rule to the *host graph*, a morphism (an occurrence or *match*) of the LHS has to be found in it. If several are found, one is selected randomly.

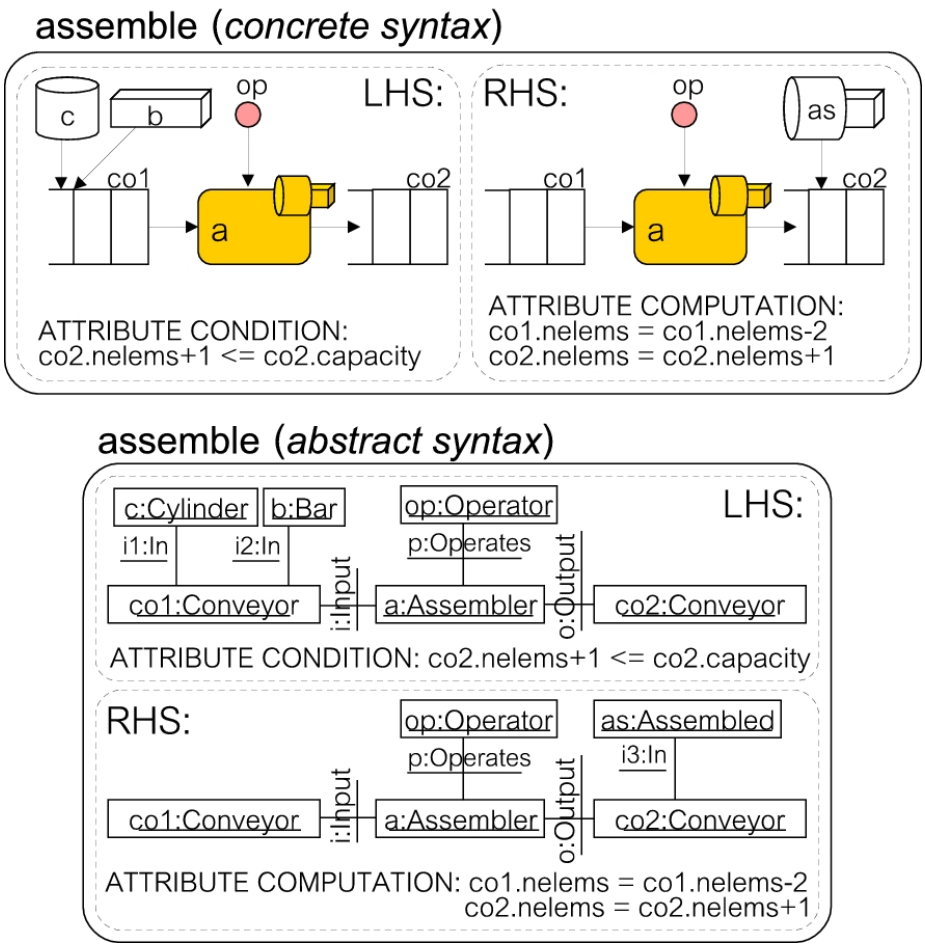


Figure 3.25: Rule in concrete (up) and abstract syntax (down)

Then, the rule is applied by substituting the match by the RHS. This process is called *direct derivation*. The grammar execution proceeds by applying the rules in non-deterministic order, until none is applicable.

Next, we show some of the rules describing the DSVL operational semantics. Rule "assemble" specifies the behaviour of an assembler machine, which converts one cylinder and a bar into an assembled piece. The rule is shown in concrete syntax in the upper part of Fig. 3.25, and in abstract syntax to the bottom. Fig. 3.26 shows its application to a model G (a sub-model of the one in Fig. 3.24) yielding a model H. First, an occurrence of the LHS is found in the model (dashed area). Then the elements in the LHS that do not appear in the RHS are deleted, whereas the elements in the RHS that do not appear in the LHS are created. Our rules may include attribute conditions, which must be satisfied by

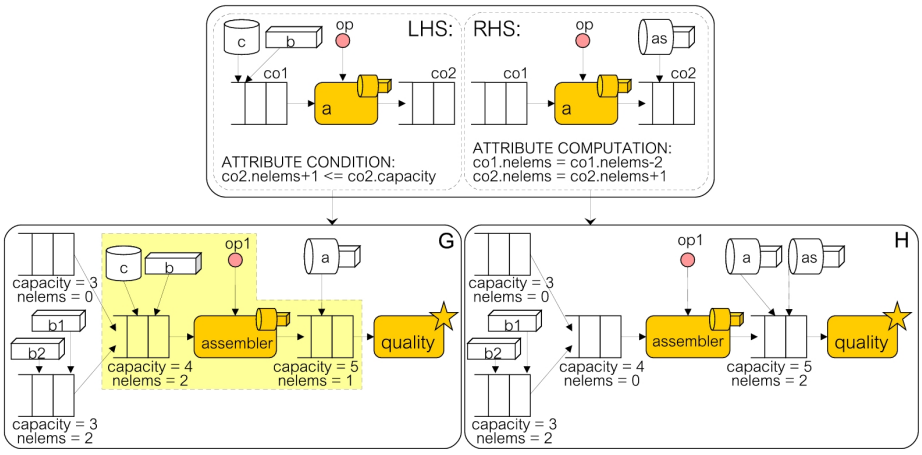


Figure 3.26: A direct derivation

the match, and attribute computations, both expressed in OCL. In Fig. 3.26, the match of conveyor *co*2 makes the rule's attribute condition $co2.nelems + 1 <= co2.capacity$ become $1 + 1 \leq 5$, and since it is true, the rule is applicable at that match. Attributes referenced to the right of an assignment in an attribute computation refer to the value of the attribute before the rule application. In the figure, the attribute *nelems* of the conveyors matched by the rule are updated.

There are two main formalizations of algebraic graph transformation [47]: DPO and SPO. From a practical point of view, their difference is that deletion has no side effects in DPO. That is, when a node in the host graph is deleted by a rule, the node can only be connected through those edges explicitly deleted by the rule. When applying the rule in Fig. 3.26, if piece "b" in model G would be connected to more than one conveyor (should that be allowed by the meta-model), then the rule could not be applied as those additional edges would become dangling in the host graph G after removing "b". This condition is called *dangling edge* condition. Instead, in SPO dangling edges are removed by the rewriting step.

A second difference is related to the injectivity of matches. A match can be non-injective, which means for example that two nodes with compatible type in the rule may be matched to a single node in the host graph. If the rule specifies that one of them should be deleted and the other one preserved, DPO forbids applying the rule at such a match, while SPO allows its application and deletes both nodes. In DPO, this is called the *identification condition*.

Fig. 3.27 shows further rules for the DSVL. Rule "move" describes the movement of pieces through conveyors. The rule has a Negative Application Condition (NAC) that forbids moving the piece if the source conveyor is the input to any kind of machine having an operator. Following [47], we take the match of the NAC as injective for both DPO and SPO. Rule "move" uses abstract nodes: piece "p" and machine "m" are abstract, and are visually represented with as-

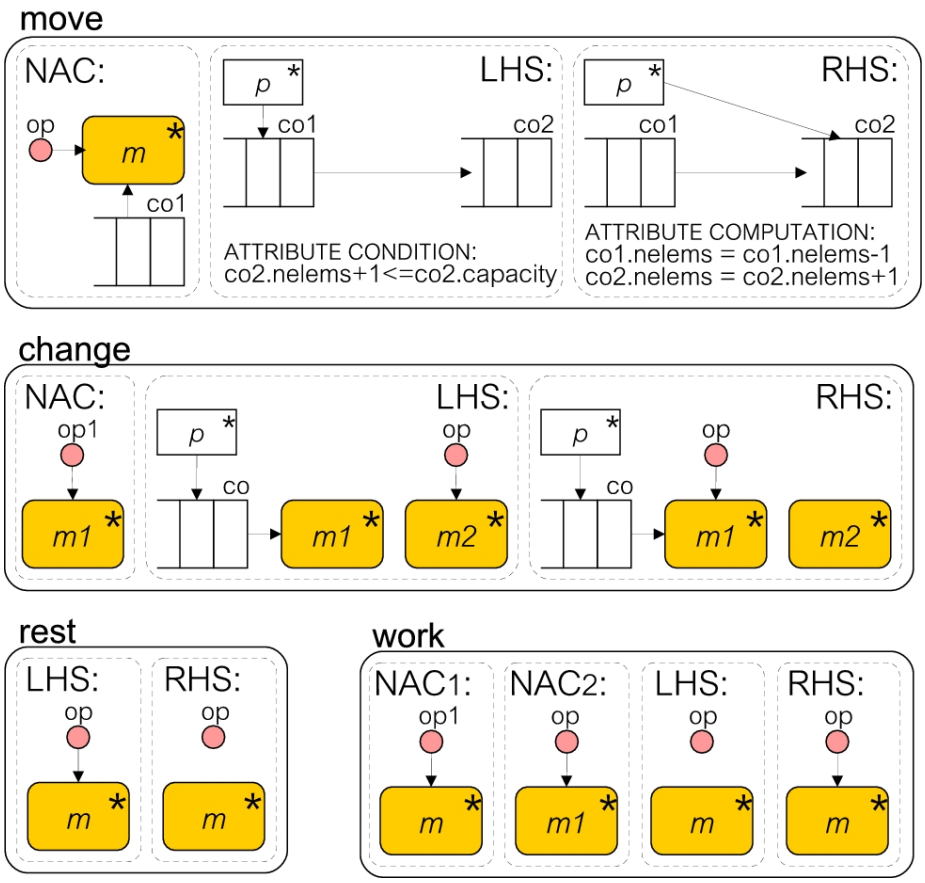


Figure 3.27: Additional rules for the DSVL simulator

terisks. Abstract nodes in a rule can get instantiated to nodes of any concrete subtype [41]. In this way, rules become much more compact. Rule "move" in the example is equivalent to 24 concrete rules, resulting from the substitution of piece and machine by their children concrete classes.

Rule "change" models an operator changing to a machine "m1" if the machine has some piece waiting to be processed and it is unattended. Rule "rest" models the break pause of an operator, by deleting its connection to a machine. Finally, rule "work" connects an idle operator (checked by NAC2) to an unattended machine (checked by NAC1).

3.7.2 From Graph Transformation to OCL

This section presents a procedure to translate graph transformation rules into an OCL-based representation. The procedure takes as input a graph trans-

formation system made of a set of rules, together with the MOF-compliant meta-model used as a context for the rules. As output, the method generates a set of semantically-equivalent declarative operations (one for each rule) specified in OCL. Declarative operations are specified by means of a contract consisting of a set of pre- and post-conditions. Roughly speaking, pre-conditions will define a set of conditions on the source model that will hold iff the rule can be applied, namely if the model has a match for the LHS pattern and no match for any NAC, while post-conditions will describe the new state of the model after executing the operation as stated by the difference between the rule's RHS and LHS.

More precisely, the input of the procedure is a tuple $(MM, ruleStyle, injMatch, GTS = \{r_j\}_{j \in J})$, where MM is a meta-model possibly restricted by OCL well-formedness rules, $ruleStyle$ and $injMatch$ are two flags indicating DPO or SPO semantics and injectivity of matches respectively, and GTS is a set of graph transformation rules. We represent DPO and SPO rules as $r = (LHS, RHS, ATT_{COND}, ATT_{COMP}, \{NAC^i, ATT^i_{COND}\}_{i \in I})$, where LHS, RHS and NAC^i are models that use the types in MM. Note that they do not necessarily have to satisfy all well-formedness rules in the meta-model (e.g. lower cardinality constraints), as these are patterns, not meant to be complete models. For rules expressing operational semantics, what is important is that the model to which the rules are applied remains consistent. Instances are identified across the models in the rule by their object identifiers, e.g. the elements preserved by the rule have the same object identifiers in LHS and RHS. ATT_{COND}, ATT^i_{COND} and ATT_{COMP} are sets of OCL expressions. The first two contain attribute conditions for the LHS and the i-th NAC, the latter contains attribute computations to state the new values for the attributes in the RHS.

The next subsections use this formalization to translate the GTS into a set of OCL operations. The name of the operations will be the name of the corresponding rule. All operations will be attached to an artificial class $System$, typically used in the analysis phase of a development process to contain the operations with the behaviour of the system [71]. Alternatively, each operation could be assigned to one of the existing classes. The GRASP patterns (General Responsibility Assignment Software Patterns [71]) can be used to choose the most appropriate class to hold each operation.

Translating the left-hand side

A rule r can be applied on a host graph (i.e. a model) if there is a match, that is, if it is possible to assign objects of the host graph to nodes in the LHS such that (a) the type in the host graph is compatible with the type in the LHS, (b) all edges in LHS can be mapped to links in the host graph and (c) the attribute conditions evaluate to *true* when symbols are replaced by the concrete attribute values in the model.

When defining the translation for condition (a) we must explicitly encode the set of quantifiers implicit in the semantics of graph transformation rules: when checking if the host graph contains a match for LHS we have to try assigning

each possible combination of objects from compatible types in the model to the set of nodes in the LHS pattern. Thus, we need one quantifier for each node in LHS. In terms of OCL, these quantifiers will be expressed as a sequence of embedded *exists* operators over the population of each node type (retrieved using the predefined *allInstances* operation).

Once we have a possible assignment of objects to the nodes in LHS we must check if the objects satisfy the (b) and (c) conditions. To do so, we define an auxiliary query operation *matchLHSr*, which returns true if a given set of objects complies with the pattern structure defined in LHS and satisfies its ATT_{COND} conditions. In particular, for each edge e linking two objects o_1 (of type t_1) and o_2 (of type t_2) in LHS, *matchLHSr* must define a $o_1.nav_{t2}$−> $includes(o_2)$ condition stating that o_2 must be included in the set of objects retrieved when navigating from o_1 to the related objects of type t_2; the right association end to use in the navigation nav_{t2} is extracted from MM according to the type of e and the type of the two participant objects. The ATT_{COND} conditions, already expressed using an OCL-like syntax in r, are directly mapped as a conjunction of conditions at the end of *matchLHSr*.

Let $L = \{L_1, \ldots, L_n\}$ denote the set of nodes in LHS and $E = \{(L_i, L_j) | L_i, L_j \in L$, and L_i connected to $L_j\}$ the set of edges. Then, according to the previous guidelines, the LHS pattern of r will be translated into the following equivalent pre-condition:

context System::r()
pre $L_1.type$::allInstances()−>exists(L_1 |
...
$L_n.type$::allInstances()−>exists(L_n |
matchLHSr($L_1, \ldots, L_n$))...)

---

context System::matchLHSr($L_1 : L_1.type$, ..., $L_n : L_n.type$) : Boolean
body $L_1.nav_{L2.type}$−>includes(L_2) and
... and
$L_i.nav_{Lj.type}$−>includes(L_j) and ATT_{COND}

where $L_i.type$ returns the *type* of the node L_i. Node identifiers are used to name the variable in the quantifier. Note that $L_i.type$::allInstances() returns all direct and indirect instances of $L_i.type$ (i.e. it returns also the instances of its subtypes) and thus abstract objects can be used in the definition of r. When E and ATT_{COND} are empty, the body of $matchLHSr$ just returns true.

As an example, the pre-condition for the "rest" rule is the following:

```
context System::rest()
 pre   Operator::allInstances()->exists(op |
       Machine::allInstances()->exists(m |
       matchLHSrest(op, m)))
-----------------------------------------------------
 context System::matchLHSrest( op: Operator,
                               m: Machine ) : Boolean
body  op.machine->includes(m)
```

where *matchLHSrest* is called for every possible combination of operators and machines in the given model (because of the two nested *exists* iterators). If one of such combinations satisfies *matchLHSrest* the pre-condition evaluates to true, meaning that the "rest" rule can be applied on the model.

Many graph transformation tools allow restricting the match to be injective, and this can be enforced in our translation procedure by setting the $injMatch$ flag to true. We can emulate injective matches by adding extra constraints in the pre-condition of the operation stating that every two objects with compatible type should be different. That is, the condition $L_i <> L_j$ is added for $L_i, L_j \in L$, if $L_i.type = L_j.type$ or if one is a subclass of the other. This technique is also used to ensure injectivity of NACs and to handle the identification condition.

Translating the negative application conditions

In presence of NACs, the pre-condition of r must also check that the set of objects of the host graph satisfying the LHS do not match any of the NACs.

The translation of a NAC pattern is similar to the translation of the LHS: an existential quantifier must be introduced for each new node in the NAC (i.e. each node not appearing also in the LHS pattern) and an auxiliary query operation $matchNACr$ will be created to determine if a given set of objects satisfy the NAC. Such $matchNACr$ operation is specified following the same procedure used to define *matchLHSr*. In addition, matches in the NAC must always be injective. Therefore, as part of the translation we must explicitly add conditions ensuring that two nodes of compatible types are not mapped to the same object in the host graph. These have the form $N_i <> L_j$ (or $N_i <> N_j$) for each node N_i in the NAC that is type-compatible with a node L_j in LHS (or another node N_j in the same NAC).

Within the pre-condition, the translation of the NACs is added as a negated condition immediately after the translation of the LHS pattern.

Let $N = \{N_1, \ldots, N_m\}$ denote the set of nodes in a NAC that do not appear in LHS. The extended pre-condition for r (LHS + NAC) is defined as:

context System::r()
pre $L_1.type$::allInstances()−>exists(L_1 |
...
$L_n.type$::allInstances()−>exists(L_n |
matchLHSr($L_1, \ldots, L_n$)
and not ($N_1.type$::allInstances()−>exists(N_1 |
...
$N_m.type$::allInstances()−>exists(N_m |
matchNACr($L_1, \ldots, L_n, N_1, \ldots, N_m$)
and $N_i <> L_j \ldots$ and $N_i <> N_l) \ldots)) \ldots)$

If r contains several NACs we just need to repeat the process for each NAC, creating the corresponding $matchNAC^i r$ operation each time.

As an example, the translation for the LHS and NAC patterns of the "work" rule is:

context System::work()
pre Machine::allInstances()−>exists(m|
Operator::allInstances()−>exists(op|
matchLHSwork(m,op)
and not Operator::allInstances()−>exists(op1|
matchNAC1work(m,op,op1) and op1 <> op)
and not Machine::allInstances()−>exists(m1|
matchNAC2work(m,op,m1) and m1 <> m))

---

context System::matchLHSwork(m:Machine,
op:Operator): Boolean
body true

---

context System::matchNAC1work(m:Machine,
op: Operator, op1: Operator) : Boolean
body m.operator−>includes(op1)

---

context System::matchNAC2work(m:Machine,
op: Operator, m1:Machine) : Boolean
body m1.operator−>includes(op)

For this rule *matchLHSwork* simply returns true since as long as a machine object and an operator exist in the host graph (ensured by the existential quantifiers in the pre-condition), the LHS is satisfied. The additional conditions imposed by the NACs state that no other operator ($op1$ in the NAC1) can be working on that machine, and that the operator in the LHS cannot be working on a different machine ($m1$ in the NAC2).

Translating the right-hand side

The effect of rule r on the host graph is the following: (1) the deletion of the objects and links appearing in LHS and not in RHS, (2) the creation of the objects and links appearing in RHS but not in LHS, and (3) the update of attribute values of objects in the match according to the ATT_{COMP} computations.

Clearly, when defining the OCL post-condition for r we need to consider not only the RHS pattern (the *new* state) but also the LHS and NAC patterns (the *old* state) in order to compute the differences between them and determine how the objects evolve from the old to the new state. In OCL, references to the old state must include the @*pre* keyword. For instance, a post-condition like $o.atr_1 = o.atr_1@pre + 1$ states that the value of atr_1 for object o is increased upon completion of the operation.

Therefore, the translation of the RHS pattern requires, as a first step, to select a set of objects of the host graph that are a match for the rule. Then, this initial set of objects will be updated according to the rule definition. Unsurprisingly, this initial condition is expressed with exactly the same OCL expression used to define the pre-condition[10] (where the goal was the same: to determine a match for r). The only difference is that in the post-condition, all references to attributes, navigations and predefined properties will include the @*pre* keyword. Note that when executing the rule it may happen that the set of objects used to satisfy the pre-condition differs from the one used in the match for the post-condition, when there are several possible matches for the rule in the graph. The goal of the pre-condition is just to check that at least one match exists. The post-condition selects one of these matches and changes it. This does not affect the correctness of our approach since graph transformation semantics are non-deterministic. If there are several possible matches, the rule will be applied again on these other matches afterwards. When evaluating the rules (see next section), our checking procedure will try all possible matches to determine their correctness.

Table 3.1: OCL expressions for changeRHSr

Element	**$\exists$ in LHS?**	**$\exists$ in RHS?**	**Update**	**OCL Expression**
Object o of type t	No	Yes	Insert o	o.oclIsNew() and o.oclIsTypeOf(t)
Object o of type t	Yes	No	Delete o	t::allInstances()−>excludes(o)
Link l between (o_1, o_2)	No	Yes	Insert l	$o_1.nav_{t2}$−>includes(o_2) and not $o_1.nav_{t2}@pre$−>includes(o_2)[11]
Link l between (o_1, o_2)	Yes	No	Delete l	$o_1.nav_{t2}$−>excludes(o_2)

Once a set of objects has been selected, it is passed to an auxiliary operation *changeRHSr* in charge of performing the changes defined by the rule. This operation is defined as a conjunction of conditions, one for each difference between the RHS and LHS patterns. Table 3.1 shows the OCL expressions that must be

[10] Though looking for a match twice is inefficient, OCL does not offer any mechanism to pass information about variable values from the pre-condition to the post-condition.

[11] This second part of the condition is only required when neither o_1 nor o_2 are new objects.

added to *changeRHSr* depending on the actions specified by r. Moreover, the ATT_{COMP} expression is added at the end of the procedure, with all references to previous attribute values extended with the `@pre` keyword. As usual, we assume in the definition of the post-condition for r that all elements not explicitly modified in the post-condition remain unchanged (*frame problem*).

Let $L = \{L_1, \ldots, L_n\}$ and $E = \{(L_i, L_j) | L_i, L_j \in L$, and L_i connected to $L_j\}$ be the set of nodes and edges in LHS, $DN = \{DN_1, \ldots, DN_q\} \subseteq L$ and $DE = \{(L_k, L_l)\} \subseteq E$ the nodes and edges in LHS but not in RHS, and $NN = \{NN_1, \ldots, NN_p\}$ and $NE = \{(NE_x, NE_y) | NE_x, NE_y \in NN \cup (L - DN)$, and NE_x connected to $NE_y\}$ the set of nodes and edges in RHS but not in LHS. Then, according to the previous guidelines, the RHS of r is translated into the following post-condition:

context System::r()
post $L_1.type$::allInstances@pre()−>exists(L_1 |
...
$L_n.type$::allInstances@pre()−>exists(L_n |
matchLHSr'($L_1, \ldots, L_n$) and
changeRHSr($L_1, \ldots, L_n$))...)

---

context System::matchLHSr'($L_1 : L_1.type$, ..., $L_n : L_n.type$) : Boolean
body $L_1.nav_{L2.type}$@pre−>includes(L_2) ... and
$L_i.nav_{Lj.type}$@pre−>includes(L_j)
and ATT_{COND}@pre

context System::changeRHSr($L_1 : L_1.type$, ..., $L_n : L_n.type$) : Boolean
body − − creation of NN nodes
$NN_1.oclIsNew()$ and
$NN_1.oclIsTypeOf(NN_1.type)$ ... and
$NN_p.oclIsNew()$ and
$NN_p.oclIsTypeOf(NN_p.type)$ and
− − removal of DN nodes
$DN_1.type$::allInstances()−>excludes(DN_1) ... and
$DN_q.type$::allInstances()−>excludes(DN_q) and
− − creation of NE links
$NE_1.nav_{NE_2.type}$−>includes(NE_2) and not
$NE_1.nav_{NE_2.type}$@pre−>includes(NE_2) ... and
$NE_x.nav_{NE_y.type}$−>includes(NE_y) and not
$NE_x.nav_{NE_y.type}$@pre−>$includes$(NE_y) and
− − removal of DE links
$L_1.nav_{L_2.type}$−>excludes(L_2) ... and
$L_k.nav_{L_l.type}$−>excludes(L_l)
− − attribute computation
and ATT_{COMP}

The previous translation pattern assumes a rule without NACs. If it has NACs, we should add OCL code for testing their satisfaction, as described in Section 3.7.2. Next we show the generated operations for "rest", the translation of the other rules is in the appendix.

```
context System::rest()
 pre   Operator::allInstances()->exists(op|
       Machine::allInstances()->exists(m|
       matchLHSrest(op,m)  ))
 post  Operator::allInstances@pre()->exists(op|
       Machine::allInstances@pre()->exists(m|
       matchLHSrest'(op,m) and changeRHSrest(op,m) ))
```

```
context System::matchLHSrest( op: Operator,
                              m: Machine ): Boolean
 body  op.machine->includes(m)
```

```
context System::matchLHSrest'( op: Operator,
                               m: Machine ): Boolean
 body  op.machine@pre->includes(m)
```

```
context System::changeRHSrest( op: Operator,
                               m: Machine ): Boolean
 body  op.machine->excludes(m)
```

Taking into account DPO and SPO semantics

The behaviour of the rules is slightly different depending on whether DPO or SPO semantics are assumed. The two differences we must consider in our translation are the *dangling edge* and the *identification* conditions. See [27] for details.

Optimizing the resulting constraints

These general translation patterns can be slightly simplified, yielding optimized OCL constraints, depending on the specific structure of each rule. In what follows we comment some possible simplifications.

- LHS nodes with no edges and no attribute conditions do not need to be passed as parameters for the *matchLHSr* operation. For those objects it is just enough to check their existence in the main pre-condition expression. E.g. in rule "work" we do not need to pass *op* and *m* as parameters for *matchLHSwork*.
- Similarly, LHS nodes not referenced in a NAC pattern do not need to be passed as parameters for the *matchNACr* operation.

- *matchLHSr* and *matchNACr* operations with an empty body (i.e. a body with just the *true* literal expression) can be skipped.
- For a rule r not including any *matchLHSr* or *matchNACr* operations, we do not need to nest the *exists* iterators in its pre-condition but just use a conjunction of separated quantifiers, improving the efficiency of its match finding process. For instance, assuming a hypothetical "assign" rule that given a piece and a conveyor puts the piece in the conveyor, the generated pre-condition for "assign" is the following:

```
context System::assign()
 pre  Piece::allInstances()->exists(p| true)
      and Conveyor::allInstances()->exists(c| true)
```

- The auxiliary *matchLHSr*, *matchNACr* and *changeRHSr* operations can be reused across different (or the same) rules sharing common patterns. As an example, the NAC1 and NAC2 patterns for the rule "work" and the NAC for the "change" rule (once their irrelevant parameters have been removed using the previous optimizations) can be merged into a single match operation that will be invoked (with different arguments) in the pre-conditions.
- Some conditions in the patterns may be subsumed by the meta-model constraints and thus can be removed from the operations. As an example, consider the condition *c.conveyor-> excluding(co1) -> isEmpty()* (in *matchLHS* for "assemble" under DPO semantics) saying that the piece c cannot be related to other conveyors except for $co1$. This condition is already implied by the maximum multiplicity constraint between *Piece* and *Conveyor* in the meta-model, which forces pieces to be related to at most one conveyor.

As an example, once we apply these optimizations to the "work" rule, (part of) its simplified translation is shown in the next table. First, *matchLHSwork* can be eliminated, then the parameters of *matchNAC1work* and *matchNAC2work* can be reduced. Finally, both operations for the NACs can be merged as they share the same pattern and parameters.

```
context System::work()
 pre  Machine::allInstances()->exists(m|
      Operator::allInstances()->exists(op|
      not Operator::allInstances()->exists(op1|
      matchNAC12work(m,op1) and  op1 <> op )
      and not Machine::allInstances()->exists(m1|
      matchNAC12work(m1,op) and m1 <> m ))
---------------------------------------------------
 context System::matchNAC12work(m:Machine,
                                op: Operator ) : Boolean
 body     m.operator->includes(op)
```

3.7.3 Verification of Rule Properties with OCL

Translating a graph grammar into a set of operations with OCL pre/post-conditions enables the analysis of relevant correctness properties of the rules using our verification method for dynamic UML/OCL methods described in the previous sections [12]

In what follows we describe the correctness properties we propose. The properties take into account the meta-model invariants that restrict the possible set of legal instantiations of the meta-model, as well as the pre- and post-conditions derived from the rules.

The following notation will be used to express these concepts: I denotes an instantiation of the meta-model, while I' represents the modified instantiation after invoking an operation. An instantiation I is called *legal*, noted as $\mathsf{Inv}[I]$, if it satisfies all the invariants of the meta-model, i.e. both the graphical restrictions such as multiplicity of roles in associations and the explicit OCL well-formedness rules. By $\mathsf{PRE}_\mathsf{r}[I]$ we denote that an instantiation I satisfies the pre-condition of an operation r. Regarding post-conditions, we write $\mathsf{POST}_\mathsf{r}II'$ to express that an instantiation I' satisfies the post-condition of an operation r given that I was the instantiation before executing the operation. As usual, to avoid the *frame problem* when interpreting $\mathsf{POST}_\mathsf{r}II'$, we assume that only the objects referenced in the post-condition can change their state during the operation execution.

Two families of properties will be studied. First, it is desirable to verify that for each rule there exists at least one valid model where it can be applied, as otherwise the rule is useless. Second, it is interesting to check whether different rules may interfere among them, making the order of application matter. Within each family of properties, several notions will be presented, each with a trade-off between the precision and the complexity of its analysis. The list is the

[12] The transformation of graph grammars into OCL can also be used for *validation*. While verification attempts to check that the graph transformation rules satisfy some required correctness properties ("is the transformation right?"), validation tries to ensure that the graph transformation rules match the intentions of the designer ("is this the right transformation?"). See [27] for details

following:

- **Applicability (AP):** Rule r is *applicable* if there is at least one legal instantiation of the meta-model where it can be applied.

$$\exists I : \mathsf{Inv}[I] \wedge \mathsf{PRE}_{\mathsf{r}}[I]$$

- **Weak executability (WE):** r is *weakly executable* if the post-condition is satisfiable in some legal instantiation.

$$\exists I, I' : \mathsf{Inv}[I] \wedge \mathsf{PRE}_{\mathsf{r}}[I] \wedge \mathsf{Inv}[I'] \wedge \mathsf{POST}_{\mathsf{r}}[I, I']$$

- **Strong executability (SE):** r is *strongly executable* if, for any legal instantiation that satisfies the pre-condition, there is another legal instantiation that satisfies the post-condition.

$$\forall I : (\mathsf{Inv}[I] \wedge \mathsf{PRE}_{\mathsf{r}}[I]) \rightarrow \exists I' : (\mathsf{Inv}[I'] \wedge \mathsf{POST}_{\mathsf{r}}[I, I'])$$

- **Overlapping rules (OR):** Two rules r and s overlap if there is at least one legal instantiation where both rules are applicable.

$$\exists I : \mathsf{Inv}[I] \wedge \mathsf{PRE}_{\mathsf{r}}[I] \wedge \mathsf{PRE}_{\mathsf{s}}[I]$$

- **Conflict (CN):** Two rules r and s are in conflict if firing one rule can disable the other, i.e. iff there is one legal instantiation where both rules are enabled, and after applying one of the rules, the other becomes disabled.

$$\begin{array}{ll} \exists I, I' : & \mathsf{Inv}[I] \wedge \mathsf{Inv}[I'] \wedge \mathsf{PRE}_{\mathsf{r}}[I] \wedge \mathsf{PRE}_{\mathsf{s}}[I] \wedge \\ & \mathsf{POST}_{\mathsf{r}} II' \wedge \neg \mathsf{PRE}_{\mathsf{s}}[I'] \end{array}$$

- **Independence (IN):** Two rules r and s are *independent* iff in any legal instantiation where both can be applied, any application order produces the same result. Four instantiations of the model will be considered to characterize this property: before applying the rules (I), after applying both rules (I''), after applying only rule r (I'_r) and after applying only rule s (I'_s).

$$\begin{array}{ccc} I & \xrightarrow{r} & I'_r \\ \downarrow s & & \downarrow s \\ I'_s & \xrightarrow{r} & I'' \end{array} \qquad \begin{array}{l} \forall I : (\mathsf{Inv}[I] \wedge \mathsf{PRE}_{\mathsf{r}}[I] \wedge \mathsf{PRE}_{\mathsf{s}}[I]) \rightarrow \\ \exists I'_r, I'_s, I'' : \\ (\mathsf{Inv}[I'_r] \wedge \mathsf{POST}_{\mathsf{r}}[I, I'_r] \wedge \mathsf{PRE}_{\mathsf{s}}[I'_r] \wedge \\ \mathsf{Inv}[I'_s] \wedge \mathsf{POST}_{\mathsf{s}}[I, I'_s] \wedge \mathsf{PRE}_{\mathsf{r}}[I'_s] \wedge \\ \mathsf{Inv}[I''] \wedge \mathsf{POST}_{\mathsf{r}}[I'_s, I''] \wedge \mathsf{POST}_{\mathsf{s}}[I'_r, I'']) \end{array}$$

- **Causal Dependence (CD):** Two rules r and s are *causally dependent* iff there is some legal instantiation where one is applicable and the other not, but applying the former makes the latter applicable.

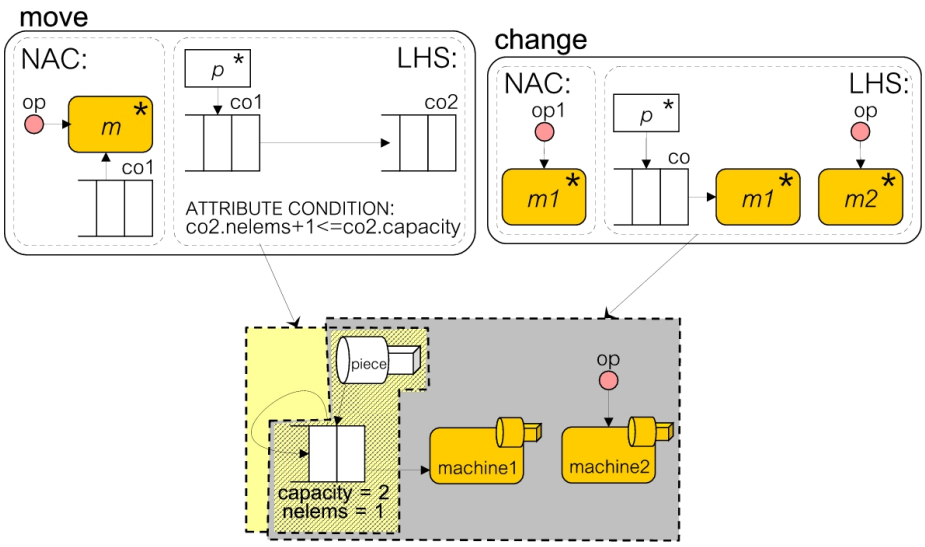


Figure 3.28: Overlapping of rules "move" and "change"

$$\exists I, I' : \quad \mathsf{Inv}[I] \wedge \mathsf{PRE_r}[I] \wedge \neg \mathsf{PRE_s}[I] \wedge \mathsf{POST_r} II' \wedge \mathsf{PRE_s}[I']$$

As an example of applicability, Fig. 3.28 shows a model in which rules "move" and "change" are applicable, since the model (I in the property definition) satisfies the invariants and both rules' pre-conditions. The matches of both rules are enclosed in different dotted polygons. The rules are applicable because the model contains occurrences of both LHSs, but not of the NACs. In fact, the model is also an example of overlapping between the rules. They overlap because they are applicable on the same model. Notice that in this match of the "move" rule, the source and destination conveyors are mapped to the same conveyor object, as there is no constraint forbidding this choice. It is worth noting that this instantiation helps to detect a problem in the system definition: non-injective matches are inadequate for rule "move", which in this case may be solved by adding an additional invariant to the meta-model stating that a conveyor cannot be next to itself, or by restricting the match of the LHS to be injective. This situation was detected when validating the system.

The difference between weak and strong executability is that the former requires the existence of just one legal model over which the rule can be successfully executed, while the strong version of the property asks for the executability of the rule in any situation in which its pre-condition is satisfied. If a rule does not satisfy strong executability, it may mean that it is underspecified regarding the OCL meta-model invariants. The needed extra constraints in rules can be either NACs or attribute conditions.

For instance, consider the situation in Fig. 3.29. The picture shows a rule

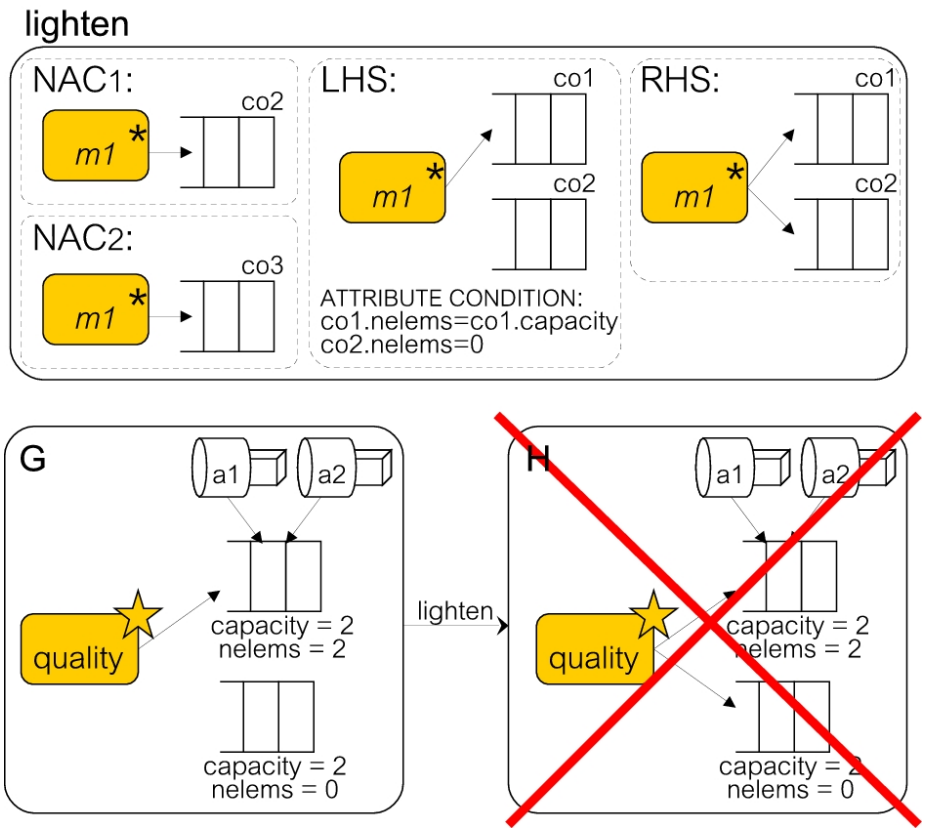


Figure 3.29: Non strongly executable rule

"lighten" that dynamically reconfigures the production plant by adding a new connection from a machine of any type to an empty conveyor, if the machine is already connected to a full conveyor and does not have further outputs. This rule is not strongly executable since there are models satisfying the LHS but where the rule cannot be applied. As an example, below the rule there is a model that satisfies the LHS by mapping the machine in the rule to a quality machine. The rule cannot be executed on this model, as an OCL constraint in the meta-model restricts quality machines to have at most one output. To make the rule strongly executable, one can add an additional NAC with just one machine of type quality mapped to $m1$. This NAC would ensure that the rule is not applied to quality machines. On the contrary, the rule is already weak executable because it can be applied as it is to any machine of type different from quality. For the same reason, the NACs of rules "change" and "work" in Fig. 3.27 are needed to ensure strong executability.

Some other times we need extra attribute conditions in the rules to ensure strong executability. For example, the attribute conditions in rules "assemble"

and "move" in Figs. 3.25 and 3.27 are necessary to ensure that both rules satisfy strong executability.

The conflict and independence properties are related to the concept of critical pairs. The term *critical pair* is used in graph transformation to denote two direct derivations in conflict (i.e. applying one disables the other), where the starting model is minimal [47, 57]. The set of critical pairs gives all potential conflicts, and if empty, it means that the transformation is confluent (i.e. a unique result is obtained from its application). For technical reasons, the algebraic approach to graph transformation usually models any attribute computation as a rewriting of edges [47]. This means that any two rules changing the same attribute of a node will be reported as conflicting. In general, this does not imply that one rule disables the other, but however ensures confluence. For example, two rules, one multiplying an attribute `x` by 2, and the other adding 1 to the same attribute, would be reported as a conflict; but no rule disables the other. On the contrary, our conflict condition is more precise about attribute computations and considers the OCL invariants, but by itself does not ensure confluence. In the previous example, if the rule multiplying by 2 is applied first, the attribute `x` becomes $2 \times x + 1$. However, if the rule adding 1 is applied first, we obtain $(x + 1) \times 2$. Hence the transformation would not be confluent. The advantage of our approach is that fewer conflicts will be reported by the conflict property, but confluence has to be checked with the independence property.

The independence property allows applying two rules in any order, obtaining the same result. This is a strong version of the local Church-Rosser theorem in DPO [47], where we require rule independence for every valid model I, and ensures confluence (i.e. same result). In the same way as the technique of critical pairs in graph transformation, we do not have to check each possible model in which rules overlap, but only the minimal ones.

The causal dependence property detects whether two rules have some dependency, in such a way that executing one enables the other. In this way, one rule may add one element that the other needs (produce-use dependency), or delete some element that is part of the NAC of the other rule (delete-forbid dependency). For example, consider rules "work" and "rest". The former associates an operator to a machine, while the latter deletes such connection. There is a produce-use dependency between the rules because "rest" needs an edge which "work" produces. These rules have also a produce-use dependency, because "rest" deletes a connection which is in the NAC of "work". Fig. 3.30 shows two host graphs, together with two derivations that illustrate these two dependencies.

Finally, note that studying whether all these properties hold in some specific host graph is possible, by replacing the $\exists I$ with some specific model I_1 and then checking if the formula holds for that I_1.

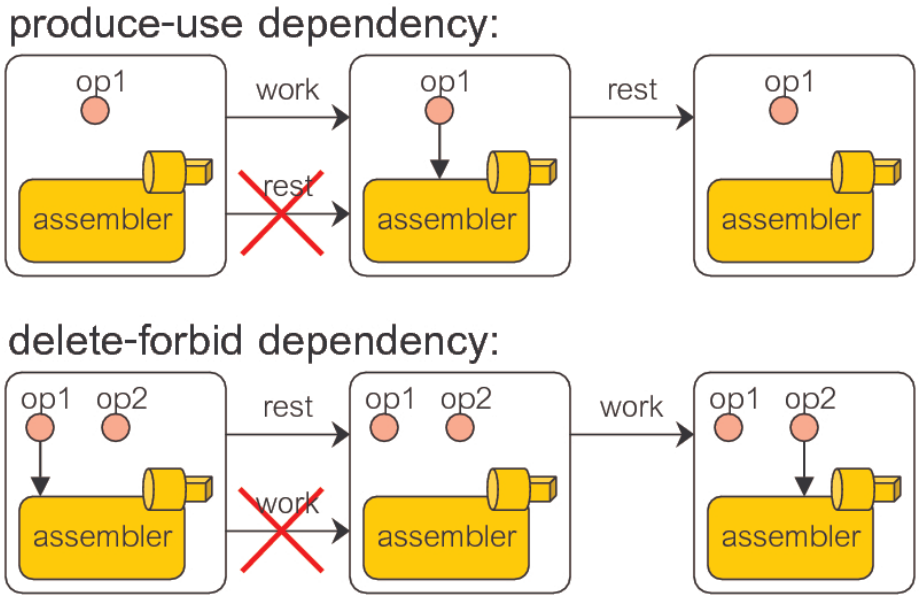


Figure 3.30: Causal dependencies

3.8 Tool Implementation

The core of our method has been implemented. Our prototype tool UML-toCSP [106] to translate UML models into a cSP is implemented as a set of ECL^{i}PS^e constraint libraries (2000 LoC) and Java classes (11500 LoC) implementing the GUI, the UML/OCL to CSP translation and glue code. The tool also uses several external libraries and tools: the OCL parser from the Dresden OCL toolkit [44], the MDR library for importing XMI files, the EMF (Eclipse Modeling Framework) libraries for importing ECore files, the ECL^{i}PS^e [100] constraint programming system for solving the CSPs and the GraphViz [55] graph visualization package for presenting the results graphically. Figure 3.31 presents the architecture of the tool.

As shown in the Figure 3.31, as a first step, the tool permits to import both the UML model and the OCL constraints from an XMI/ECore file and a text file, respectively. More specifically, the UML class diagram can be imported from an XMI (XML Metadata Interchange) file, such as those generated by CASE tools like ArgoUML, or in the ECore format from the Eclipse Modeling Framework[13]. The text file containing the OCL expressions is parsed using the Dresden OCL toolkit. Next, users can choose to generate the CSP from the UML/OCL model. As part of the translation process, users must indicate the correctness property they want to check on the model and (optionally) the

[13]EMF refers to the Eclipse IDE (`http://www.eclipse.org`.) not the ECL^{i}PS^e constraint solver.

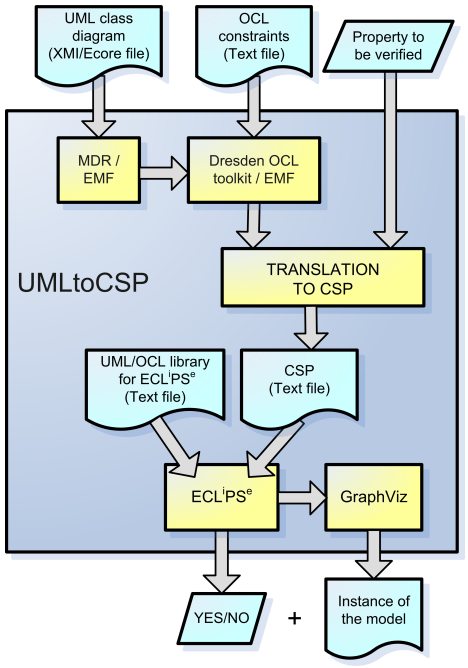


Figure 3.31: Architecture of UMLtoCSP.

ranges for the domains to be used for the variables in the CSP (otherwise default values are used). Finally, the tool generates the CSP with the support of our UML/OCL CSP library (included with the tool). The resulting CSP is executed using the ECL^{i}PS^e constraint solver API to try to find one solution for it, and thus, to determine the correctness of the original UML/OCL model. When such a solution exists, it is shown to the user graphically as an object diagram. If no solution exists, the tool prompts a message explaining this fact and suggesting a revision or the model or a change in the size of the search state space.

Figure 3.32 illustrates the graphical user interface used in this verification flow. Figure 3.32(a) shows the initial screen once the input models have been parsed. The classes and associations in the diagram are displayed in a tree view, and the textual OCL constraints appear in a separate frame. Also in this window, designers may optionally parametrize the search space by defining the domain of each attribute, the number of objects of a class or the number of links in an association. If the user does not feel the need to customize the search space, the tool automatically uses the suggested default domains based on the type of the attribute, e.g. 0 or 1 for boolean attributes and a finite range for integer attributes. For example, Figure 3.32 (a) shows how the user assigns the domain of the boolean attribute *isStudent*, which is 0..1 by default.

Then, using the menu from Fig. 3.32 (b), designers can select the properties of interest for the analysis. For properties like liveliness or constraint redun-

dancy which affect a specific class or constraint, a drop-down list provides the list of candidates from the model. After this step, verification is fully automated: the tool translates the UML class diagram, OCL constraints and the correctness properties into a CSP, which is passed to the ECL^{i}PS^e constraint solver API to find a solution to the CSP, and thus, to determine the correctness of the original UML/OCL model. When such a solution exists, it is shown to the user graphically as an object diagram. Fig. 3.32 (c) shows an example of a result depicted by UMLtoCSP. In particular, it considers a modified version of the running example from Figure 3.2 where the multiplicities of the roles *submission* and *manuscript* have been changed to 0..1, i.e. there can be authors that do not review papers and reviewers that do not write papers. This modified version is strongly satisfiable, as shown by the object diagram displayed by the tool which satisfies all the constraints of the model: papers have one or two authors and three referees, no author reviews his own paper, the maximum paper length is not exceeded, student papers have at least one student author, students do not act as reviewers and there are between 1 and 5 student papers.

An advantage of this architecture is that the translation and verification are completely hidden from the designer. Therefore, users of UMLtoCSP do not need any kind of knowledge of constraint programming to analyze a model and interpret the results, since the results provided by the CSP solver are reinterpreted in terms of the original UML/OCL models and returned to the user as a UML object diagram that the user can directly understand. Furthermore, they can provide the input to the tool directly as a UML class diagram in the format being used by their CASE tool.

As future work, we are exploring the full integration of UMLtoCSP as a plug-in of the Eclipse IDE, in order to further improve its usability and facilitate its use by the Eclipse community. Some extensions of the tool to accomodate a more direct verification of model transformations is also under development. For instance, for graph transformations, we have bn able to hide the analysis process behind a graphical front-end tool called AToM^3. This meta-modelling tool generates customized modelling environments for DSVLs, and allows defining model manipulations by graph transformation. In order to analyse the graph transformation rules, an OCL generator provides the input to UMLtoCSP for analyzing the rules' properties. The results are then shown back in AToM^3 using the concrete syntax of the DSVL. In this way, the analysis mechanism is kept transparent to the rule designer. This approach follows the line of *hidden formal methods* [14], which advocate an intuitive presentation of the verification results, probably in terms of the input language.

3.9 Problem Complexity and Efficiency Issues

This section discusses the complexity of the problem we are dealing with and the strategies we have followed to improve the efficiency of our method. For the sake of simplicity, we focus on the UML/OCL static models scenario.

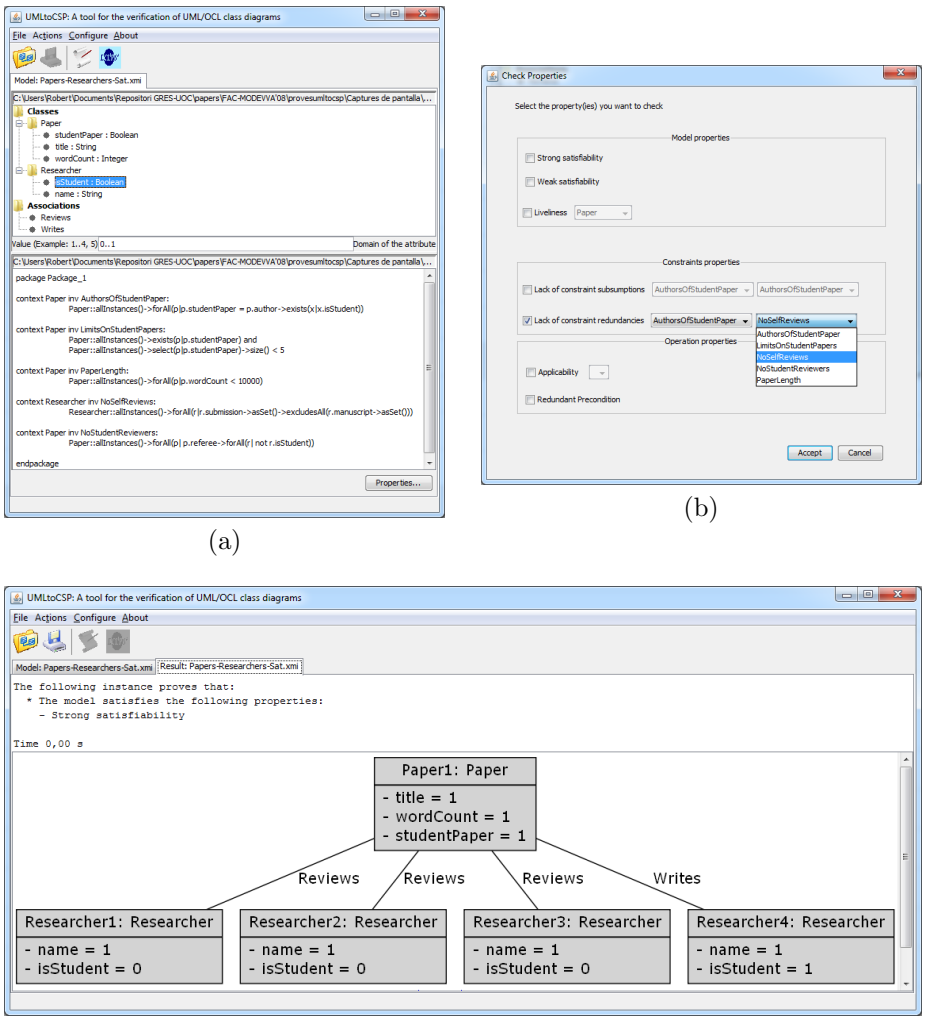


Figure 3.32: GUI of the UMLtoCSP tool: (a) loading the model, (b) selecting the property to be verified, (c) showing the result of the analysis on a modified version of the running example.

3.9.1 Problem Size and Complexity

Reasoning on UML class diagrams is EXPTIME-complete [13] and, when general OCL constraints are allowed, it becomes undecidable. Therefore, if this problem is addressed as a search problem, i.e. locating a correct or incorrect instance within the space of all possible instances, a careful analysis of this search space is required. The goal of this preliminary analysis is extracting useful heuristics that can guide the search in order to make it more efficient and ensure its termination.

The search space can be organized as a *search tree*, where each leaf corresponds to a solution (either feasible or unfeasible) and each internal node represents a decision in the search process (assigning a value to a variable from the domain of eligible values). A measure of the complexity of a problem is the *size* of this search tree, measured as the number of leaves. In general, the size is determined as the product of all the cardinalities of the domains for each variable in the problem. In order to calculate it, for the specific problem we are working in, variables and domains introduced in Section 3.4.2 are used.

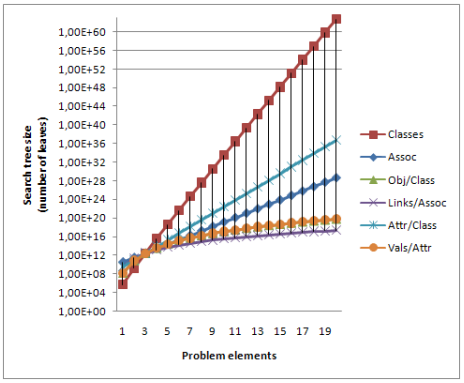
- For the classes: we consider the number of possible objects per class c ($|domain(size_c)|$) and, for each class, the number of possible values of each attribute f_i ($|domain(f_i)|$).

$$\prod_{c \in Cl} |domain(size_c)| \cdot \prod_{c \in Cl} size_c \cdot (size_c \cdot \prod_{f_i \in f} |domain(f_i)|) \tag{3.1}$$

- For the associations: we consider the number of links in each association as ($|domain(size_{as})|$) and, in each association, the number of objects in each participating class p_i ($|domain(p_i)|$).

$$\prod_{as \in As} |domain(size_{as})| \cdot \prod_{as \in As} size_{as} \cdot \prod_{p_i \in PCas} |domain(p_i)| \tag{3.2}$$

The total number of leaves can be found multiplying Eq.'s 3.1 and 3.2. It is easy to see that, even for small models, it is not possible to visit the whole tree in order to get the solution. Several parameters of the input model will affect the size of this search tree: the number of classes and attributes of the model, the number of attributes per class, the domain of each attribute and the number of allowed objects/links per class/association. In order to provide a rough idea of the magnitude of this size and the effect of these parameters, Fig. 3.33 illustrates some sample data. The graphic has been build fixing, for each parameter line, the rest of parameters to three. This is an average value and permits to analyse the evolution of the tree search size increasing a single parameter. It is easy to notice that a unitary increase in any of these parameters implies an exponential growth in the size of the search tree.



Classes Number of classes in the UML design

Assoc Number of associations in the UML design

Obj/Class Number of objects of a given class

Links/Assoc Number of links per association

Attr/Class Number of attributes of a given class

Vals/Attr Number of possible values for a given attribute

Figure 3.33: Search tree leaves depending on the size of the input model (logarithmic scale).

3.9.2 Search Strategy

Efficiency improvements can be implemented at run-time level (i.e. parametrising the CSP solver by selecting the traversal algorithm, controlling the search,...) or at design-level (choosing the right set of variables, constraints and domains for the CSP in order to optimize the search).

Back-tracking, Search Tree Pruning and Search Control

In Constraint Programming (CP) the search space defined by variables and their domains is visited by a depth-first search algorithm. Each step of the traversal process assigns a value to a variable, extending the current partial solution until a complete solution is found in a leaf. Although many other approaches have been explored, e.g. backjumping, it is backtracking the most commonly used for this traversal.

One of the drawbacks of backtracking is the *late detection of inconsistencies*, i.e. the failure of the search algorithm to find out that the branch being currently explored takes to an unfeasible solution until it reaches the end and evaluates its feasibility. In order to avoid this, constraints among variables are used by CP to obviate visiting non feasible solutions. This is achieved by propagation (see Section 3.5.3). Thus, with the help of forward checking [105] and other consistency techniques (e.g. node-consistency, arc-consistency, etc. [105]), constraints help the search engine to explore only feasible solutions, removing the rest in advance.

Thanks to these techniques, it is possible to detect that a partial solution cannot possibly be extended into a feasible solution. In this way, backtracks can be performed without having to compute each complete unfeasible solution. Thus, these techniques partially avoid the main disadvantage of backtracking mentioned before: the search engine backtracks when a decision just taken

implies no further feasible solutions and hence no need to go on that branch of the tree. These optimizations may greatly reduce the number of visited leaves in the search tree.

However, the benefits of constraint propagation cannot be quantified in general, since they are completely problem-dependent. The ECL^{i}PS^{e}solver already uses both backtracking and constraint propagation techniques by default.

Besides this, and differently from other paradigms, CP allows to control the search phase, in order to speed it up. This means that selecting the most efficient structure and traversal of the search is fundamental, and these are determined by (1) the order in which variables are assigned a value and (2) the order in which potential values are selected. Several heuristics come usually, by default, within CP languages, e.g. assign first the variable with the most constraints or assign first the smallest value within a domain. However, users can program other heuristics useful for their specific problems. There are neither better nor worse heuristics: their adequacy is dependent on the problem so it is important to check all the combinations and analyse the reasons making one better than others. This analysis can help the designer to find out certain problem properties which might guide in the improvement of search. During the tool construction, all the possible combinations of the heuristics offered by the ECL^{i}PS^e solver for variable ordering and value selection have been evaluated and prioritized. Thus, the election of these heuristics becomes totally transparent to the final user.

Further Search Improvements Apart from deciding the way the search tree is visited, some specific features of a problem could lead the programmer to make additional decisions. Typical improvements can raise from the splitting of the set of variables into independent subsets (or like in our case, into directly dependent groups), the removal of both non interesting variables and structural search tree symmetries, etc. These improvements are left for further work.

Search Design

In what follows, we present the optimization strategies we follow during the generation of the CSP.

UML/OCL Search: dependent variable subsets Given the CSP translated from a model, it is easy to detect two kinds of directly dependent variables: on the one hand, those indicating the number of instances of a class or association (i.e. the objects and the links) and, on the other hand, those representing everything contained by the objects and links themselves (i.e. oids, attributes, etc.). It is clear that the latter depend directly on the instantiation of the former ones. Thus, the objects of a class cannot be created until the number of objects of that class is already known[14]. This approach creates two dependent subsets of variables: when a variable in the first subset is bound, its corresponding

[14]There are alternative ways to model this scenario.

set of variables in the second one can be created and instantiated. Surprisingly enough, the first subset corresponds to those variables modeling the UML schema, while the second models the OCL constraints.

This peculiarity takes us to divide the search into two steps we call UML and OCL steps — or structural constraints and global constraints problems (see Fig. 3.10). In a first stage, variables related to the UML class diagram are bound, and after finding a complete instantiation for this first subproblem, objects and associations are created. From this point, a second search is performed for the second subproblem. The two main reasons for doing this are:

- The dependence between both variables subsets complicates the management of the corresponding variables. The splitting of the search simplifies this complexity.
- Often, the problem to solve does not contain OCL constraints and hence only the first part of the tree is generated and after the creation of objects and associations, these are automatically instantiated — with no backtracking at all.

The following two sections analyse the design of these two stages.

UML Search: Structural Subproblem The first CSP — the structural subproblem — is defined by:

1. Variables for the number of objects of each class, $Size_c$.
2. Variables for the number of links of each association, $Size_{as}$.
3. Constraints related to the cardinalities of associations (see Section 3.4.2): *bounds on cardinalities* and *number of links*.
4. Constraints removing symmetries (see below).

The size of the tree search for this subproblem is given by Eq.3.3.

$$\prod_{c \in Cl} |domain(size_c)| \cdot \prod_{as \in As} |domain(size_{as})| \tag{3.3}$$

Fig. 3.34 shows the evolution of the size of the search tree for this subproblem when changing input parameters. Although this is a small problem compared with the complete one, it is still EXPTIME-complete.

For this problem, the solver (with the heuristics chosen in the previous section) is able to reach a solution in a *backtrack-free* search[15]. This makes it possible to get an instant response either if there is a solution or not.

[15] A search is told to be *backtrack-free* when no backtracks are necessary to reach the solution. This usually means the model is so good it is able to guide the search avoiding all non-feasible solutions.

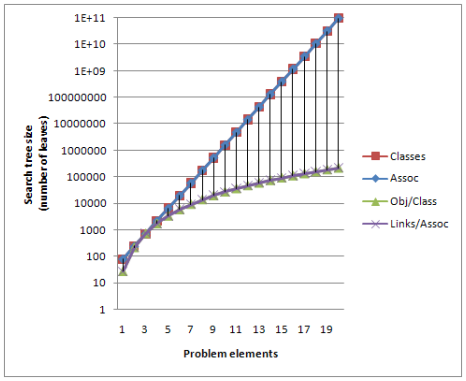


Figure 3.34: UML search tree size for different problem sizes

OCL Search: Global Constraints The second CSP — the global constraints subproblem — is defined by:

1. Variables for the objects of each class, $Instances_c$.
2. Variables for the links of each association, $Instances_{as}$.
3. Constraints related to the OCL rules of the model: *number of instances*, *distinct oids*, *uniqueness of links*, etc.
4. Constraints removing symmetries (see below).

The size of the tree search for this subproblem is given by Eq. 3.4. Notice that all but two parameters ($Instances_c$ and $Instances_{as}$) have been bound to an integer value in the previous stage. Furthermore, it is important to take into account that there is one of these search trees for every leaf of the previous stage, i.e. the solution found for the structural subproblem.

$$\prod_{c \in Cl} size_c \cdot (size_c \cdot \prod_{f_i \in f} |domain(f_i)|) \cdot \prod_{as \in As} size_{as} \cdot \prod_{p_i \in PCas} |domain(p_i)| \quad (3.4)$$

Fig. 3.35 shows the evolution of the size of the search tree for this subproblem when changing input parameters.

To improve the efficiency of this second problem, we have implemented the next two strategies.

Symmetry removal The encoding used to map a problem as a CSP should try to avoid potential *symmetries*, i.e. the fact that several assignments to variables of the CSP are equivalent because they correspond to a single solution of the original problem. Symmetries are undesirable because they cause additional overhead to the solver: it has to waste time checking the same solution several times, once for every symmetric assignment.

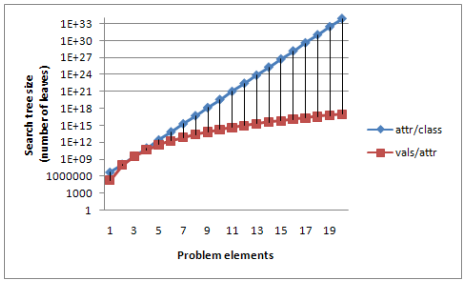


Figure 3.35: OCL search tree size for different problem sizes

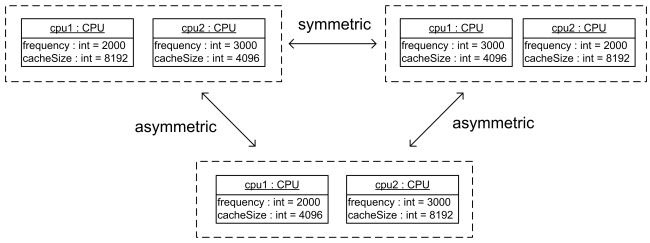


Figure 3.36: Example of symmetries in UML/OCL diagrams.

Symmetry removal is the process of ensuring that the solver considers only one assignment for each family of symmetric solutions. There are several techniques for removing symmetries from a CSP encoding. One of the most effective techniques is the inclusion of additional constraints in the CSP which forbid alternative symmetric assignments. For instance, a typical symmetry removal constraint is requiring part of the assignment to be sorted according to some ordering criteria. This can be done, for instance, when the order among different solutions does not matter: the solver will only consider the smallest solution according to the ordering, discarding the rest.

The choice of these additional constraints depends on the specific CSP encoding used for the problem. Notice however that symmetry removal constraints also cause an overhead to the solver, as they need to be evaluated during the search. For practical reasons, symmetry removal constraints should not be overly expensive to evaluate: the solver should execute faster with these symmetry removal constraints than without them. Therefore, symmetries which are very complex to detect will not be removed.

In the context of UML/OCL diagrams, there are several degrees of symmetry. First, each instance of the diagram can be abstracted as a labeled graph, where objects are the vertices, associations are the edges and each object is labeled with its type and attribute values. Intuitively, if among two instances there is a graph isomorphism that preserves labels, it means that both instances are equivalent. However, detecting graph isomorphism is computationally complex and thus trying to avoid this symmetry would be counterproductive. Instead, we will

focus on another type of symmetries caused by our encoding of instances into variables: the ordering of instances of a class or association. In our encoding, variables are assigned sequentially and therefore there is an implicit ordering among them. However, from the point of view of the instance, the ordering of objects and links is irrelevant. For instance, if there are two objects of the same class, there will be two possible assignments to their attributes which will be symmetric. That is, they all can be swapped between these two objects without changing the solution. Figure 3.36 illustrates this example for a class diagram with a single class (CPU) and two attributes (*frequency* and *cacheSize*). The two instances in the top are symmetric among them, while they are distinct from the one in the bottom. This symmetry can be removed by a adding a constraint enforcing the attributes of the objects of class CPU to be ordered lexicographically according to the pair (*frequency*, *cacheSize*). In this way, the instance on the top right would be discarded by the solver because the constraint $(3000, 4096) \leq (2000, 8192)$ does not hold. Similarly, it is possible to remove symmetries among links of an association by imposing a lexicographic ordering among links.

Ruling out irrelevant attribute and associations Reducing the number of variables and/or simplifying its structure in the CSP has also a clear impact in its complexity. Therefore, during the translation process we avoid translating those model elements that do not affect the verification process. In particular:

- We discard all attributes that do not participate in any of the constraints in the model. A correct instantiation may contain any value in those attributes.
- We discard associations that are not referenced (i.e. navigated) in any constraint and that do not state any multiplicity constraint (all participants have a "0..*" multiplicity). The population of those associations does not affect the correctness of a solution of the CSP.

The next section describes the efficiency level achieved by our tool once all the optimizations described herein have been integrated.

3.9.3 Experimental Results and Efficiency Assessment

In order to evaluate the efficiency of the tool, we have considered three scenarios based on the class diagrams in Figs. 3.37 and 3.38. These are artificial examples, where a "ring" of n classes are connected by n binary associations. These scenarios are designed in order to make it scalable for arbitrary values of n, and thus, to provide information on the scalability of the tool.

In the first scenario (Fig. 3.37), the goal is the analysis of the tool in the structural subproblem. Therefore, we consider that there are no integrity constraints and we focus on the multiplicities of association ends. This class diagram would be strongly satisfiable if all those multiplicities were 1..1, e.g. by creating

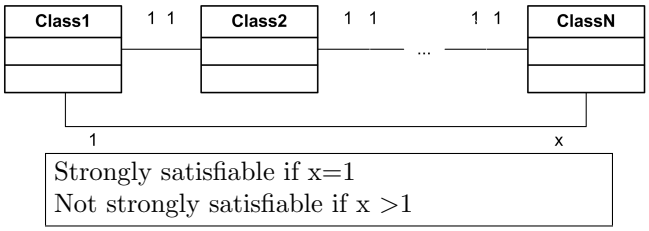


Figure 3.37: Example without OCL constraints.

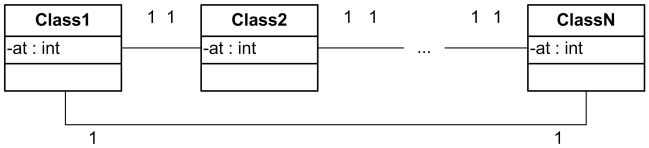
a single object of each class and connecting them through the corresponding associations to the two *neighbour* objects. However, if one of the cardinalities were 2..2, the class diagram would become unsatisfiable. In this way, it is possible to evaluate the behavior of our tool both with a satisfiable and an unsatisfiable versions of the class diagram.

In the second and third scenarios (Fig. 3.38) we are interested in considering a model with OCL constraints. Hence, we consider that all association ends have a multiplicity of 1..1, making the structural subproblem satisfiable. For the OCL subproblem, we define n constraints, each defining a relationship between the value of an attribute in class i to the value of the corresponding object in class $i+1$. Depending on the relationship operator that we choose ($>$ or $\geq$), the class diagram may be strongly satisfiable or not.

The difference among the second and third scenarios is the location of the inconsistency. The second scenario (Fig. 3.38 left) assumes that the inconsistency arise due to the incompatibility of two constraints involving *Class1* and *Class2*. In this sense, the incompatibility is localized in a fragment of the class diagram. In contrast, the third scenario considers a case where the incompatibility arises from the interaction of all constraints in the model, which establish a cyclic dependence on the values of the attributes of all classes. Precisely, the unsatisfiable version of this third scenario has been designed as the *worst-case scenario* for our approach, as all variables of a potential solution have to be assigned in order to detect that the solution is unfeasible.

These examples have been tested for models of different sizes, consisting of 2, 5, 10, 100 and 1000 classes on a Xeon 5050 3Ghz with 4Gb of RAM. All these examples have been measured with the following domains: the number of objects per class can be between 0 and 5, there are three possible values for each attribute and the number links in each association is between 0 and 10. The reported execution times consider only the verification of the model, excluding the time required to parse the input XMI and OCL files. Table 3.2 depicts the experimental results for these examples. Every cell of the table contains the search time spent to reach the final answer in the two subproblems, structural (i.e. only the UML part) and global (the full UML/OCL model), separately.

These results show that our approach very easily detects satisfiable problems (usually, not even backtracking is needed, i.e. propagation is powerful enough to guide search without failures) even when the model includes OCL



context Class1 **inv**: self.at *op* self.class2.at
context Class2 **inv**: self.at *op* self.class3.at
...
context ClassN-1 **inv**: self.at *op* self.classN.at
context Class1 **inv**: self.class2.at *op* self.at

context Class1 **inv**: self.at *op* self.class2.at
context Class2 **inv**: self.at *op* self.class3.at
...
context ClassN-1 **inv**: self.at *op* self.classN.at
context ClassN **inv**: self.at *op* self.class1.at

Strongly satisfiable if *op* is $\geq$
Not strongly satisfiable if *op* is $>$

Figure 3.38: Examples with OCL constraints: local inconsistency (left) and global inconsistency (right)

	No OCL constraints		**Local OCL inconsistency**		**Global OCL inconsistency**	
n	**Sat**	**Non Sat**	**Sat**	**Non sat**	**Sat**	**Non sat**
2	0.00s	0.00s	0.00s	3.67s	0.00s	1.56s
5	0.01s	0.00s	0.01s	3.78s	0.01s	1.84s
10	0.01s	0.00s	0.01s	3.78s	0.01s	5146.70s
100	0.14s	0.05s	0.17s	4.50s	0.17s	—
1000	3.58s	2.07s	3.81s	31.55s	4.97s	—

Table 3.2: Execution time for $n = 2, 5, 10, 100$ and 1000 classes

	Sat structural subproblem	**Unsat structural subproblem**
Sat global subproblem	MEDIUM	EASY (backtrack-free)
Unsat global subproblem	DIFFICULT	EASY (backtrack-free)
Without OCL constraints	EASY (backtrack-free)	EASY (backtrack-free)

Table 3.3: Analysis of the problems depending on the final and intermediate responses to search trees

constraints (Local and Global OCL inconsistency results). As a matter of fact, for the structural problem, being satisfiable or not, the solution is always found backtrack-free. This shows a search time which is almost independent from the number of classes of the structural problem.

From this analysis we conclude that most of the search difficulties are found in the second stage of the search, being this more complex when there is no feasible solution (i.e. when the OCL constraints make the model not satisfiable). This was expected because the absence of a positive answer is translated into the full search tree generation. Normally, early evaluation of constraints allows the detection of an unfeasible solution before assigning all of its variables, thus pruning the search tree. For example, this is what happens in the second scenario, the local OCL inconsistency, where even for large models the tool reach a conclusion in a small amount of time. However, this is not the case in the third scenario. Recall that this third scenario (global OCL inconsistency) is the worst possible case for our approach: the constraints have been carefully designed so that the inconsistency cannot be discovered until all variables have been assigned, and thus constraint propagation is not successful in prunning the tree. In this worst-case scenario, which is not expected in real-world models, efficiency is only acceptable in small input models.

To sum up, the execution time is not overly dependent on the number of classes and associations or the model, nor the number of constraints in the model. Instead, it is dependent on the characteristics of the specific problem instance received as input and the interaction among its constraints. Some problem instances, such as the worst-case example described in this section, have a prohibitive execution time for medium and large models. Meanwhile, the expected average case (where there are no inconsistencies or the inconsistencies involve a fragment of the model and not the model as a whole) show a reasonable execution time.

Table 3.3 presents an analysis of the different kinds of problems to solve — classified by their feasibility and kinds of constraints forming them. The conclusion is that, like model-checkers or SAT solvers, UMLtoCSP may be most efficient when the underlying problem is satisfiable. Long execution times are indicative of the existence of inconsistencies in the input model. Therefore, and even if this could not be used as a conclusive response, the inclusion of a timeout mechanism in the tool could help in providing an early response in those situations that are most likely representing an inconsistent global subproblem.

Table 3.4: Comparison of several methods for the verification of UML/OCL class diagrams.

Tool	**Formalism**	**Translation**	**Verification**	**Limitations**
[8, 13, 107]	Description Logics	Automatic	Automatic	No OCL support
[36, 75]	CSP	Manual	Automatic	No OCL support, bounded verification
[76]	Linear Programming	Automatic	Automatic	No OCL support
Alloy [61]	Relational Logics	Manual or [4]	Automatic	Bounded verification, limited arithmetic support
[97]	SAT	Automatic	Automatic	Bounded verification, limited arithmetic support
[109]	Syntax patterns	Automatic	Automatic	Incomplete, limited to specific constraint patterns
HOL-OCL [22]	Higher-Order Logics	Automatic	User-assisted	Undecidability
PVS [70]	Higher-Order Logics	Automatic	User-assisted	Undecidability
CQC [91, 92]	Deductive DB queries	Automatic	Automatic	Limited arithmetic support
USE [52]	ASSL	Manual	Automatic	Validation only
UMLtoCSP	CSP	Automatic	Automatic	Bounded verification

3.10 Related work

In this section, we will compare our approach with the related work in the area of static consistency analysis of class diagrams. We will not discuss extensions of this work to deal with dynamic properties, e.g. model checking or analysis of operations contracts e.g. [12, 30, 61, 93]. Furthermore, we will restrict ourselves to the application of consistency analysis to model verification and validation. Even though the examples and counter-examples computed by these tools can also be applied to test-case generation [45,110], research on model-based testing focuses on a more abstract problem, the definition of suitable testing criteria, while the generation of tests cases for a given testing criterion is solved using the tools described in this section.

Typically, approaches devoted to the verification of UML/OCL class diagrams (as our own approach) transform the diagram into a formalism where efficient solvers or theorem provers are available. However, there are complexity and decidability issues to be considered. As it was mentioned before, reasoning on UML class diagrams is EXPTIME-complete without OCL constraints and undecidable when general OCL constraints are allowed. By choosing a particular formalism, each method commits to a different trade-off regarding the verification of correctness properties of UML/OCL diagrams. Table 3.4 briefly compares the tool described in this section, UMLtoCSP, to other related tools. For each approach, the following information is listed: the underlying formalism, the translation procedure from UML/OCL to the formalism (manual or automated), the degree of automation in the verification (user-assisted or automated) and other limitations of the method. UMLtoCSP offers both automated translation and verification procedures and supports general OCL constraints. Additionally, our tool is able to provide valid instantiations for satisfiable models.

Several previous works focus on the verification of UML class diagrams *without* OCL constraints (or just with some specific types of basic constraints), i.e. the decidable version of the problem. Some approaches in this category are based, among others, on Description Logics [8, 13, 107] or Constraint or Linear Programming [36, 75, 76]. Recent results [46] have extended the description logics formalism (in short, a decidable subset of first-order logic) to define and reason on operation contracts. However, these approaches need to restrict the constructs that may appear in the model to keep the reasoning decidable. Thus, most OCL operations cannot be translated into this formalism.

It is also possible to achieve an efficient[16] analysis if only specific patterns of OCL constraints are allowed. Some examples of these *constraint patterns* are the uniqueness of an identifier or the lack of cyclic dependencies among objects. In such cases, it is possible to derive a priori the consistency lemmas required for the constraint pattern to hold. These lemmas can be checked efficiently by finding syntactical patterns in the OCL constraints [109]. An advantage of this approach is its efficiency, as it is polynomial in the size of the model contrary to the rest of methods dealing with OCL, which have an exponential worst-case behavior. On the other hand, the method is restricted to the analysis of a specific set of constraint patterns and therefore it does not support general OCL constraints. Furthermore, the method is incomplete as the analysis of syntactical patterns may be insufficient to prove the consistency lemmas, even if they hold.

Another related approach is the USE tool [52]. However, USE is more focused on validation than in verification, that is, it permits to construct finite snapshots of a UML model that satisfy a set of OCL constraints but the generation of snapshots is not supposed to be exhaustive: USE does not attempt to automatically explore a whole range of values to determine the correctness of the model. Rather, the generation process is user-driven. Users define a list of desired characteristics for the instances to be created and their number. Then, the tool generates and tests the validity of such instance set. In contrast, our approach is fully automatic and decidable. To the best of our knowledge, the approach presented here is the first method addressing the verification of UML class diagrams *with OCL constraints* based on Constraint Programming.

Regarding verification of UML class diagrams *with* general OCL constraints, some examples of formalisms used in this problem are Relational Logics (Alloy [61]), Higher-Order Logics (HOL-OCL [22, 23, 70]) and deductive database queries (CQC [91,92]). However not all these formalisms are as expressive as the UMLtoCSP method. Some approaches support only a subset of UML or OCL constructs, e.g. Alloy cannot directly manipulate operations involving integers and CQC supports only OCL expressions that compute a boolean value. Undecidability also imposes several limitations on these approaches, e.g. requiring user-interaction to complete proofs as in HOL-OCL and PVS. In some cases, it is possible to detect that the analysis of a model will be decidable and improve

[16] Though it is difficult to discuss efficiency of UML verification methods since many approaches either have not publish efficiency results achieved with them

the efficiency of the verification for that particular model [92].

Among all these approaches, the most similar in terms of features are UML2Alloy [4] and [97]. In both approaches, the underlying reasoning engine is a SAT solver: the problem and the correctness property are translated into a boolean formula whose satisfiability needs to be determined. In the case of UML2Alloy, there is an intermediate step, the transformation of the UML/OCL model into the Alloy notation, which the Alloy Analyzer internally translates into a SAT instance. Meanwhile, [97] proceeds by directly generating the SAT instance and passing it to the SAT solver MiniSAT. UMLtoCSP offers an advantage with respect to these two bounded verification approaches. In SAT-based methods, constraints involving numbers must also be expressed in terms of boolean variables, meaning that (1) users must specify the number of bits being used to encode each value and (2) operations on numbers (e.g. addition, difference, multiplication, less-than, ...) must be encoded as boolean formulas operating at the bit-level. All these factors lead to a combinatorial explosion in the size of the formula when the bit-width of integers increases. In a CSP, increasing the range of a numeric value also increases the search space, but encoding complex arithmetic expressions on integers or floats is straightforward. As an example of this problem, let us consider the running example from Fig. 3.2. When this model is written in the Alloy notation, we realize that one of the constraints (*PaperLength*) contains the integer constant 10000. Encoding this constant at the boolean level requires 15 bits per integer which requires a large amount of CPU time just to generate the SAT instance (the generation of the SAT instance did not finish after 1 hour of CPU time in an Intel Core Duo T2400 1.83Ghz with 1 Gb RAM), while UMLtoCSP computes the result in less than one second. Therefore, any model where it is not possible to use "small" constants and avoid floating point values would be a good candidate to be analyzed with UMLtoCSP.

Another benefit of UMLtoCSP with respect to Alloy is an advantage in terms of usability. UML2Alloy and Alloy are separate tools, meaning that a user has to launch UML2Alloy, load the model and translate it, then launch Alloy, load the translation and verify it. Meanwhile, UMLtoCSP offers an integrated environment for verification which also supports the Ecore format, and therefore, the range of Eclipse EMF tools. On the other hand, it should be noted that Alloy is a mature tool, with a consolidated implementation. For example, some interesting features of Alloy which are not supported by UMLtoCSP are bounded model checking capabilities and the computation of *unsatisfiable cores* [104], i.e. minimal sets of conflicting constraints within the model.

Unlike other approaches, our approach does not impose theoretical limitations that restrict any UML or OCL constructs besides those explicitly mentioned early on. On the other hand, like all bounded verification methods our approach is decidable but *not complete*: results are only conclusive if a solution to the CSP is found. In that sense, our method only guarantees that if a solution to the CSP exists within the parameters provided by the user, it will be discovered. Nevertheless, the absence of solutions within a finite search space cannot be used as a proof: a solution may still exist outside the search space

defined by the parameters. An observation which alleviates this limitation is the *"small scope hypothesis"*, i.e. it is possible to identify a large percentage of errors in system by considering all possible instances within small domain. This limitation is shared by Alloy, while HOL-COL and the CQC method provide complete proof procedures.

Nonetheless, an efficient decidable procedure may provide more useful information than a semidecidable procedure, even if the answer is not conclusive. For example, when checking for satisfiability, the maximum population value for classes and associations can be always kept low. In practice, it may be as problematic to have a non-satisfiable model as to have a model that to be satisfiable requires populating the classes with too many instances, e.g. a model that requires creating more than fifty instances of each class to be satisfiable may be unusable in practice and may deserve further inspection anyway.

Our approach can be complemented with other verification approaches addressing other kinds of behavioural UML diagrams (as state machines) in order to provide more global results.

A similar scenario applies to the verification of model transformations, There are two main sources of related work in the analysis of graph transformation rules: those analysing rules using DPO and SPO theory, and those that translate rules to other domains for analysis. In the former direction, graph transformation has developed a number of analysis techniques [47, 57, 95], but they usually work with simple type graphs (i.e. without OCL constraints).

Regarding the transformation of graph rules into other domains, their translation into OCL pre- and post-conditions has been also proposed in [53]. Here we give a more complete OCL-based characterization of rules that considers DPO and SPO semantics, NACs, and that encodes the LHS's matching algorithm as additional pre-conditions. In [53] the match is passed as parameter to the OCL expressions, assuming a predefined existing external mechanism. In addition, we exploit the resulting OCL expressions in order to enable the tool-assisted analysis of different rule properties. In [10], rules are translated into Alloy in order to study the applicability of sequences of rules and the reachability of models. Even though Alloy is equipped with a SAT solver (so that similar properties to the ones we verify could be analysed), the properties in [10] are only related to reachability. Moreover, the integration with meta-model integrity constraints is not discussed. In our case, an encoding of reachability properties like the one proposed by the authors is also possible, but left for future work.

In this line of work, other approaches like [108] rely on model-checking techniques to analyse reachability and invariants. In particular, in [108], rules, models and meta-models are transformed into Promela for model-checking with SPIN. Finally, in [94] a transformation of rules into the rewriting logic system Maude is proposed, where reachability analysis and LTL model checking is performed, using the Maude model checker. These approaches do not take into account meta-models with integrity constraints or OCL constraints in rules. However, there are several efforts for supporting OCL in Maude, and integrating OCL in the approach of [94] is feasible. Our use of OCL as intermediate

representation has the benefit that it is a tool independent, standard language, and moreover we can easily integrate attribute conditions and meta-model constraints. Finally, the kind of properties we analyse is also different. While these approaches focus on reachability and model checking, our use of UMLtoCSP – and the fact that it formulates the UML/OCL model as a CSP – makes it possible to analyse properties like applicability, overlapping and executability of rules, based on model finding capabilities, in contrast with space-state generation and exploration techniques.

3.11 Summary

We have presented a fully automatic, decidable and expressive method for the formal verification of UML/OCL class diagrams (and its application to the verification of model transfomraitons). Our method is based on the translation of the class diagram into a CSP. This approach has been implemented in a prototype tool [106].

As a trade-off the verification procedure is not complete: the user must provide a set of parameters to limit the search space. Our procedure guarantees that this search space will be explored exhaustively. We believe this is a reasonable trade-off given the advantages of our method with respect to alternative approaches. However, if desired, it is also possible to use the transformation of UML/OCL models into CSPs on infinite domains: constraint solvers also allow an *incomplete search* [5] although termination is not guaranteed and depends on heuristics to guide the search process. In that way, our method would become semidecidable but complete (for properties that can be satisfied by *finite* instances). Moreover, the instance generation nature of this approach (i.e. the fact that properties are proven by creating legal instances of the model), makes it amenable for model validation or test generation purposes.

We believe our approach is a promising starting point to face the verification grand challenge described in the further work chapter.

Chapter 4

Future Research Directions

Building on the results on Model-driven engineering achieved so far I believe we can conclude that MDE has reached a maturity level where core tools and techniques for defining and manipulating models (like the ones described at the beginning of the document) are already available.

Nevertheless, the widespread use of MDE is raising new challenges that may impair the increasing adoption of MDE in practice (affecting the benefits this would bring to software engineers):

1. **MDE scalability**. Current MDE techniques do not scale and thus fail to work satisfactorily once we go beyond toy examples. This is making some companies to rethink their MDE strategy and even consider going back to programming as a way to solve their efficiency issues.

2. **Model quality**. Despite the key importance of the model concept in MDE, research on model quality is still very preliminary. As an example, no popular modeling tool includes any kind of quality checks for models beyond the simple conformance relationship (i.e. making sure that a model is a correct instantiation of its metamodel). This is a fundamental concern, e.g. code is derived from the models so wrong models will generate flawed software implementations. We have previously described our approach for model verification. We believe this approach can used as the core component of a new approch for model quality analysis.

3. **Integration with legacy systems**. Most software implementations do not start from scratch but are reengineered/wrapped versions of existing legacy code. Thus, we really need to better *understand* these legacy systems in order to be able to successfully integrate them in MDE (software modernization) processes.

4. **Collaborative Development**. All development process encourage an active participation of the end-users in the development process. Nevertheless, the specification process of modeling artefacts as important as the

(domain-specific) languages those users will employ to model their systems do not yet take the users into account.

I believe research in MDE should focus on addressing these challenges in the next years. In this sense, I would like to finish this HdR scientific report by sketching my (and my team) research program for the next years based on four different axis, each one corresponding to one of the challenges mentioned above. These axes offer a good combination of core vs application-oriented and of mid-term vs long-term research problems. This research will be done following our team objectives of maintaining scientific excellence at the international level through publication in top level journals and conferences, continuing the production of high quality open source software and conducting technology transfer activities, always in collaboration with our research and industrial partners.

4.1 Very Large Models

As in other disciplines [1], manipulation of large artifacts poses specific problems that must be properly addressed in order to be able to use the technology in real industrial scenarios. We predict the same situation will be an important issue in MDE. In fact, our industrial partners have already experienced problems in this sense, especially when dealing with the huge models typically resulting from reverse engineering processes.

Therefore, as part of our research program, we plan to open a new research area focused on the development of scalable techniques for model manipulation, inspired whenever possible in how other technical spaces have faced similar problems. Some of the research lines we believe promising in this area are:

- Incremental, lazy and infinite execution modes for model transformations. Keeping as much as possible the syntax of existing languages (so that users do not need to learn new ones) we plan to provide new execution modes that can be used when manipulating huge models. Incremental model transformation refers to the ability of modifying only the possibly affected subset of the target model when changing the source model (instead of re-executing the whole transformation). A lazy model transformation only computes a subset of the target model on-demand, i.e. when the user wants to access that part (avoiding an initial costly computation of the full target model). In the infinite execution mode, the transformation cannot wait for the source model to be completely available (e.g. the source model can be a data stream) and must be able to output pieces of the target model as soon as a subset of the source becomes available. All of them will be implemented in our ATL transformation language.

- Modeling as a Service (MaaS). We believe it is worth researching the use of cloud infrastructures for the execution of scalable modeling services. This

[1] For instance, in the database community, the Very Large DataBases Conference is one of the top three in the area and is celebrating now its 37th edition.

possibility has not yet been explored. We are confident that the technical challenges to make MaaS a reality can be overcome by learning from how SaaS has developed. Two key issues here would be the development of parallel model manipulation algorithms (to maximize the benefits of the cloud), and the definition of modeling service description (and choreography) languages.

- Query languages and scripts for batch processing of models and model repositories. More and more, software processes generate huge and heterogeneous modeling artifacts that get stored in model repositories. We need to be able to automatically retrieve and process (including as part of complex transformation chains) all these artifacts. Existing model search approaches (like [20], [74]) are more focused on searching by model content than by model structure and inter/intra-model relationships, and offer limited model processing capabilities for the retrieved results.

- A virtualization mechanism for models. A virtual model is presented to the user as a normal model but it is internally designed as a set of references to the model elements in the base model/s. This mechanism is especially useful as part of model composition scenarios, where instead of creating the composed model we can immediately create a virtual representation of such composed model (regardless the composition and merge strategy employed [50, 58, 68, 90]) that will forward all the read/write requests on it to the contributing models. This model virtualization approach will be contributed to the new EMF Facet [2] Eclipse project in which AtlanMod is participating with industrial partners.

4.2 Pragmatic formal model verification

The verification techniques described in the previous section have shown promising results. Nevertheless, there are still several open issues to be improved before model verification is widely adopted by practitioners as one of the usual development practices in their daily work.

As a first step we envision the development of a *quality framework* in charge of automatically selecting the most appropriate analysis technique for an input model depending on the characteristics of the model (e.g. complexity, expressiveness and so on) and the target quality property. Each available technique presents a different trade-off regarding the verification process employed. Depending on the model one approach may be more suited than others. For instance, for UML models without integrity constraints (a decidable problem) it may be better to use complete approaches (as those based on Description Logics) instead of approaches based on bounded verification as the ones highlighted in this document. Again, we cannot assume the designer has enough expertise on formal methods to decide himself which family of approaches works better

[2] http://www.eclipse.org/modeling/emft/facet/

for the particular modeling scenario he is interested in. Therefore, a framework able to reason on the characteristics of the input problem to decide the best problem solving strategy is required.

A second set of improvements is related to the scalability of the verification methods. When following our pragmatic formal verification approach we can tame the exponential complexity of the verification problem but still, all available techniques need to be improved to cope with real industrial scenarios. Some of the possible research lines we believe promising in this area are:

- **Model partition to improve performance.** In most cases, the verification of a model m can be defined in terms of the verification of the submodels $m_i, \ldots, m_n$. Techniques for slicing the model in a subset of independent submodels (with the subsets to be computed depending on the property to be verified) will definitely help in improving the efficiency of the process due to its exponential nature.

- **Search space reduction for bounded methods.** Bounded methods require a finite search space. A smaller search space improves the efficiency but impairs the completeness of the verification. A preliminary analysis of the model could provide some insight on the best bounds of the search space as a trade-off between the two properties.

- **Apply SAT Modulo Theories to model satisfiability.** SAT Modulo Theories (SMT) is a promising technique for checking the satisfiability of a complex formula which combines recent improvements in SAT tools with the power of a custom solver specialised in a given logic [80]. In the case of model satisfiability, the challenge is identifying a subset of the modeling language which is sufficiently expressive yet allows efficient decision procedures.

- **Incremental verification.** The specification of a model is an iterative process where the model is continuously refined by means of adding, changing or deleting some of its elements. Clearly, once a first version has been verified, we should be able to prove the correctness of new model versions without verifying the whole model again. Instead, only the "updated" parts should be considered. This would make a significant impact in the verification process since it would be fast to reverify models as part of the normal model evolution process in new development iterations.

- **Model normalization.** Normalizing a model, i.e. rewriting complex modeling constructs in terms of more basic ones, prior to the verification process helps to reduce the complexity of the verification algorithms that now do not need to consider the full language expressiveness. For instance, see [34] for some rules for normalizing OCL constraints.

Another very important area that has not yet been addressed by current methods is the problem of the quality of the feedback received by the user when the verification process detects that a model is not correct. Tools should not

only be able to answer whether the model is correct. If the answer is no, the tool should be able to explain where, why and how it can be corrected. This information has to be provided in terms of the original model (i.e. in a way that the designer can understand, not in terms of the internal verification language used by the tool). This will help designers to "easily" improve defective models. A promising approach in this direction is the use of *explanation-based constraint solvers* [37], i.e. constraint solvers which are capable of identifying the subset of constraints causing the inconsistency of a CSP. This subset of constraints could then be traced back to the subset of the model that generated them to provide valuable feedback to the designer.

Finally, a general problem of the area is the lack of standard and public community benchmarks that facilitate the comparison among different tools and approaches. This is necessary since benchmarks provide an excellent resource to measure the progress and significance of a given contribution. The existence of widely accepted benchmarks for model analysis can foster progress and allow also other existing approaches to mature and exchange ideas.

4.3 Modernization of legacy systems.

Legacy systems embrace a large number of technologies, making the development of tools to cope with legacy systems evolution a tedious and time consuming task. As modernization projects have to address different combinations of legacy technologies and various modernization scenarios, model-driven approaches and tools offer the required abstraction level to build up mature and flexible modernization solutions.

Our work in this area started with the MoDisco Eclipse project[3]. MoDisco [24] provides an extensible framework to develop model-driven reverse engineering tools to support use-cases of existing software modernization. Modernizing an existing software system implies describing the information extracted out of the artifacts of this system, understanding the extracted information in order to take the good modernization decisions and transforming this information to new artifacts facilitating the modernization (metrics, document, transformed code, etc).

MoDisco aims at supporting the reverse engineering process by providing metamodels to describe existing systems (i.e. our Java metamodel allows a complete representation of all Java code in the system, including not only the structure of the classes but also the complete modeling of the methods behavior), discoverers to automatically create models of existing systems and generic tools to transform complex models created out of those systems. So far, model-driven reverse engineering approaches have targeted specific technologies (e.g. COBOL [9]) or focused on specific activities (like software data analysis [6] using the Moose re-enginering platform) while Modisco pretends to provide a more generic framework where specific discoverers for concrete technologies and new use cases can be easily integrated.

[3] http://www.eclipse.org/MoDisco/

Nevertheless, MoDisco and in general all reverse engineering tools have a very important limitation, they do not help engineers to really understand the legacy system, they just re-express it using a different (abstract and concrete) syntax. Software engineers are still in charge of processing and querying the model in order to extract the relevant information they need to understand the semantics of the system. For instance, they would need to reverse-engineer design patterns used in the implementation to get a more conceptual/domain-oriented view, deduce the business rules enforced by the code from the low-level program behavior and so on. We believe it is necessary to provide a new generation of (model driven) reverse engineering methods that close the gap between the software implementation and the conceptual representation of the system expected by the end-user. The three main research topics we plan to develop in this area are:

- Reverse engineering of business rules. So far, MoDisco provides a model-based representation of the code. This is not enough to directly discover and understand the business rules embedded in it. We envision a more complete model-driven business reengineering process in which the low-level models can be analyzed using a pattern-based approach to extract the hidden rules implemented in the code. These rules then can be expressed using a business rule language, like the OMG standard SBVR specification readable by the business people (e.g. a similar approach but starting from UML/OCL models instead of from code would be [32]). Big companies with large code bases have expressed their interest in following the results of this work.

- Reverse Engineering of the (enterprise) architecture of the system. Right now, reverse engineering focuses on single system components, e.g. we can do reverse engineering of a set of Java classes, a relational database and a set of JSP pages but what we cannot do at the moment is to reverse engineer the system as a whole, where elements in one component model are automatically linked to the elements in the others according to their semantic relationship. The final goal is to be able to reverse engineer the full enterprise architecture embedded in the information system, and represent it using a standard for enterprise architectures specification like TOGAF. This global view of the system will remarkably facilitate the software modernizations that stakeholders must take.

- Automatic creation of metamodels and model discoverers from the source code [60], required to start the reverse engineering process. This should include not only the programs but also the (external) APIs used by the running applications.

These results will be contributed to the MoDisco Eclipse project. Similarly to what we have done with ATL, we have developed a partnership with MIA-Software (Sodifrance group) for the industrialization of MoDisco. This agreement will provide resources to ensure the long-term availability of the project.

IBM is funding a PhD student to collaborate on the first item in the list. This research line is closely related to the very large models axe, since reengineering usually creates huge models that push the limits of existing modeling platforms.

4.4 Collaborative Development of Languages

It's well known that domain-specific languages can improve the productivity of end-users in focused domains. Nevertheless, to guarantee that the language fits the users' needs, it's important that end-users (which, in the end, are the domain experts) have a direct and continuous participation in the creation and evolution of the language. This includes recording the proposals, comments, alternative solutions and, in general, all the argumentation that takes place during the language creation so that at any time is possible to trace back to the reasons that motivated a given design decision. It's also important to monitor how users employ the language in order to automatically detect (and suggest for validation) irrelevant language constructs, missing ones, etc.

Collaboration should be done not only at the abstract syntax level (i.e. when discussing the concepts that must be included in the language) but also at the concrete syntax level (i.e. when deciding which particular graphical or textual elements should be use as a notation to represent the domain concepts).

Chapter 5

Dissemination and Industrialization strategy

AtlanMod strong focus on the dissemination and industrialization of its research results is one of the main assets of the team. To conclude this document, in what follows, we detail the team's strategy regarding the open source communities, the industrialization of our research prototypes and the standardization bodies. We will emphasize even more this strategy in the next research period.

5.1 Open source community and Eclipse

All the software components developed in the context of AtlanMod are released as open source. This fundamental choice has been made several years ago with the objective of allowing a worldwide public dissemination of both the implemented tools and the underlying research ideas. Quite naturally, Eclipse has been selected as the base open source community and platform for our experimentations, benefiting from the Eclipse Public License (EPL) as an efficient way to largely facilitate the collaboration with industrial partners (cf. next subsection).

In the past years, this has led us to the creation, development and dissemination of several official Eclipse projects. The most remarkable one until now is the M2M-ATL project providing the AtlanMod Transformation Language as well as the corresponding integrated tooling and resources. This project has been actually used as an important support for communication of our various research results on model transformation (and related topics like model weaving, with the integration of the AtlanMod Model Weaver, AMW, inside the ATL project). Later, we created the MDT-MoDisco project with the goal of offering a generic and extensible framework for the elaboration of model-driven reverse-engineering solutions. Following an approach similar to the ATL project, MoDisco has been used as a very interesting (and relevant) playground for our research around the application of MDE core principles and techniques to the

reverse-engineering field. The project also includes the use of global model management techniques, as developed in the AtlanMod MegaModel Management (AM3) prototype which is now part of MoDisco. In both cases, the Eclipse projects have exponentially increased the visibility of the team in the corresponding fields and opened the door to several collaborations.

Another kind of dissemination we recently experimented on, still within the Eclipse community, concerns the creation of Eclipse Lab (Eclipse-tagged Google Lab) for some of our more immature prototypes. For instance, we have recently created the Portolan Eclipse lab as a quick and efficient way to make available to the community the result of our collaboration with BNP Paribas around the topic of model driven cartography.

Generally, one of the main goals of AtlanMod is to capitalize on these successful experiences in order to continue progressing on the open dissemination of our research results. An example is the recently created EMFT-EMF Facet project, as a spin-off of MoDisco, which is providing a generic framework for dynamic model extension. Such a project offers us a good public visibility and very interesting possibilities in terms of research experimentations for the future. Moreover, for the same reasons, we are working on the creation of another official Eclipse project called EMF Quality as an initiative to support the new research line on the general problem of model quality evaluation. Obviously, even if largely involved in the Eclipse community, we still continue to be receptive to possibly interesting ideas and progresses made in the context of other different worldwide communities such as notably Google or Microsoft MSDN.

5.2 Industrialization strategy

Historically, AtlanMod has always had fruitful collaborations with companies or groups of different sizes and domains of activity. Although the name and type of the companies we are working with may differ according to the context and topic, the general principles we try to systematically apply remain the same. Based on our experience, we have extracted an industrialization triangle business model which we have already successfully applied several times.

We believe this model is a viable solution to one of the main problems of any research group. Research groups develop plenty of tools aimed at solving real industrial problems. Unfortunately, most of these tools remain as simple proof-of-concept tools that companies consider too risky to use due to their lack of proper user interface, documentation, completeness, usability, support, etc. Therefore, most of the tools are only used to convince reviewers to accept a research paper and then are completely forgotten due to lack of resources to invest in tool development (funding for non-core research activities is very difficult to get/justify). This is very bad for research groups that risk missing the opportunity of having a large user base for their tools along with the benefits that this brings to the table (e.g. empirical validation of their research, feedback, visibility, collaboration opportunities and so on).

The solution we have adopted in the AtlanMod team is to pursue the indus-

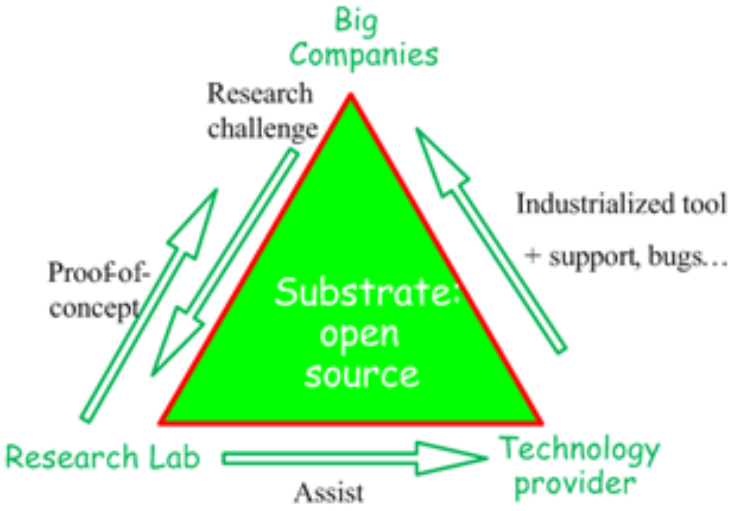


Figure 5.1: Our industrialization strategy.

trialization of our research tools by developing a partnership with a technology provider that ensures the existence of an open source but commercial-quality version of the tool. As part of the agreement, the technology provider commits resources on the non-core aspects of the tool and takes over traditional software development and maintenance tasks (including performance and usability improvements, bug fixing and user support) in exchange for visibility and the possibility of offering specialized services around it (e.g. trainings or customizations to specific clients). In our experience, this is a sustainable business model for the technology provider and very beneficial for the research group.

Obviously, in order to make sense for the technology provider to invest on the tool, the tool has to be valuable to a big community of users (or big companies). That's why in fact our strategy involves three different actors with different characteristics:

First, the big company or large user group proposes concrete use cases that they would like to have a solution for. According to the formulated request, the research lab checks if we are in front of a challenging research problem and, if so, it provides its scientific knowledge and vision in order to elaborate solutions for it. Finally, the technology provider brings the technical expertise required for the developed prototypes to be implemented as high-quality industrialized solutions directly deployable into the real environments. This is an application-driven approach which is obviously supported by an iterative process implying frequent exchanges at each of its steps. Even if not strictly required, the use of open source as a base largely facilitates both the communication between the different involved partners and the progressive dissemination of the research and industrialization results.

Actually, the fundamental property of this business model is that it is a

win-win approach. The big companies can benefit from a real expertise, both scientifically and technically. The technology providers can make significant returns on investment by working for these big groups on innovative techniques while knowing from the beginning that there's indeed a need for that product and that the solution is feasible (as proved by the research lab). Research labs can have a priority access to research challenges practically relevant from the industrial perspective and benefit from the availability of high-quality tools (as built by the technology provider) to disseminate their research results.

AtlanMod has already applied concretely this approach, via the partnership with Obeo, on various topics related to the use of model transformation with ATL. A similar process has also been performed, focusing on the reverse engineering domain, via the partnership with Mia-Software on MoDisco. Our objective is to apply again this same approach on other relevant topics and possibly different contexts.

5.3 Standardization

Using and contributing to industry standards (de jure or de facto) is also an important activity for research teams. AtlanMod has supported different standardization initiatives such as notably the Object Management Group (OMG) Model Driven Architecture (MDA) ones. In fact, the ATL model transformation tool was historically relying on the Meta Object Facility (MOF) specification for (meta)model representation. The ATL language in itself is QVT-like, respecting the main recommendations from the Query/View/Transformation (QVT) specification (for which it was one of the influences) such as the use of the Object Constraint Language (OCL) for model navigation. Following the general evolution of Model Driven Engineering (MDE) and the emergence of the Eclipse platform, our tools progressively switched to the use of the Eclipse Modeling Framework (EMF) which is nowadays considered as the de facto standard by the IT community. Thus, for evident integration and interoperability reasons, all our prototypes and tools are today based on EMF for the model manipulation part. As explained above, AtlanMod is very active in the Eclipse community, proposing and leading several Eclipse projects that have become standards in it.

More recently, within the specific context of MoDisco, we started a closer collaboration with a particular group from the OMG: the Architecture Driven Modernization (ADM) Task Force. This has resulted in the official recognition of MoDisco by the OMG ADM initiative that has chosen MoDisco as the reference implementations for several ADM standards inside MoDisco: the Knowledge Discovery Metamodel (KDM), the Abstract Syntax Tree Metamodel (ASTM) and the Software Metrics Metamodel (SMM).

The OMG is still the main actor concerning the standardization of the MDE domain, and we are looking forward to other possibly relevant collaborations or applications in the future. However, we continue to keep an eye on the other major standardization organisms such as the World Wide Web Consor-

tium (W3C), the Organization for the Advancement of Structured Information Standards (OASIS), etc.

Bibliography

[1] D. H. Akehurst, S. Kent, and O. Patrascoiu. A relational approach to defining and implementing transformations between metamodels. *Soft. and Syst. Mod.*, 2(4):215–239, 2003.

[2] M. Albert, J. Cabot, C. Gómez, and V. Pelechano. Automatic generation of basic behavior schemas from uml class diagrams. *Software and System Modeling*, 9(1):47–67, 2010.

[3] M. Albert, J. Cabot, C. Gómez, and V. Pelechano. Generating operation specifications from uml class diagrams: A model transformation approach. *Data Knowl. Eng.*, 70(4):365–389, 2011.

[4] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. UML2Alloy: A challenging model transformation. In *ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MODELS 2007)*, volume 4735 of *Lecture Notes in Computer Science*, pages 436–450, 2007.

[5] K. R. Apt and M. G. Wallace. *Constraint Logic Programming using* ECL$^{\text{i}}$PS$^{\text{e}}$. Cambridge University Press, Cambridge, UK, 2007.

[6] G. Arévalo, S. Ducasse, S. E. Gordillo, and O. Nierstrasz. Generating a catalog of unanticipated schemas in class hierarchies using formal concept analysis. *Information & Software Technology*, 52(11):1167–1187, 2010.

[7] A. Artale, D. Calvanese, and A. Ibánez-García. Full satisfiability of UML class diagrams. In J. Parsons, M. Saeki, P. Shoval, C. Woo, and Y. Wand, editors, *Proc. of Conceptual Modeling (ER'2010)*, volume 6412 of *Lecture Notes in Computer Science*, pages 317–331. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-16373-9-23.

[8] M. Balaban and A. Maraee. A UML-based method for deciding finite satisfiability in Description Logics. In *Proc. of the 21st International Workshop on Description Logics (DL'2008)*, volume 353 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[9] F. Barbier, S. Eveillard, K. Youbi, O. Guitton, A. Perrier, and E. Cariou. *Model-Driven Reverse Engineering of COBOL-Based Applications*, pages 283–299. Morgan Kaufmann, 2010.

[10] L. Baresi and P. Spoletini. On the use of alloy to analyze graph transformation systems. In A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, and G. Rozenberg, editors, *ICGT*, volume 4178 of *Lecture Notes in Computer Science*, pages 306–320. Springer, 2006.

[11] A. Baruzzo and M. Comini. Static verification of UML model consistency. In D. Hearnden, J. S, N. Rapin, and B. Baudry, editors, *3rd Workshop on Model Design and Validation (MoDeV2a)*, pages 111–126, 2006.

[12] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

[13] D. Berardi, D. Calvanese, and G. D. Giacomo. Reasoning on UML class diagrams. *Artificial Intelligence*, 168:70–118, 2005.

[14] D. Berry. Formal methods: the very idea. Some thoughts about why they work when they work. *Science of Computer Programming*, 42(1):11–27, 2002.

[15] J. Bézivin, F. Büttner, M. Gogolla, F. Jouault, I. Kurtev, and A. Lindow. Model transformations? transformation models! In Nierstrasz et al. [79], pages 440–453.

[16] J. Bézivin, F. Büttner, M. Gogolla, F. Jouault, I. Kurtev, and A. Lindow. Model transformations? transformation models! In Nierstrasz et al. [79], pages 440–453.

[17] J. Bézivin, F. Büttner, M. Gogolla, F. Jouault, I. Kurtev, and A. Lindow. Model transformations? Transformation models! volume 4199 of *LNCS*, pages 440–453. Springer, 2006.

[18] J. Bézivin and O. Gerbé. Towards a precise definition of the omg/mda framework. In *ASE*, pages 273–280. IEEE Computer Society, 2001.

[19] A. Borgida, J. Mylopoulos, and R. Reiter. On the frame problem in procedure specifications. *IEEE Trans. Software Eng.*, 21(10):785–798, 1995.

[20] A. Bozzon, M. Brambilla, and P. Fraternali. Searching repositories of web application models. In B. Benatallah, F. Casati, G. Kappel, and G. Rossi, editors, *ICWE*, volume 6189 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2010.

[21] M. Brambilla, J. Cabot, and S. Comai. Extending conceptual schemas with business process information. *Adv. Software Engineering*, 2010, 2010.

[22] A. Brucker and B. Wolff. Semantics, calculi, and analysis for object-oriented specifications. *Acta Informatica*, 46:255–284, 2009. 10.1007/s00236-009-0093-8.

[23] A. D. Brucker and B. Wolff. The HOL-OCL book. Technical Report 525, ETH Zurich, 2006.

[24] H. Bruneliere, J. Cabot, F. Jouault, and F. Madiot. Modisco: a generic and extensible framework for model driven reverse engineering. In C. Pecheur, J. Andrews, and E. D. Nitto, editors, *ASE*, pages 173–174. ACM, 2010.

[25] J. Cabot. From declarative to imperative uml/ocl operation specifications. In Parent et al. [88], pages 198–213.

[26] J. Cabot. From declarative to imperative UML/OCL operation specifications. In Parent et al. [88], pages 198–213.

[27] J. Cabot, R. Clarisó, E. Guerra, and J. de Lara. A uml/ocl framework for the analysis of graph transformation rules. *Software and System Modeling*, 9(3):335–357, 2010.

[28] J. Cabot, R. Clarisó, E. Guerra, and J. de Lara. Verification and validation of declarative model-to-model transformations through invariants. *Journal of Systems and Software*, 83(2):283–302, 2010.

[29] J. Cabot, R. Clarisó, and D. Riera. Verification of UML/OCL class diagrams using constraint programming. In *MoDeVVa 2008. ICST Workshop*, page available online: http://gres.uoc.edu/pubs/MODEVVA08.pdf, 2008.

[30] J. Cabot, R. Clarisó, and D. Riera. Verifying UML/OCL operation contracts. In *Proc. 7th Int. Conf. on Integrated Formal Methods (IFM'2009)*, volume 5423 of *Lecture Notes in Computer Science*, pages 40–55. Springer-Verlag, 2009.

[31] J. Cabot, A. Olivé, and E. Teniente. Representing temporal information in uml. In P. Stevens, J. Whittle, and G. Booch, editors, *UML*, volume 2863 of *Lecture Notes in Computer Science*, pages 44–59. Springer, 2003.

[32] J. Cabot, R. Pau, and R. Raventós. From uml/ocl to sbvr specifications: A challenging transformation. *Inf. Syst.*, 35(4):417–440, 2010.

[33] J. Cabot and R. Raventós. Conceptual modelling patterns for roles. pages 158–184, 2006.

[34] J. Cabot and E. Teniente. Transformation techniques for OCL constraints. *Science of Computer Programming.*, 68(3):179–195, 2007.

[35] J. Cabot and E. Teniente. Incremental integrity checking of uml/ocl conceptual schemas. *Journal of Systems and Software*, 82(9):1459–1478, 2009.

[36] M. Cadoli, D. Calvanese, G. D. Giacomo, and T. Mancini. Finite satisfiability of UML class diagrams by Constraint Programming. In *Proc. of the 2004 International Workshop on Description Logics (DL'2004)*, volume 104 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2004.

[37] H. Cambazard and N. Jussien. Identifying and exploiting problem structures using explanation-based constraint programming. *Constraints*, 11(4):295–313, 2006.

[38] CHIP V5. `http://www.cosytec.com/production\_scheduling/chip/optimization\_product\_chip.htm`.

[39] Comet. `http://dynadec.com/`.

[40] Cream. `http://bach.istc.kobe-u.ac.jp/cream/`.

[41] J. de Lara, R. Bardohl, H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. Attributed graph transformation with node type inheritance. *Theor. Comput. Sci.*, 376(3):139–163, 2007.

[42] J. de Lara and E. Guerra. Pattern-based model-to-model transformation. volume 5214 of *LNCS*, pages 426–441. Springer, 2008.

[43] J. de Lara and H. Vangheluwe. Defining visual notations and their manipulation through meta-modelling and graph transformation. *J. Vis. Lang. Comput.*, 15(3-4):309–330, 2004.

[44] B. Demuth. The Dresden OCL toolkit and its role in Information Systems development. In *Proc. of the 13th International Conference on Information Systems Development (ISD'2004)*, Vilnius, Lithuania, 2004.

[45] T. T. Dinh-Trong, S. Ghosh, and R. B. France. A systematic approach to generate inputs to test uml design models. In *17th International Symposium on Software Reliability Engineering (ISSRE'2006)*, pages 95–104. IEEE Computer Society, 2006.

[46] C. Drescher and M. Thielscher. Integrating action calculi and description logics. In J. Hertzberg, M. Beetz, and R. Englert, editors, *KI*, volume 4667 of *Lecture Notes in Computer Science*, pages 68–83. Springer, 2007.

[47] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer, 2006.

[48] H. C. Esfahani, J. Cabot, and E. S. K. Yu. Adopting agile methods: Can goal-oriented social modeling help? In P. Loucopoulos and J.-L. Cavarero, editors, *RCIS*, pages 223–234. IEEE, 2010.

[49] H. C. Esfahani, E. S. K. Yu, and J. Cabot. Situational evaluation of method fragments: An evidence-based goal-oriented approach. In B. Pernici, editor, *CAiSE*, volume 6051 of *Lecture Notes in Computer Science*, pages 424–438. Springer, 2010.

[50] F. Fleurey, B. Baudry, R. B. France, and S. Ghosh. A generic approach for automatic model composition. In H. Giese, editor, *MoDELS Workshops*, volume 5002 of *Lecture Notes in Computer Science*, pages 7–15. Springer, 2007.

[51] GNU-Prolog. `http://www.gprolog.org/`.

[52] M. Gogolla, J. Bohling, and M. Richters. Validating UML and OCL models in USE by automatic snapshot generation. *Journal on Software and System Modeling*, 4(4):386–398, 2005.

[53] M. Gogolla, F. Büttner, and D.-H. Dang. From graph transformation to ocl using use. In A. Schürr, M. Nagl, and A. Zündorf, editors, *AGTIVE*, volume 5088 of *Lecture Notes in Computer Science*, pages 585–586. Springer, 2007.

[54] M. Gogolla and M. Richters. Expressing UML class diagrams properties with OCL. In A. Clark and J. Warmer, editors, *Object Modeling with the OCL, The Rationale behind the Object Constraint Language*, pages 85–114, London, UK, 2002. Springer-Verlag.

[55] Graphviz. Graph visualization software. `http://www.graphviz.org`.

[56] E. Guerra and J. de Lara. Event-driven grammars: Relating abstract and concrete levels of visual languages. *Soft. and Syst. Mod., special section on ICGT'04*, pages 317–347, 2007.

[57] R. Heckel, J. M. Küster, and G. Taentzer. Confluence of typed attributed graph transformation systems. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *ICGT*, volume 2505 of *Lecture Notes in Computer Science*, pages 161–176. Springer, 2002.

[58] C. Herrmann, H. Krahn, B. Rumpe, M. Schindler, and S. Völkel. An algebraic view on the semantics of model composition. In D. H. Akehurst, R. Vogel, and R. F. Paige, editors, *ECMDA-FA*, volume 4530 of *Lecture Notes in Computer Science*, pages 99–113. Springer, 2007.

[59] ILOG CP. `http://www-01.ibm.com/software/integration/optimization/cplex-cp-optimizer/`.

[60] J. L. C. Izquierdo and J. G. Molina. A domain specific language for extracting models in software modernization. In Paige et al. [87], pages 82–97.

[61] D. Jackson. *Software Abstraction: Logic, Language and Analysis*. MIT University Press, Cambridge (MA), USA, 2006.

[62] JaCoP. `http://jacop.osolpro.com/`.

[63] C. Jones, P. O'Hearn, and T. J. Woodcock. Verified software: A grand challenge. *IEEE Computer*, 39(4):93–95, 2006.

[64] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. Atl: A model transformation tool. *Sci. Comput. Program.*, 72(1-2):31–39, 2008.

[65] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez. ATL: a QVT-like transformation language. In *OOPSLA Companion*, pages 719–720. ACM, 2006.

[66] F. Jouault and J. Bézivin. Km3: A dsl for metamodel specification. In R. Gorrieri and H. Wehrheim, editors, *FMOODS*, volume 4037 of *Lecture Notes in Computer Science*, pages 171–185. Springer, 2006.

[67] W. Kling, F. Jouault, D. Wagelaar, M. Brambilla, and J. Cabot. MoScript: A DSL for querying and manipulating model repositories. In *Software Language Engineering (SLE2011)*, Braga, Portugal, Oct. 2011.

[68] D. S. Kolovos, R. F. Paige, and F. Polack. Merging models with the epsilon merging language (eml). In Nierstrasz et al. [79], pages 215–229.

[69] A. Königs and A. Schürr. Tool integration with triple graph grammars - a survey. *ENTCS*, 148(1):113–150, 2006.

[70] M. Kyas, H. Fecher, F. S. de Boer, J. Jacob, J. Hooman, M. van der Zwaag, T. Arons, and H. Kugler. Formalizing UML models and OCL constraints in PVS. *Electron. Notes Theor. Comput. Sci.*, 115:39–47, January 2005.

[71] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*. Prentice Hall, 2004. .

[72] L. Lengyel, T. Levendovszky, and H. Charaf. Constraint validation in model compilers. *Journal of Object Technology*, 5(4):107–127, 2006.

[73] F. J. Lucas, F. Molina, and A. Toval. A systematic review of UML model consistency management. *Information and Software Technology*, 51(12):1631 – 1645, 2009.

[74] D. Lucrédio, R. P. de Mattos Fortes, and J. Whittle. Moogle: A model search engine. In K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, and M. Völter, editors, *MoDELS*, volume 5301 of *Lecture Notes in Computer Science*, pages 296–310. Springer, 2008.

[75] H. Malgouyres and G. Motet. A UML model consistency verification approach based on meta-modeling formalization. In *Proc. of the 2006 ACM symposium on Applied Computing (SAC'2006)*, pages 1804–1809, New York, NY, USA, 2006. ACM Press.

[76] A. Maraee and M. Balaban. Efficient reasoning about finite satisfiability of UML class diagrams with constrained generalization sets. In *Proc. of the 3rd Model Driven Architecture- Foundations and Applications (ECMDA-FA'2007)*, volume 4530 of *Lecture Notes in Computer Science*, pages 17–31. Springer-Verlag, 2007.

[77] K. Marriott and P. J. Stuckey. *Programming with Constraints: An Introduction*. MIT Press, Cambridge, Massachussetts, 1998.

[78] P. Mohagheghi, V. Dehlen, and T. Neple. Definitions and approaches to model quality in model-based software development - a review of literature. *Information and Software Technology*, 51(12):1646 – 1669, 2009. Quality of UML Models.

[79] O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors. *Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006, Proceedings*, volume 4199 of *Lecture Notes in Computer Science*. Springer, 2006.

[80] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT an SAT Modulo Theories: from an Abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, Nov. 2006.

[81] A. Nugroho and M. R. V. Chaudron. Evaluating the impact of uml modeling on software quality: An industrial case study. In A. Schürr and B. Selic, editors, *MoDELS*, volume 5795 of *Lecture Notes in Computer Science*, pages 181–195. Springer, 2009.

[82] Object Management Group. *MDA Guide V1.0.1*, 2003.

[83] Object Management Group. *UML 2.0 OCL Specification*, 2003.

[84] Object Management Group. *UML 2.0 Superstructure Specification*, 2004.

[85] OMG. MOF 2.0 Query/View/Transformation specification, 2007.

[86] Oz. `http://www.mozart-oz.org/`.

[87] R. F. Paige, A. Hartman, and A. Rensink, editors. *Model Driven Architecture - Foundations and Applications, 5th European Conference, ECMDA-FA 2009, Enschede, The Netherlands, June 23-26, 2009. Proceedings*, volume 5562 of *Lecture Notes in Computer Science*. Springer, 2009.

[88] C. Parent, K.-D. Schewe, V. C. Storey, and B. Thalheim, editors. *Conceptual Modeling - ER 2007, 26th International Conference on Conceptual Modeling, Auckland, New Zealand, November 5-9, 2007, Proceedings*, volume 4801 of *Lecture Notes in Computer Science*. Springer, 2007.

[89] E. Planas, J. Cabot, and C. Gómez. Verifying Action Semantics Specifications in UML Behavioral Models. In *CAiSE*, volume 5565 of *LNCS*, pages 125–140. Springer, 2009.

[90] R. Pottinger and P. A. Bernstein. Merging models based on given correspondences. In *VLDB*, pages 826–873, 2003.

[91] A. Queralt and E. Teniente. Reasoning on UML class diagrams with OCL constraints. In D. W. Embley, A. Olivé, and S. Ram, editors, *ER*, volume 4215 of *Lecture Notes in Computer Science*, pages 497–512. Springer-Verlag, 2006.

[92] A. Queralt and E. Teniente. Decidable reasoning in UML schemas with constraints. In Z. Bellahsene and M. Léonard, editors, *Proc. of the 20th Int. Conf. on Advanced Information Systems Engineering, (CAiSE'2008)*, volume 5074 of *Lecture Notes in Computer Science*, pages 281–295. Springer-Verlag, 2008.

[93] A. Queralt and E. Teniente. Reasoning on UML conceptual schemas with operations. In *Proc. of the 21st Int. Conf. on Advanced Information Systems Engineering (CAiSE'2009)*, volume 5565 of *Lecture Notes in Computer Science*, pages 47–62. Springer-Verlag, 2009.

[94] J. E. Rivera, E. Guerra, J. de Lara, and A. Vallecillo. Analyzing rule-based behavioral semantics of visual modeling languages with maude. In D. Gasevic, R. Lämmel, and E. V. Wyk, editors, *SLE*, volume 5452 of *Lecture Notes in Computer Science*, pages 54–73. Springer, 2008.

[95] G. Rozenberg. *Handbook of Graph Grammars and Computing by Graph Transformations*. World Scientific, 1987.

[96] A. Schürr. Specification of graph translators with triple graph grammars. In *WG'94*, volume 903 of *LNCS*, pages 151–163. Springer, 1994.

[97] M. Soeken, R. Wille, M. Kuhlmann, M. Gogolla, and R. Drechsler. Verifying uml/ocl models using boolean satisfiability. In *Design, Automation and Test in Europe (DATE'2010)*, pages 1341–1344. IEEE, 2010.

[98] M. Stölzel, S. Zschaler, and L. Geiger. Integrating ocl and model transformations in fujaba. *ECEASST*, 5, 2006.

[99] R. V. D. Straeten, T. Mens, J. Simmonds, and V. Jonckers. Using description logic to maintain consistency between UML models. volume 2863 of *LNCS*, pages 326–340. Springer, 2003.

[100] The ECL$^{\text{i}}$PS$^{\text{e}}$ Constraint Programming System. `http://www.eclipse-clp.org`, mar 2007. version 5.10.

[101] M. Tisi, J. Cabot, and F. Jouault. Improving higher-order transformations support in atl. In L. Tratt and M. Gogolla, editors, *ICMT*, volume 6142 of *Lecture Notes in Computer Science*, pages 215–229. Springer, 2010.

[102] M. Tisi, F. Jouault, P. Fraternali, S. Ceri, and J. Bézivin. On the use of higher-order model transformations. In Paige et al. [87], pages 18–33.

[103] M. Tisi, S. M. Perez, F. Jouault, and J. Cabot. Lazy execution of model-to-model transformations. In J. Whittle, T. Clark, and T. Kühne, editors, *MoDELS*, volume 6981 of *Lecture Notes in Computer Science*, pages 32–46. Springer, 2011.

[104] E. Torlak, F. S.-H. Chang, and D. Jackson. Finding minimal unsatisfiable cores of declarative specifications. In *Formal Methods 2008, (FM'2008)*, volume 5014 of *Lecture Notes in Computer Science*, pages 326–341. Springer-Verlag, 2008.

[105] E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.

[106] UMLtoCSP. A tool for the formal verification of UML/OCL models based on Constraint Programming. `http://gres.uoc.edu/UMLtoCSP`.

[107] R. van der Straeten, T. Mens, J. Simmonds, and V. Jonckers. Using description logic to maintain consistency between UML models. In *Proceedings of 6th International Conference UML 2003 - The Unified Modeling Language*, pages 326–340, Oct. 2003.

[108] D. Varró. Automated formal verification of visual modeling languages by model checking. *Software and System Modeling*, 3(2):85–113, 2004.

[109] M. Wahler, D. Basin, A. D. Brucker, and J. Koehler. Efficient analysis of pattern-based constraint specifications. *Software and Systems Modeling*, 2010. To appear.

[110] S. Weißleder and B.-H. Schlingloff. Quality of automatically generated test cases based on ocl expressions. In *International Conference on Software Testing, Verification, and Validation (ICST'2008)*, pages 517–520. IEEE Computer Society, 2008.

[111] R. Wieringa. A survey of structured and object-oriented software specification methods and techniques. *ACM Comput. Surv.*, 30(4):459–527, 1998.

[112] R. Wieringa. A survey of structured and object-oriented software specification methods and techniques. *ACM Comput. Surv.*, 30(4):459–527, 1998.

[113] F. Yu, T. Bultan, and E. Peterson. Automated size analysis for OCL. In *Proc. of the 6th Joint Meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC-FSE '07)*, pages 331–340. ACM, 2007.