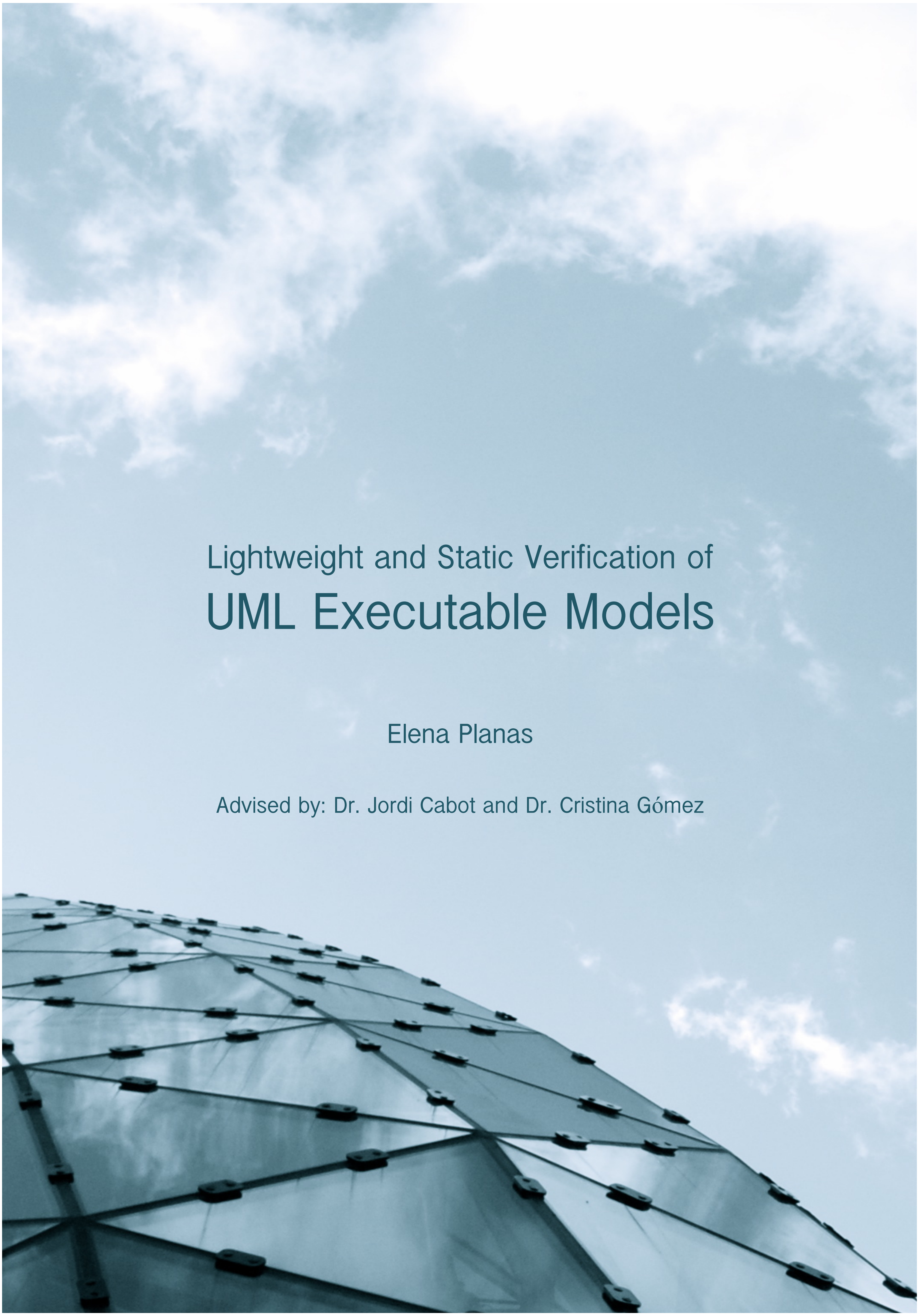


Lightweight and Static Verification of UML Executable Models

Elena Planas

Advised by: Dr. Jordi Cabot and Dr. Cristina Gómez

DOCTORAL STUDIES

Doctoral Thesis
– February 10, 2013 –

LIGHTWEIGHT AND STATIC VERIFICATION OF UML EXECUTABLE MODELS

Student: *Elena Planas Hortal*
Advisors: *Dr. Jordi Cabot Sagrera*
Dr. Cristina Gómez Seoane

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Elena Planas Hortal

Estudis d'Informàtica, Multimèdia i Telecomunicació
Rambla del Poblenou, 156
08018 - Barcelona (Spain)

eplanash@uoc.edu

A la meva família,
per acompanyar-me en aquest camí.

A dissertation presented by Elena Planas Hortal in partial fulfillment of the requirements for the degree of *Doctor per la Universitat Politècnica de Catalunya*

Barcelona, February 10, 2013

Agraïments

Doing a PhD thesis is a lonely task that
cannot be carried out alone.
Anonym

Aquesta tesi és el resultat d'uns quants anys de feina, durant els quals he rebut el suport, tan professional com personal, de moltes persones. Aquests agraïments són per tots vosaltres.

En primer lloc, vull donar les gràcies als meus directors de tesi, al Dr. Jordi Cabot i a la Dra. Cristina Gómez, per donar-me l'oportunitat de treballar i aprendre al seu costat. Al Jordi, moltes gràcies per aportar una visió clara i resolutiva dels problemes, pels seus consells precisos que sempre m'han resultat molt útils i per la seva disponibilitat des de qualsevol lloc del món. A la Cristina, moltes gràcies pel seu rigor científic i grau d'exigència, i també per la seva proximitat, dedicació i interès constant. A tots dos, moltes gràcies per la vostra orientació i suport que m'han permès arribar fins aquí.

Moltíssimes gràcies a totes les persones que han col.laborat directament amb el desenvolupament d'aquesta tesi. Un agraïment molt especial al David Sánchez, per haver-me ajudat amb la implementació de la tesi, però sobretot pel seu entusiasme en tot moment. També a l'Esther Guerra i al Juan de Lara, per permetre'm ampliar la visió del meu propi treball aplicant-lo a d'altres camps.

També vull donar les gràcies als examinadors d'aquesta tesi: el Dr. Antoni Olivé i a la Dra. Ruth Raventós, de la Universitat Politècnica de Catalunya (UPC), al Dr. Robert Clarisó, de la Universitat Oberta de Catalunya (UOC), al Dr. Vicente Pelechano, de la Universitat Politècnica de València (UPV) i al Dr. Manuel Wimmer, de la Vienna University of Technology (TUWien) per acceptar formar part del tribunal. També, als revisors externs que van llegir una versió prèvia d'aquesta tesi i em van fer arribar els seus suggeriments per millorar-la.

A tots els meus companys del Grup de Modelització Conceptual (GMC): Gràcies pels vostres comentaris i consells al llarg del desenvolupament d'aquesta tesi. Un agraïment especial a l'Antonio Villegas i al David Aguilera, per proporcionar-me la plantilla d'aquesta tesi i resoldre els meus dubtes de LaTeX amb una rapidesa increïble.

A tots els companys i amics dels estudis d'Informàtica, Multimèdia i Telecomunicació de la UOC. Al Josep Prieto, per donar-me totes les facilitats per poder acabar aquesta tesi. Al Santi Caballé i a l'Elena Rodríguez, per haver-se fet càrrec de les meves assignatures mentre jo he estat redactant aquest document. Als meus companys de despatx, el David Bañeres i el Jordi Conesa, pels moments compartits al llarg de tots aquests anys. A l'Àgata Lapedriza, l'Àngels Rius, i tants d'altres, per formar part del meu dia dia.

A very special acknowledgment (in english) to all my dutch collegues from the Software Engineering and Technology group (SET) at the Eindhoven University of Technology (TU/e).

Thanks specially to Mark van den Brand for allowing me to take part of his group and for the useful discussions we had during my stay in Eindhoven. Also many thanks to the rest of the people of the group: Ulyana Tikhonova, Yanja Dajsuren, Luna Luo, Luc Engelen, Bogdan Vasilescu, Tom Verhoeff, Anton Wijs, John Businge, Maarten Manders, Arjan van der Meer and Ruurd Kuiper for sharing her time with me. Dank u wel!

També vull donar les gràcies a tots els amics i amigues que, fora del meu entorn laboral, m'han ajudat a desconnectar de la feina i, de vegades sense saber-ho, m'han injectat la motivació necessària per seguir endavant. A la Déborah, la M.Àngels, la Raquel i el Joan, gràcies per estar sempre a prop malgrat la distància. A les meves amigues de dansa, la Núria, la Marta, la Laia, la Raquel l'Ariadna, la Ópal i la Clara, per la seva amistat més enllà de la dansa. I a les amigues tribaleres, la Marta C., la Sílvia, la Cristina, la Marga i la Marta A., per fer dels divendres un dels millors dies de la setmana. I també als amics de Campdevànol, la Laura, en Gil, l'Anna, el Páez i l'Urbon, per les nostres trobades i viatges sempre memorables.

Vull agrair a la meva família la seva estimació i suport constant al llarg de tota la vida. Molt especialment als meus pares, Imma i Joan, pel seu interès en aquesta tesi, malgrat encara no els hi hagi sabut explicar bé de què va. A la meva germana, Esther, per ser tan important a la meva vida. A la meva àvia, pel seu afecte per damunt de tot. I un record molt especial pel meu avi, que encara que no hagi pogut veure el final d'aquesta tesi, estic segura que n'estaria molt orgullós.

Finalment, i molt important per mi, moltes, moltíssimes gràcies a l'Hug per compartir la vida amb mi. Gràcies per la teva paciència i comprensió en els moments difícils, per recolzar-me i animar-me en tots els projectes però, sobretot, pel teu amor incondicional. I moltes gràcies també a la seva família (especialment a la Isabel), que també és la meva família.

A totes aquelles persones que d'una manera o altra m'han ajudat, encara que el seu nom no figuri de forma explícita en aquestes línies; sense elles tampoc hauria estat possible.

This work has been partly supported by the
Ministerio de Ciencia y Tecnología under
TIN2008-00444 project, *Grupo Consolidado.*

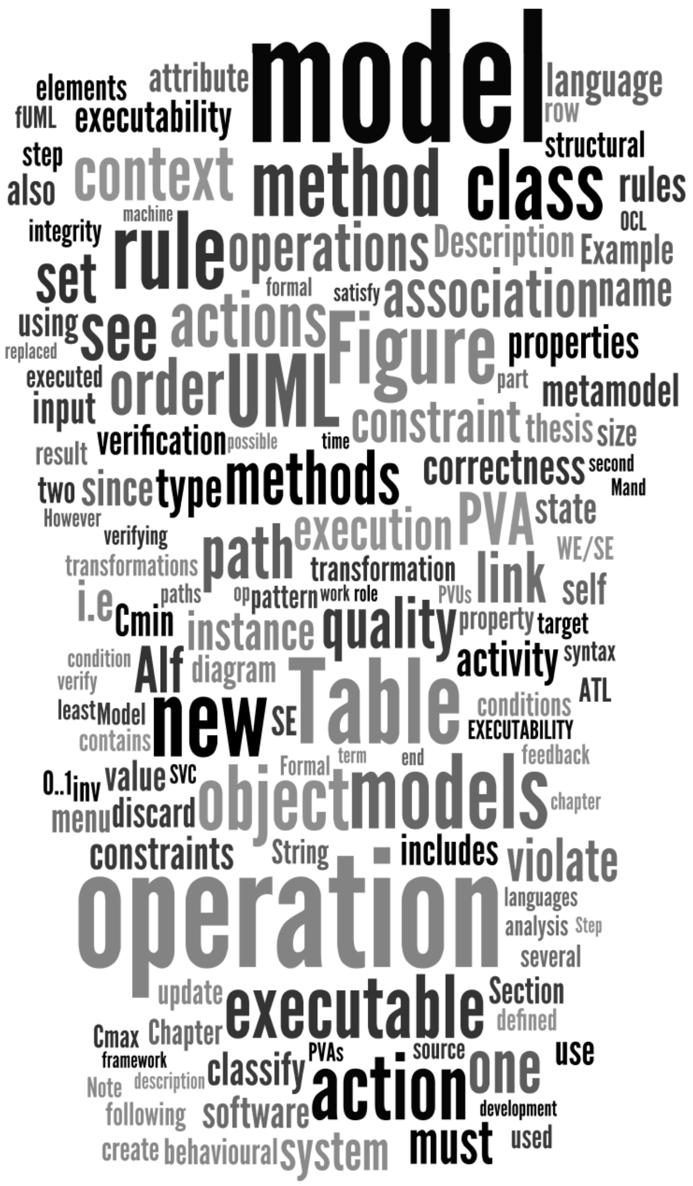
Abstract

Executable models play a key role in many development methods (such as MDD and MDA) by facilitating the immediate simulation/implementation of the software system under development. This is possible because executable models include a fine-grained specification of the system behaviour using an action language. Executable models are not a new concept but are now experiencing a comeback. As a relevant example, the OMG has recently published the first version of the "Foundational Subset for Executable UML Models" (fUML) standard, an executable subset of the UML that can be used to define, in an operational style, the structural and behavioural semantics of systems. The OMG has also published a beta version of the "Action Language for fUML" (Alf) standard, a concrete syntax conforming to the fUML abstract syntax, that provides the constructs and textual notation to specify the fine-grained behaviour of systems. The OMG support to executable models is substantially raising the interest of software companies for this topic.

Given the increasing importance of executable models and the impact of their correctness on the final quality of software systems derived from them, the existence of methods to verify the correctness of such models is becoming crucial. Otherwise, the quality of the executable models (and in turn the quality of the final system generated from them) will be compromised.

Despite the number of research works targetting the verification of software models, their computational cost and poor feedback makes them difficult to integrate in current software development processes. Therefore, there is the need for suitable and useful methods to check the correctness of executable models and tools integrated to the modelling tools used by designers.

In this thesis we propose a *verification framework* to help the designers improve the quality of their executable models. Our framework is composed of a set of *lightweight and static methods*, i.e. methods that do not require executing the model in order to check the desired property. These methods are able to check several properties over the behavioural part of an executable model (for instance, over the set of *operations* that compose a behavioural executable model) such as *syntactic correctness* (i.e. all the operations in the behavioural model conform to the syntax of the language in which it is described), *executability* (i.e. after the execution of an operation, the reached system state *is* -in case of *strong executability*- or *may be* -in case of *weak executability*- consistent with the structural model and its integrity constraints) and *completeness* (i.e. all possible changes on the system state can be performed through the execution of the operations defined in the executable model). For incorrect models, the methods that compose our verification framework return a meaningful feedback that helps repairing the detected inconsistencies.


model
elements
attribute
language
fUML
executability
row
structural
step
context
method
class
rules
also
machine
OCL
integrity
rule
operations
Description
Example
set
formal
satisfy
association
name
using
see
actions
Figure
replaced
properties
executed
order
UML
part
metamodel
input
constraint
thesis
size
result
verification
possible
time
correctness
second
two
since
type
methods
Mand
However
PVA
state
verifying
execution
path
WE/SE
transformations
transformation
link
self
i.e
paths
op
pattern
work
role
PVUs
Cmin
instance
quality
property
target
condition
diagram
activity
syntax
verify
Alf
Table
ATL
least
Model
new
SE
conditions
EXECUTABILITY
contains
Formal
term
end
feedback
0..1
inv
value
svc
object
models
chapter
menu
discard
constraints
String
includes
violate
operation
languages
analysis
Step
several
update
Section
executable
defined
Cmax
Chapter
source
use
framework
PVAs
classify
one
description
Note
action
development
following
software
must
create
behavioural
system
used

Contents

Part I

Preface

The White Rabbit put on his spectacles. "Where shall I begin, please your Majesty" he asked. "Begin at the beginning," the King said gravely, "and go on till you come to the end; then stop."

Lewis Carroll

1 Introduction

1.1 Software development

"The entire history of software engineering is that of the rise in levels of abstraction" said Grady Booch in his talk "The Promise, the Limits, the Beauty of Software" [21]. This history has a direct relationship with the history of computer programming [183], where many different programming languages have been developed in order to increase the levels of abstraction (see Figure 1.1).

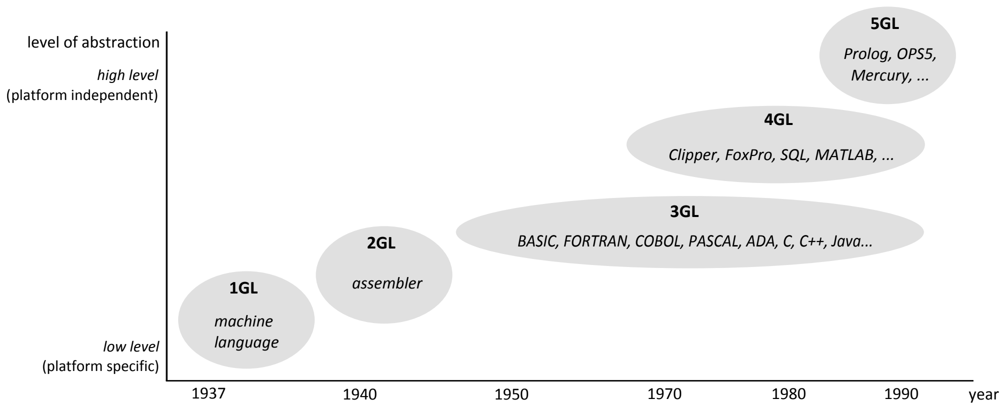


Figure 1.1. Rise in levels of abstraction in programming languages.

Programmers of the first computers had to use machine language (a binary language consisting of 0s and 1s to represent the status of a switch). Machine language, which is referred to as a first generation programming language (1GL), could be used to communicate directly with the computer. However, using machine language was difficult, error-prone and the programs were platform specific. Then, 1GL programming was quickly superseded by similarly machine specific, but mnemonic, second generation languages (2GL) known as assembly languages or assembler. Assembly languages are similar to machine languages, but they are much easier to program in because they allow using meaningful names or abbreviations instead of numbers. Although assembly languages were easier to use for humans than machine language, the programs were still platform specific. Later in the 1950s, assembler, which had evolved to include the use of macro instructions, was followed by the development of the third generation languages (3GL), referring to high-level languages, such as BASIC, FORTRAN, COBOL, PASCAL, ADA, C, C++, Java, etc. High-level programming languages have English-like instructions and are easier to use than machine language and assembler.

Lying above high-level languages are languages called fourth generation languages (4GL), which are programming languages (closest to human languages) or programming environments designed with a specific purpose in mind, such as the development of commercial business software. Some fourth generation languages are Clipper, FoxPro, SQL, MATLAB, etc. The 4GL was followed by efforts to define and use a fifth generation languages (5GL), which are programming languages based around solving problems using constraints given to the program, rather than using an algorithm written by a programmer. Most constraint based and logic programming languages and some declarative languages are fifth generation languages. Fifth generation languages are used mainly in artificial intelligence research and neural networks. Some fifth generation languages are Prolog, OPS5, and Mercury. 4GL and 5GL projects are more oriented toward problem solving and systems engineering.

Today, instead to evolve to more abstract programming languages, there is a growing trend toward using models rather (or before than) code. Models are abstract representations of the code. They are the next logical, and perhaps inevitable, evolutionary step in the ever-rising level of abstraction at which software engineers express software solutions. In fact the software engineering community envisages a future in which, rather than elaborate an analysis product into a design product and then write code, developers of the future will use tools to translate abstract application constructs into executable entities.

This shift is made possible by the confluence of two main factors:

- The Model-Driven Development (MDD) paradigm.
- The adoption of the Precise Action Semantics for the Unified Modeling Language (UML) specification by the Object Management Group (OMG).

In the next subsections we review these factors and discuss the implications of them relative to the future of software development.

1.1.1 Introducing Model-Driven Development

Model-Driven Development (MDD) [10, 162] is a software development paradigm that emphasizes the use of models as the primary artifact in all phases of the development life-cycle. MDD promotes the automatic generation of the system implementation based on its model, either directly or by first transforming the model into a new model adapted to the specific features and characteristics of the target platform.

MDD was practiced by different organizations using different architectures and tools. However, the Object Management Group (OMG) standardized this practice by publishing standard specifications for MDD and renamed this paradigm as **Model-Driven Architecture** (MDA). Then, MDA [110] is the OMGs particular vision of MDD and thus relies on the use of OMG standards (such as Meta-Object Facility (MOF), XML Metadata Interchange (XMI), Unified Modeling Language (UML), Common Warehouse Metamodel (CWM), among others). In the MDA paradigm there are two kinds of models: PIM (Platform-Independent Model), that provides a formal specification of the structure and behaviour of the system by abstracting away technical details, and PSM (Platform Specific Model), that specifies the system in terms of the implementation constructs that are available in one specific implementation technology. MDA essentially calls for the mapping of a PIM onto a PSM.

The MDA paradigm provides an important benefit: the intellectual capital invested in the PIM is forever protected from changes or advances in the underlying technologies. The PIM lets you model a solution both visually and at the higher level of abstraction. Then, MDA avoids the necessity of re-writing applications to take advantage of newer technology. Assuming that there will always be a next greatest technology, the profit of using MDA is very significant.

Both MDD and MDA can be regarded as a specific practices of **Model-Driven Engineering** (MDE), because MDE goes beyond of the pure development activities and encompasses other model-based tasks of a complete software engineering process (e.g. the Model-Driven Reverse Engineering of a legacy system). All the variants of Model-Driven *Whatever* are often referred to with the acronym MD* (Model-Driven star) [24] (see Figure 1.2).

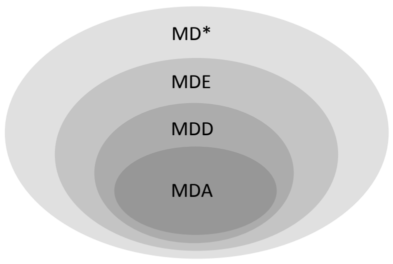


Figure 1.2. Relationship between the different MD* acronyms.

1.1.2 Adopting Action Semantics for UML

The usefulness of MDD and MDA paradigms largely depends on the possibility to define executable models (i.e. models which can be run). Executable models are detailed enough in order to be used to (semi)automatically implement the software system. For this purpose, in 2001 the OMG introduced the Precise Action Semantics for UML. Since then, actions have evolved to the latest standards fUML [125] (a precise semantics for actions) and Alf [124] (a precise text-based action language). These standards provide an unambiguous set of actions for specifying the behavioural aspects of a UML model to a level of detail such that a self-contained and completely executable application can be generated from that model. For instance, these standards include actions to support the synchronous manipulation of objects (create, read, write, delete), the generation and handling of asynchronous events (signals) and the logical constructs that support the specification of algorithms (conditional and loop structures).

The breakthrough notions of using an action language to specify the behaviour in a UML model are that: (1) the action language allows designers to define behavioural specifications at a higher level of abstraction; and (2) the action language is independent of any specific underlying technology in the execution environment. This implies that actions allow only direct manipulation of elements at the model level. To accomplish that goal, designers give up low-level control (for instance, pointer manipulation, persistence, data storage, and so on). Instead, application configuration deals with such issues at code generation time. Because an action language (and indeed the entire PIM UML model) is by definition independent of the underlying execution technology, designers can target the application solution to multiple and diverse execution platforms, even future platforms that did not exist at the time the PIM was developed. Then, the adoption of an action language supports the viability of executable models.

1.2 Software quality

One of the pivotal issues during the software development, either using models or directly code, is the quality of these representations.

Quality is a complex and multifaced concept which is often defined by what is lacking rather than by its contents. Lack of quality leads to failures. For instance, in 2010, Toyota announced a recall of almost half million new hybrid cars. The cars, including Toyota's Prius line, had a software defect, which would cause a lag in the anti-lock-brake system. Class-action lawsuits resulting from those recalls, including the software defect, were estimated to cost Toyota as much as $3 billion. More recently, in 2012, Nokia discovered that its Lumia 900 smartphone had been introduced with a software bug that could prevent users connecting to the Internet. After less than fifteen days a software update to fix the problem was released. However, the cost of this issue was estimated in $10 million on likely sales in that period.

These are only two recent examples of history's software bugs. They point out that human errors are unavoidable. However, in order to avoid that errors become disasters, it is highly important (specially in concurrent and critical systems) to check the system quality before its release.

The meaning of *quality* has been widely discussed. Everybody agrees that quality is an important property of products, but what does *quality* really mean? In 1984 Garvin [68] identified five major approaches of defining the **quality of a product** arising from scholars in several disciplines (philosophy, economics, marketing, and operations management). They provided the following definitions for the quality of a product:

- **Transcendental approach:** Quality is an “innate excellence” that cannot be defined precisely, but it is intuitively and universally recognizable through experience.
- **User-based approach:** Quality is fitness for intended use.
- **Value-based approach:** Quality is defined in terms of *costs* and *prices*: A quality product is one that provides performance at an acceptable *price* or conformance at an acceptable *cost*.
- **Manufacturing-based approach:** Quality is the conformance to the product’s requirements.
- **Product-based approach:** Quality is a precise and measurable variable that reflects the presence or absence of an assessable and desired product properties.

Later, in 2000, the ISO/IEC 9126 [88] (an international standard for the evaluation of software quality consistent with ISO 9000 [87], a family of standards related to quality management) defined the **quality of a software** as: “The totality of features and characteristics of a product or service that bear on its ability to satisfy stated or implied needs”.

The above definition of software quality may be applied within the context of several dimensions of software. Unhelkar [174] classifies UML-based software quality into seven categories (from lower to higher level of abstraction):

- **Data quality:** Focused on the data, resulting in quality work ensuring integrity of the data.
- **Code quality:** Focused on the programs and their underlying algorithms.
- **Model quality:** Focused on software models and their meanings. It covers the quality of models, modelling languages and even transformations performed on models.
- **Architecture quality:** Focused on the ability of the system to be deployed in operation.
- **Process quality:** Focused on the activities, tasks, roles and deliverables employed in developing the software.
- **Management quality:** Focused on the plan, budget and monitoring, as well as the “soft” or human aspects of a project.
- **Quality environment:** Focused on all aspects of creating and maintaining the quality of a project, including all of the above aspects of quality.

Similarly, ISO/IEC 9126 [88] classifies software quality into four categories:

- **Process quality:** Quality of the software lifecycle processes.
- **Internal quality:** Quality of the intermediate products, including static and dynamic models, documentation and source code.
- **External quality:** Quality of the final system as assessed by its external behaviour.
- **Quality in use:** Effect of the system in use - the extent to which users can achieve their goals using the system.

Despite the above classifications into categories, there is a relevant connection between the different categories of quality. For instance, improving quality of models (internal quality) will help to improve quality of the final system (external quality).

Although several definitions and classifications of software quality have been proposed, currently there is no agreed definition about what does quality for these specific categories of software mean. The meaning of quality for all these specific categories should be consistent with ISO/IEC 9126 [88], as all the categories exist in the context of a software system. Besides, it should also be consistent with ISO 9000 [87], as a software system is simply a particular type of product (see Figure 1.3).

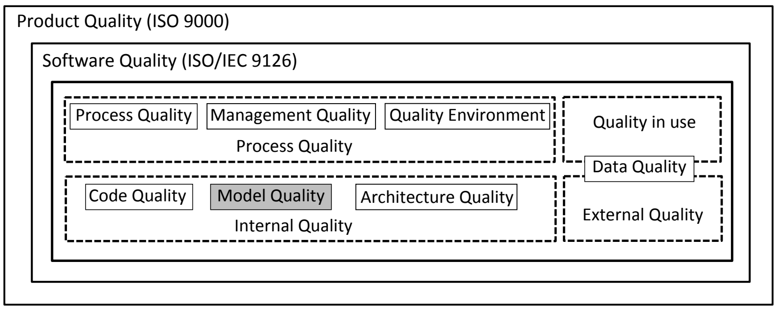


Figure 1.3. The context of model quality.

Since this thesis is focused on the development phases, in which only internal properties of the software can be measured, we focus on *internal quality* (in particular, on *model quality*) and we adopt (and adapt) the definition proposed in [113] as:

The **quality of a software model** is the degree to which a set of *internal properties* (also called *quality goals* in the literature) is present.

1.2.1 Software Models quality

Traditionally, the focus of software quality has been on evaluating the final product [115]. However, according to the Boehm's first law (1975) [19]: "Errors are most frequent during the requirements and design activities and are the more expensive the later they are removed" (see Figure 1.4).

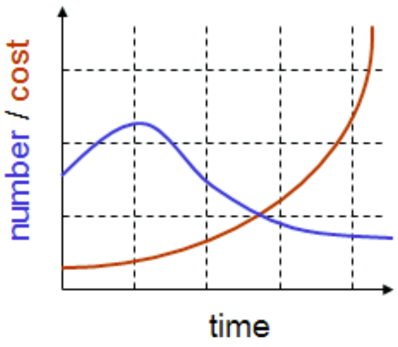


Figure 1.4. Boehm curve.

Most errors occur in early phases of the system development (i.e. during the requirements ellicitation and design phases). An error analysis made by Endres in 1975 [57] concluded that "about 60-70% of all errors to be found in a project are either requirements or design errors". Requirement errors are made when the developer does not know the domain. Nobody tells the developer that she is trying to solve the wrong problem until she interacts with the stakeholders again, which may be as late as deployment. On the other hand, design errors are found by the responsible developer, the developer of a related construct, or by a user who discovers the problem either accidentally or through a review process. In this thesis we focus on design errors over platform independent and executable models.

The cost of an error depends on when it is removed. Several studies conclude that the cost of repairing an error is the higher the longer it stays in the product. For instance, Fagan [61] determine that "the cost of rework in the later stages of a project can be greater than 100 times the cost of correction in the early stages". More recently, in 2001, a study conducted by Boehm and Basili [17] pointed out that "about 80% of avoidable rework seem to come from 20% of the defects".

As a consequence of Boehm's first law, the cost of the errors rises with their lifespan (i.e. the time between introduction of an error and its removal). Since errors are unavoidable, it is highly important to reduce its lifespan, i.e. to remove the errors as soon as possible after the time of commission.

In the context of MDD and MDA paradigms, where models are the basis of the whole development process, the quality of the models has a high impact on the final quality of software systems derived from them [121]. It means that a quality model (i.e. a model that fulfills several correctness properties) will lead to a higher quality information system. As a consequence, note that models may directly affect both the *efficiency* (i.e. time, cost and effort of development) and the *effectiveness* (i.e. quality of the information system derived from them) of information systems development.

1.3 Problem Statement

Based on the previous discussion we formulate the problem statement that is central to this thesis as:

Given the increasing importance of executable models and the impact of their correctness on the final quality of the software systems derived from them [121], the existence of methods to verify the correctness of executable models is becoming crucial. Otherwise, the quality of the executable models (and in turn the quality of the final system generated from them) will be compromised. Despite the number of research works targetting the verification of software models, their computational cost and poor feedback makes them difficult to integrate in current software development processes. Therefore, there is the need for suitable and useful methods to check the correctness of executable models and tools integrated to the modelling tools used by designers.

The list of open problems presented in [127] by Olivé includes the *Complete and Correct Executable Schemas*. Similarly, Genero et al. [69] suggest that more work is needed on model quality assessment. More precisely, Perseil [132] also points out that additional work remains to be done to verify Alf specifications. These overall goals are aligned with the purpose of this thesis.

1.4 Research Questions

To address the problem statement stated in Section 1.3, we formulate the main research question this thesis aims to answer as follows:

Main Research Question: *How can the quality of executable models be improved?*

In order to answer this research question, we decompose it into four specific research questions.

According to our definition of *model quality*, the first step in addressing the main research question is to define the internal quality properties that we take into account when analyzing the quality of executable models.

Research Question 1 (RQ_1): *How can the quality of executable models be decomposed into quality properties?*

Once quality has been decomposed into several quality properties, each property can be individually assessed. Several types of methods (static/dynamic and non-formal/formal) can be used for checking the quality of software models.

Research Question 2 (RQ_2): *What methods can be employed to support the verification of the quality properties of executable models?*

One important goal of our research is to provide information to help the designer improving her models before implementing them.

Research Question 3 (RQ_3): *What kind of feedback can help the designer to improve her executable models?*

The main objects of our research are executable models. However, executable models can encompass various types of models (i.e. model of a software system, model to model transformations, and so on).

Research Question 4 (RQ_4): *What kinds of executable models can be verified using these methods?*

1.5 Research Method

Along the development of this thesis we have followed the basic criteria of the *Design-Science Research* (DSR) paradigm.

Design-Science Research (DSR), proposed by Hevner et al. in [83], is a problem-solving paradigm that consists of activities concerned with the construction and evaluation of technology artifacts to meet organizational needs as well as the development of their associated research theories. Figure 1.5 presents the DSR paradigm for understanding, executing, and evaluating information systems research combining behavioural-science and design-science paradigms.

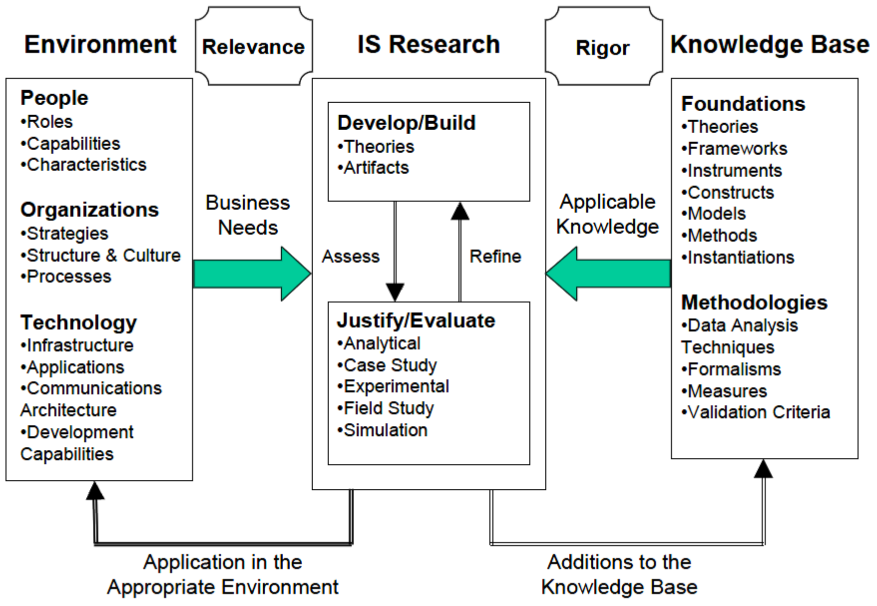


Figure 1.5. Design-Science Research paradigm.

Furthermore, Hevner et al. [83] propose a set of guidelines for conducting and evaluating DSR. They are summarized in Table 1.1.

Table 1.1. Design-Science Research Guidelines, adapted from [83].

Guideline	**Description**
Guideline 1: Design as an Artifact	DSR must produce a viable artifact in the form of a construct, a model, a method, or an instantiation.
Guideline 2: Problem Relevance	The objective of DSR is to develop technology-based solutions to important and relevant business problems.
Guideline 3: Design Evaluation	The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods.
Guideline 4: Research Contributions	Effective design-science research must provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies.
Guideline 5: Research Rigor	DSR relies upon the application of rigorous methods in both the construction and evaluation of the design artifact.
Guideline 6: Design as a Search Process	The search for an effective artifact requires the usage of available means to reach desired ends while satisfying laws in the problem environment.
Guideline 7: Communication of Research	DSR must be presented effectively both to technology-oriented as well as management-oriented audiences.

According to [83], the fundamental principle of design-science research is that knowledge and understanding of a design problem and its solution are acquired in the building and application of an artifact. That is, DSR requires the creation of an innovative, purposeful artifact (Guideline 1) for a specified problem domain (Guideline 2). Because the artifact is purposeful, it must yield utility for the specified problem. Hence, thorough evaluation of the artifact is crucial (Guideline 3). Novelty is similarly crucial since the artifact must be innovative, solving a heretofore unsolved problem or solving a known problem in a more effective or efficient manner (Guideline 4). In this way, design-science research is differentiated from the practice of design. The artifact itself must be rigorously defined, formally represented, coherent, and internally consistent (Guideline 5). The process by which it is created, and often the artifact itself, incorporates or enables a search process whereby a problem space is constructed and a mechanism posed or enacted to find an effective solution (Guideline 6). Finally, the results of the design-science research must be communicated effectively (Guideline 7) both to a technical audience (researchers who will extend them and practitioners who will implement them) and to a managerial audience (researchers who will study them in context and practitioners who will decide if they should be implemented within their organizations).

Additionally to the set of guidelines proposed in [83], some authors extend this paradigm by suggesting a methodology for conducting DSR in information systems. According to the methodology presented by Peffers et al. in [131], the DSR process includes six steps (see Figure 1.6): (1) problem identification, motivation and relevance, (2) definition of the objectives for a solution, (3) design and development of an artifact for solving the problem, (4) demonstration about the usefulness of the artifact, (5) evaluation of the solution to determine its effectiveness and efficiency, and (6) communication to the research community.

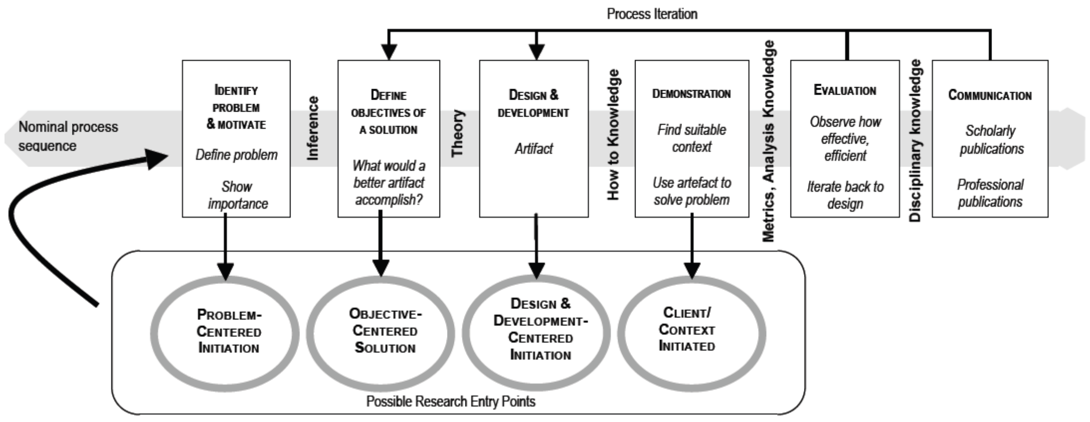

Process Iteration
Nominal process sequence
Identify problem & motivate
Define problem
Show importance
Inference
Define objectives of a solution
What would a better artifact accomplish?
Theory
Design & development
Artifact
How to Knowledge
Demonstration
Find suitable context
Use artefact to solve problem
Metrics, Analysis Knowledge
Evaluation
Observe how effective, efficient
Iterate back to design
Disciplinary knowledge
Communication
Scholarly publications
Professional publications
Problem-Centered Initiation
Objective-Centered Solution
Design & Development-Centered Initiation
Client/Context Initiated
Possible Research Entry Points

Figure 1.6. Design-Science Research Methodology.

1.6 Contributions of this thesis

In this thesis we propose a **verification framework to help the designers improve the internal quality of their executable models** (see Figure 1.7). Our framework is composed of a set of **static methods** that do not require to execute the model in order to check the desired property. Each of these methods assesses the verification of a specific correctness property, in particular:

- **Syntactic correctness**. An executable model is *syntactically correct* if all the elements in the model conform to the syntax of the language in which it is described.
- **Executability**. An entity of behaviour[1] of an executable model is *executable* if, after its execution, the reached system state *is* - in case of *strong executability* - or *may be* - in case of *weak executability* - consistent with the structural model and its integrity constraints.
- **Completeness**. The whole executable model is *complete* when all possible changes on the system state can be performed through the execution of the behaviour entities defined in the executable model. Otherwise, there will be parts of the system that users will not be able to modify since no available behaviour addresses their modification.

[1]In this introductory chapter, by *entity of behaviour* we mean a piece of behaviour which is part of a behavioural model such as a UML behavioural diagram, a MSM transformation, etc.

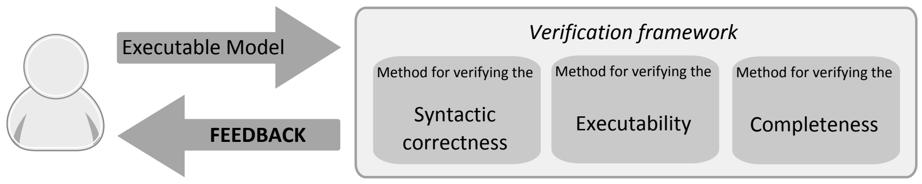


Figure 1.7. Verification framework overview.

We consider all the above properties (*syntactic correctness*, *executability* and *completeness*) are mandatory properties such that all correct executable models should satisfy.

The input of all the methods that compose our verification framework is an **executable model** (consisting of a UML action-based executable model - see Chapter 2 - or in a model to model (M2M) transformation - see Chapter 8 -). As a result, each particular method provides a meaningful **feedback** (showing *what* is wrong, *why*, and suggesting repairing procedures) to the designer.

1.6.1 Contributions in the context of Design-Science Research

The characteristics of our research have a clear relationship with the guidelines of DSR paradigm [83] and the DSR methodology [131] presented in Section 1.5. In the following we show how DSR has been applied to this thesis according to the guidelines of Table 1.1:

- *Design as an Artifact.* The artifacts obtained from our research are a set of static verification methods within the context of a verification framework and a tool which implements these methods.
- *Problem Relevance.* As we have introduced, given the increasing importance of executable models and their impact on the quality of the final software systems derived from them, the existence of methods to verify the correctness of such models is becoming crucial.
- *Design Evaluation.* The proposed framework has been evaluated through experimentation.
- *Research Contributions.* The contributions of our research are the artifacts themselves.
- *Research Rigor.* This research has been conducted by DSR methodology.
- *Design as a Search Process.* This thesis has been driven iteratively. In each iteration, we search a new refined solution to discover an effective solution to our problem.
- *Communication of Research.* We have published the main contributions of this thesis in research publications aimed to communicate the research results (see Section 1.6.2).

1.6.2 Publications

Most of the contributions of this thesis have been already published in the following publications (chronologically ordered):

International Conferences & Workshops:

- Elena Planas: *A Framework for Verifying UML Behavioural Models.* In: Proceedings of the 21st International Conference on Advanced Information Systems Engineering (CAiSE Doctoral Consortium 2009). Amsterdam, The Netherlands, June 2009 [135].
- Elena Planas, Jordi Cabot, Cristina Gómez: *Verifying Action Semantics Specifications in UML Behavioural Models.* In: Proceedings of the 21st International Conference on Advanced Information Systems Engineering (CAiSE 2009), volume 5565 of LNCS, pages 125-140. Amsterdam, The Netherlands, June 2009 [137].
- Elena Planas, Jordi Cabot, Cristina Gómez, Esther Guerra, Juan de Lara: *Lightweight Executability Analysis of Graph Transformation Rules.* In: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2010), pages 127-130. Leganés-Madrid, Spain, September 2010 [141].
- Elena Planas, Jordi Cabot, Cristina Gómez: *Two Basic Correctness Properties for ATL Transformations: Executability and Coverage.* In: Proceedings of the 3rd International Workshop on Model Transformation with ATL (MtATL 2011), volume 742 of CEUR Workshop proceedings, pages 1-9. Zurich, Switzerland, July 2011 [140].
- Elena Planas, Jordi Cabot, Cristina Gómez: *Lightweight Verification of Executable Models.* In: Proceedings of the 30th International Conference on Conceptual Modeling (ER 2011), volume 6998 of LNCS, pages 467-475. Brussels, Belgium, November 2011 [139].
- Elena Planas, David Sanchez-Mendoza, Jordi Cabot, Cristina Gómez: *Alf-verifier: An Eclipse Plugin for Verifying Alf/UML Executable Models.* In: Proceedings of the 31st International Conference on Conceptual Modeling (ER Workshops 2012), volume 7518 of LNCS, pages 378-382. Florence, Italy, October 2012 [143].

National Conferences:

- Elena Planas, Jordi Cabot, Cristina Gómez: *Verificación de la ejecutabilidad de operaciones definidas con Action Semantics.* In: Actas de los Talleres de las Jornadas de Ingeniería del Software y Bases de Datos (DSDM - JISBD 2008), Vol. 2, No. 3, pages 62-71. Gijón, Spain, October 2008 [136].

Research Reports:

- Elena Planas, Jordi Cabot, Cristina Gómez: *Verifying Action Semantics Specifications in UML Behavioural Models (Extended Version).* LSI-09-6-R LSI Research Report, UPC (2009) [138].

1.7 Thesis Outline

In this thesis we report the results of our research addressing the quality of executable models. This section gives an outline of the structure of the remaining of this document, which is structured in several parts:

Part II: Background: This part presents the context where this thesis is based on.

- **Chapter 2: Executable Models.** This chapter reviews the basic concepts about executable models and introduces the OMG standards to represent them (UML, fUML and Alf).
- **Chapter 3: Quality of Models.** This chapter reviews some well-known frameworks for evaluating the quality of models and classifies some of the main practices to analyze the correctness of a model.

Part III: Contributions: This part presents the contributions of this thesis.

- **Chapter 4: Verifying Executable Models.** This chapter roughly describes the verification framework where the methods provided in this thesis are organized on. The various methods are explained in detail in Chapters 5 to 8.
- **Chapter 5: Syntactic correctness.** This chapter provides a precise definition of the *syntactic correctness* property and presents a specific lighweight and static method to assess this property over a UML action-based behavioural model as part of an executable model.
- **Chapter 6: Executability.** This chapter provides a precise definition of the *executability*property and presents a specific lightweight static method to assess this property over a UML action-based behavioural model as part of an executable model.
- **Chapter 7: Completeness.** This chapter provides a precise definition of the *completeness* property and presents a specific lightweight static method to assess this property over a UML action-based behavioural model as part of an executable model.
- **Chapter 8: Application to M2M Transformations.** This chapter adapts the previous methods to be used in the context of model to model transformations.
- **Chapter 9: Tool Implementation.** This chapter presents the architecture of a prototype tool that implements the most relevant methods proposed as part of our verification framework.
- **Chapter 10: Experimentation.** This chapter presents an experimentation to evaluate the relevance and the efficiency of our verification framework.
- **Chapter 11: Related Work.** This chapter discusses the related work, including a comparison of our methods with respect to other existing methods in the literature.
- **Chapter 12: Conclusions.** This chapter exposes the conclusions of this thesis and points out further research lines.

Part II

Background

Models that don't execute are like cars without engines.

Stephen Mellor

2 Executable Models

It is widely accepted that "a *model* is a simplified representation of a complex reality, usually for the purpose of understanding that reality, and having all the features of that reality necessary for the current task or problem" [27]. We build models to improve our understanding of something complex. Architects build models of a proposed building to show a costumer what to expect; airplane manufacturers test models in a wind tunnel for flight attributes; and software developers build models of projects they undertake.

Software models can be used in many ways. Models can be used only informally, that is, sketching out a few diagrams to discuss the abstractions before coding. Models can also be used as a blueprint that specifies the software structure. This approach intends near one-to-one correspondence between the models and the code. Then, a tool can generate code based on these models, and the developer fills in the rest using the models as a guide. However, if the developer finds a better solution while coding, the model will no longer reflect the code.

But software models (from now on we call them simply "models") can do better than this. They can be described in sufficient detail to be directly executed. Executable models are neither sketches nor blueprints. As their name suggests, executable models can be executed and translated into code. In this sense, executable models act just like code, but they are described at a higher (and platform independent) level of abstraction.

The aim of this chapter is to provide an outline about executable models. It is divided into three sections: Section 2.1 defines what an executable model is and which improvements it can contribute to conceptual modelling; Section 2.2 describes how an executable model may be specified using the OMG standard languages devoted to executable modelling; and finally, Section 2.3 summarizes and concludes the chapter.

2.1 Overview

Broadly, a software development process based on models involve two main steps (see Figure 2.1): (1) the modellers create models and deliver them to the developers; and then (2) the developers create software artifacts (in the best case) based on these models and, in some cases, they provide feedback to the modellers. This lifecycle outlines several problems. For instance, developers could choose not to follow the models, or they could decide not to provide any feedback to the modellers. Besides, it is hard to keep the models and software artifacts synchronized during the development (and maintenance), and it is also hard to verify the correctness of models before development.

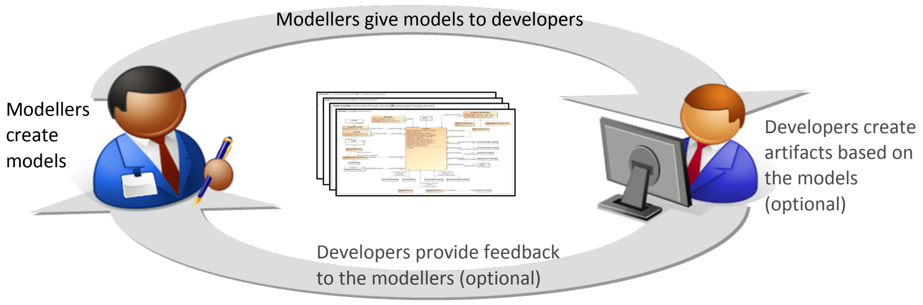

Modellers give models to developers
Modellers create models
Developers create artifacts based on the models (optional)
Developers provide feedback to the modellers (optional)

Figure 2.1. How a model-based development process usually works.

The adoption of the Model-Driven Development (MDD) paradigm [10, 162] can cope with the above problems. As we introduced in Chapter 1, the MDD paradigm gives the model a central role in the development process. It promotes the automatic generation of the system implementation based on its model, either directly or by first transforming the model into a new model adapted to the specific features and characteristics of the target platform.

However, the power of MDD cannot be fully exploited without the use of executable models, where "executable" means that it is possible to write an engine program (i.e. a model compiler) that executes (or runs) the model. According to Sage et al. [155], an executable model "is a dynamic model that can be used to analyze the properties of the architecture and it can also be used to carry out simulations".

Although the OMG (Object Management Group) has recently published several standards related to executable models (see them in Section 2.2.2), neither of these standards include a definition about what an executable model is. We contribute our own definition based on the definition provided by E. Seidewitz (one of the authors of the new OMG standards related to executable modelling) in [30]:

An **executable model** is a model with a behavioural specification detailed enough so that it can be systematically implemented or executed in the production environment.

Executable models are not a new concept. In 1992 D. Harel (the creator of the statecharts notation) predicted that in the future most developments will be based on an executable specification [77]. Few years later, in 2001, Stephen J. Mellor and Mark J. Balcer published the xUML (eXecutable UML [167]), a software development methodology which relies on a subset of the UML notations (class diagram, statechart diagram, action language, etc.) designed to precisely define the semantics of executable models.

In the recent years, executable models have experienced a comeback, becoming a relevant topic by the OMG. As a relevant example, this organization has recently published the first version of the “Foundational Subset for Executable UML Models” (fUML) standard [125] (see Section 2.2.2), an executable subset of the UML that can be used to define, in an operational style, the structural and behavioural semantics of systems. The OMG has also published a beta version of the “Action Language for fUML” (Alf) standard [124] (see Section 2.2.2), a concrete syntax conforming to the fUML abstract syntax, that provides the constructs and textual notation to specify the fine-grained behaviour of systems. The OMG support to executable models is also substantially raising the interest of software companies for this topic [54].

According to E. Seidewitz [161], the combination of executable models within the MDD paradigm, makes possible a new and improved software development process based on two main steps (see Figure 2.2): (1) the modellers create executable models and iteratively verify and update them; and once the models are correct (2) the executable models are deployed in a production environment. There are several alternative strategies to deploy executable models. One way is through *code-generation* [82], i.e. using a model compiler (many times defined as a model-to-text transformation) to generate a lower-level representation of the model using existing programming languages and platforms (e.g. Java). Another way is through *model interpretation* [151], i.e. using a virtual machine able to directly read and run the model.

In this thesis, we only focus on the first step of the development process, in particular, on ensuring the correctness of executable models at design time. Whatever is the strategy to deploy the executable model (code-generation or model interpretation), such model should be correct before its deployment.

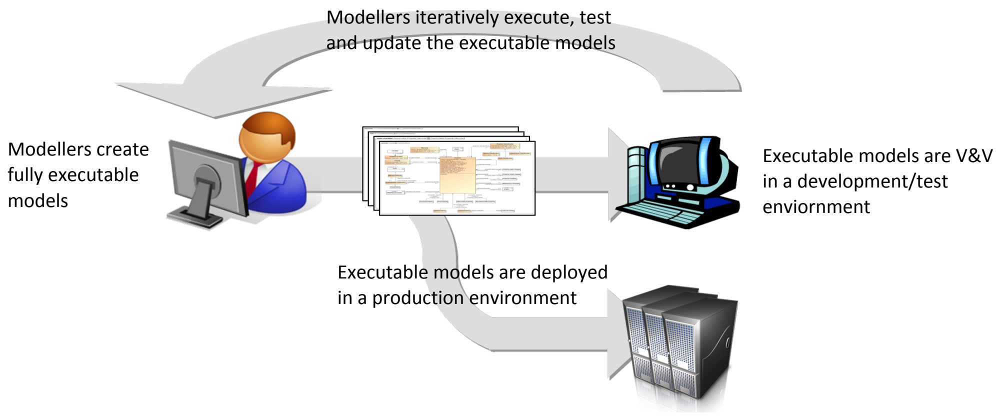

Modellers iteratively execute, test and update the executable models
Modellers create fully executable models
Executable models are V&V in a development/test enviornment
Executable models are deployed in a production environment

Figure 2.2. How a MDD-based development process using executable models usually works.

2.1.1 Why do we need executable models?

The use of executable models is being promoted given the value they can contribute:

- Executable models **increase productivity** by raising the level of abstraction. In [25], Brooks concludes that “the number of bugs per line of code is constant regardless of what level of abstraction you are working at”. Designing at higher levels of abstraction requires less code for the same functionality. Therefore there are fewer bugs per function when coding at a higher level of abstraction, resulting in higher productivity.

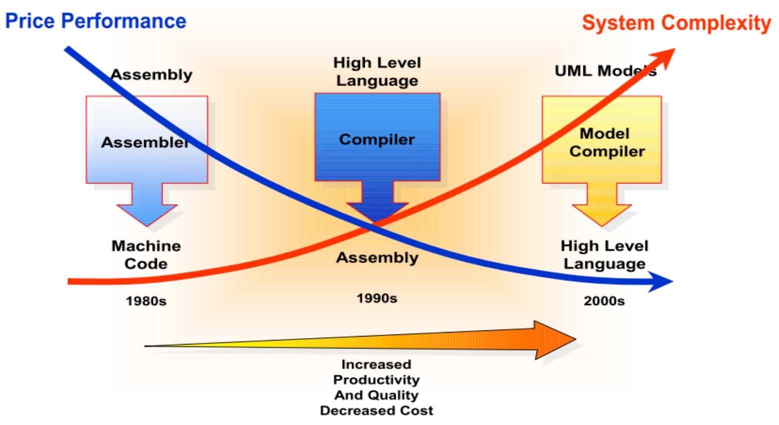


Figure 2.3. Abstraction increases productivity.

If we look at the languages used to design software (see Figure 2.3), the improvement in productivity was dramatic when the move was made from the assembly language to the first high-level languages. Nowadays, moving from high-level languages to executable modelling languages represents a similar advance. Executable models are at the next higher layer of abstraction, abstracting away both specific programming languages and decisions about the organization of the software.

- Executable models **reduce costs** by describing systems independently of their implementation and then compiling to each software platform. Executable models allow for the true separation of concerns. This significantly increases the ease of reuse and lowers the cost of software development. This also enables executable models to be cross-platform and not tied to any specific programming language, platform or technology, so an executable model can be deployed in various software environments without change. In this way, models can survive for decades.

- Executable models **improve quality** of the final system by facilitating early verification. They allow the modeller to verify the design before any code is written. Then, defects may be found sooner by execution, avoiding to waste effort in constructing the wrong code.

2.2 Specifying executable models

A model must describe both the structure of the system it represents (*what it is?*) and its behaviour (*what it does?*). For being executable we must put special emphasis on the behaviour's description, so that it must be precise enough to be executed.

Since its standardization by the OMG in 1997, the UML (Unified Modeling Language) [154] has become a *de facto* standard to specify models. UML provides a unified graphical notation for the representation of various aspects of software systems.

In this document, unless otherwise indicated, we assume that executable models are written using the OMG standards. However, as we explain in Chapter 8, the ideas presented in this document could be adapted to models specified by means of other languages.

We represent an **executable model** (ExM) as a 2-tuple ⟨SM, BM⟩, where:

- SM is a structural model, and
- BM is a behavioural model.

Sections 2.2.1 and 2.2.2 review the concepts of structural and behavioural model respectively and how they can be specified using the standard languages provided by the OMG.

2.2.1 Structural Model

The *structural model* specifies the static part of an information system, i.e. the general knowledge about the system domain [128].

We represent an **structural model** (SM) as a UML *class diagram*, that is, a 5-tuple ⟨*cl*, *attr*, *assoc*, *gen*, *ic*⟩, where:

- *cl* is a set of classes,
- *attr* is a set of attributes of each class,
- *assoc* is a set of associations among classes,
- *gen* is a set of generalizations among classes, and
- *ic* is a set of integrity constraints (i.e. conditions that must be satisfied in all states of an information system [128]).

All elements in the class diagram are assumed to be correct instances of the corresponding metaclasses of the UML metamodel [126].

Some integrity constraints (mainly cardinalities and disjointness/covering constraints in generalizations) may be graphically represented in the CD, while the rest of them may be textually specified in OCL [123].

In this thesis we take into account a subset of integrity constraints, in particular those that conform to one of the syntactic patterns shown in Table 2.1. According to [43] these constraints are the most commonly used integrity constraints in UML models. Table 2.1 is divided into several columns:

- *Constraint*: Identifies the constraint type.
- *Abbreviation*: Shows the abbreviated notation that we use throughout this document.
- *Description*: Briefly describes the meaning of each constraint.
- *Formalization*: Provides its formal description in OCL.

Table 2.1. Constraint types supported by our method.

Constraint	*Abbreviation*	*Description*	*Formalization in OCL*
Minimum cardinality of a class	Cmin(`cl`)	Expresses the minimum objects of class `cl` that must exist simultaneously.	**`context`** `cl` **`inv:`** `cl.allInstances() ->size() >= Cmin(cl)`
Maximum cardinality of a class	Cmax(`cl`)	Expresses the maximum objects of class `cl` that may exist simultaneously.	**`context`** `cl` **`inv:`** `cl.allInstances() ->size() <= Cmax(cl)`
Minimum cardinality of an association	Cmin(`as`,`r`)	Expresses the minimum multiplicity of the member end (i.e. role) `r` of an association `as` between `cl` (with role `r`) and `cl'`.	**`context`** `cl` **`inv:`** `cl.r->size() >= Cmin(as,r)`
Maximum cardinality of an association	Cmax(`as`,`r`)	Expresses the maximum multiplicity of the member end (i.e. role) `r` of an association `as` between `cl` (with role `r`) and `cl'`.	**`context`** `cl` **`inv:`** `cl.r()->size() <= Cmax(as,r)`
Mandatory attribute	Mand(`attr`, `cl`)	Expresses the attribute `attr` of class `cl` must have at least one value.	**`context`** `cl` **`inv:`** `not cl.attr->oclIsUndefined()`
Covering of a generalization	Cov(`cl`, {cl_1,..., cl_n}) (`cl` generalizes cl_1,..., cl_n)	Requires each instance of `cl` to be an instance of at least one cl_i.	**`context`** `cl` **`inv:`** `self.oclIsTypeOf(`cl_1`)` `OR ... OR` `self.oclIsTypeOf(`cl_n`)`

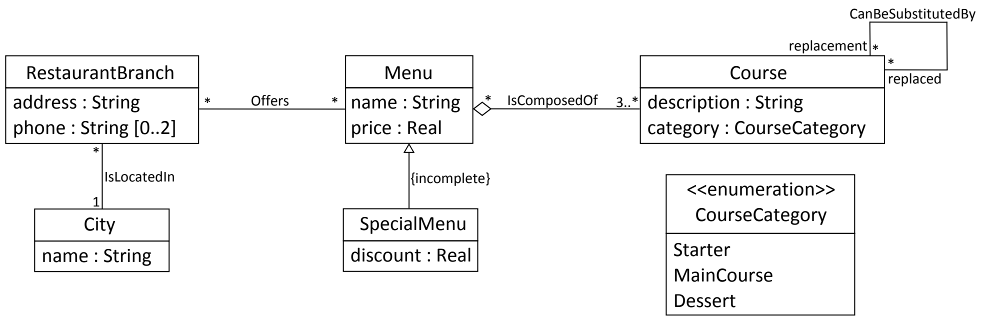
Continued on next page

Table 2.1 – continued from previous page

Constraint	*Abbreviation*	*Description*	*Formalization in OCL*
Disjointness of a generalization	Disj(`cl`, $\{cl_1,\ldots, cl_n\}$) (`cl` generalizes $cl_1,\ldots, cl_n$)	Requires each instance of `cl` to be instance of at most one cl_i.	**`context`** `cl` **`inv:`** `self.oclIsTypeOf(`cl_i`)` `implies not` `self.oclIsTypeOf(`cl_x`),` where x,i=1..n and $x \neq i$.
Identifier of an attribute	ID(`attr`,`cl`)	Expresses the attribute `attr` uniquely identifies instances of `cl`.	**`context`** `cl` **`inv:`** `cl.allInstances() -> isUnique(attr)`
Symmetry of a recursive association	Sym(`as`)	Guarantees if an object o_1 is `as`-related to o_2, then o_2 is `as`-related to o_1.	**`context`** `cl` **`inv:`** `self.r -> forAll(o|o.r -> includes(self)),` where `r` is a member end of `as`.
Asymmetry of a recursive association	Asym(`as`)	Guarantees that if an object o_1 is `as`-related to o_2, then o_2 is not `as`-related to o_1.	**`context`** `cl` **`inv:`** `self.r -> forAll(o|o.r -> excludes(self)),` where `r` is a member end of `as`.
Irreflexivity of a recursive association	Irrefl(`as`)	Guarantees that an object o is never `as`-related to itself.	**`context`** `cl` **`inv:`** `self.r->excludes(self),` where `r` is a member end of `as`.
Value comparison	ValueComp (`attr,op,v`)	States a restriction on the value of the attribute `attr`: the expression `attr op v` (where `op` is a comparison operator and `v` is a value) must be true.	**`context`** `cl` **`inv:`** `self.attr <op> v`
Referential integrity constraint	Referential(`cl`, `as`)	Guarantees each participant in the association `as` (in which `cl` participates) is an instance of its corresponding class.	**`context`** `cl` **`inv:`** `not self.r->oclIsUndefined()`

Example 1 As an example throughout this document we will use the class diagram shown in Figure 2.4, meant to (partially) model the menus offered by a restaurant chain. The class diagram contains information about the restaurant branches, the menus they offer (the price of each menu is the same in all branches) and the courses (at least three) that compose each menu. A course may have several

substituting courses, which are suggested to the customer when the desired course is sold out. Discounts for special menus can be offered.
In addition to the cardinalities included in the CD, on the bottom, a subset of its most representative constraints are expressed in OCL (left) and using our abbreviated representation (right). First constraint (`menuPrimaryKey`) states the name of a menu uniquely identifies it. Second constraint (`atMost3SpecialMenus`) states there may exist at most three special menus simultaneously. Third constraint (`validDiscount`) states the discount of a special menu must be at least 10 (per cent). Last constraint (`symmetricAssociation`) states the association `CanBeSubstitutedBy` is symmetric, i.e. if a course `c1` can be substituted by a course `c2` then `c2` can also be substituted by `c1`.



```
context Menu inv menuPrimaryKey: Menu.allInstances()->isUnique(name)                          | ID(name,Menu)
context SpecialMenu inv atMost3SpecialMenus: SpecialMenu.allInstances()->size()<=3            | Cmax(SpecialMenu)=3
context SpecialMenu inv validDiscount: self.discount >= 10                                    | ValueComp(self.discount,>=,10)
context Course inv symmetricAssociation: self.replaced -> forAll(c|c.replaced -> includes(self)) | Sym(CanBeSubstitutedBy)
```

Figure 2.4. Excerpt of a restaurant chain class diagram.

An information system maintains a representation of the system state in its information base. The *state* of the system at any given point in time is the set of instances of the classes and associations defined in the class diagram that exist in the domain at that time [128].

Example 2 Given the class diagram of Figure 2.4, a possible system state called *currentState* would be a state in which there is a restaurant branch with address "Sardenya Street, 457, Barcelona", which offers a non-special menu called "Anticrisis menu" for 5€.

A state s **satisfies** an integrity constraint ic iff ic evaluates to true in this state. We denote by *Satisfies(s,ic)* the proposition that represent the state s satisfies the integrity constraint ic. Otherwise, we say that the constraint is $unsatisfied$ or *violated*.

Let $ExM = \langle SM, BM \rangle$ be an executable model, a system state s is **consistent** regarding SM iff $\forall\ ic \in SM$, *Satisfies(s,ic)*. We denote by *IsConsistent(s,SM)* the proposition that represent the state s is consistent regarding the structural model SM.

Example 3 The state *currentState* (see Example 2) does not satisfy the minimum cardinality constraint of the association `IsComposedOf` in the role `course` (Cmin(`IsComposedOf`,`course`)=3), since the menu "Anticrisis menu" does not contain any course.

That is: *Satisfies(currentState,Cmin(*`IsComposedOf`*,* `course`*)=3) = false.*

Therefore, the previous state is not consistent with our the structural model ($RestaurantChain\_CD$) of Figure 2.4.

That is: *IsConsistent(currentState,RestaurantChain_CD) = false.*

2.2.2 Behavioural Model

The *behavioural model* specifies the dynamic part of an information system, i.e. the valid changes in the system state, as well as the functions that the system can perform [128].

In UML there are several alternatives to specify the behaviour of a system, for instance, using use case diagrams, activity diagrams, statechart diagrams, etc. The above diagrams usually work at a high level of abstraction. However, as we have introduced, in order to be executable, the behavioural models must be detailed enough. For this reason, we believe the best way to precisely define the behaviour of an executable model is using low-level actions. In this thesis, in order to define a detailed behavioural model, we use action-based operations. Operations (which are attached to UML classes) are sequences of atomic steps that users may execute to query and/or modify the information modelled in the structural model (SM).

In this document we assume that a *behavioural model* (BM) is composed of a set of action-based operations $\langle op_1, \ldots, op_n \rangle$.

We consider all operations of the behavioural model are executed in an atomic transaction. This means that the sequence of actions in the operations either all occur or nothing occurs. A guarantee of atomicity prevents updates to the information base occurring only partially, which can cause greater problems than rejecting the whole series outright.

In order to model behavioural executable models in general, and operations in particular, whilst the UML specification is necessary, it is not sufficient. This is because of two facts:

1. UML is not specified precisely enough to be executed. Although UML defines some execution semantics it is not expressive enough to describe each computable function. However, this is not fully clear because UML semantics is not as precisely defined as necessary to clarify this question.
2. Graphical modelling notations are not good for detailed specifications. Graphical notation tends to be very tedious for exhaustive specifications, confusing the specification rather than enhance it. Diagrams are preferred when the diagram is intuitive, but if the diagram is more verbose than a textual representation, then textual is preferred.

In order to overcome these issues the OMG has extended the UML standard to allow the models to be executable. In particular, two new standards has been recently added to the

UML standard: the "Foundational Subset for Executable UML Models" (fUML) [125] and the "Action Language for fUML" (Alf) [124]. In the following we introduce both standards and we exemplify its usage.

OMG standards for specifying executable models: fUML and Alf

The **Foundational Subset for Executable UML Models** (also known as "Foundational UML") (fUML) [125] is an executable and simplified subset of the standard UML that can be used to define, in an operational style, the structural and behavioural semantics of systems (see Figure 2.5).

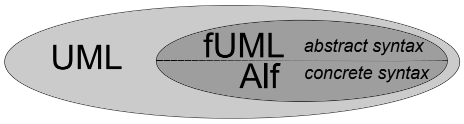


Figure 2.5. fUML overview.

In order to precisely specify the fine-grained behaviour, fUML includes the concept of *action*. An action is the fundamental unit of behaviour specification. It takes a set of inputs (*input pins*) and converts them into a set of outputs (*output pins*), where a pin is a typed and multiplicity element that provides values to actions and accepts result values from them. Some of the actions modify the state of the system in which the action is executed.

Actions were added first to the UML 1.4 in 2001 as the *Precise Action Semantics*. Since then, actions have evolved with the evolution of UML standardization (in fact, some features from UML have been excluded from the fUML to make this subset fully executable). However, nor UML 2.X neither fUML provide any concrete textual syntax for actions, but they only provide an abstract syntax (textually described) which is not really precise. Without an easy-to-use concrete and well-defined syntax, the creation of executable UML models remains a difficult task.

In order to cover this lack, the OMG proposed on October 2011 the **Action Language for Foundational UML** (Alf) [124], the first beta version of a concrete syntax conforming to the abstract syntax of the standard fUML. Essentially, Alf is an unambiguous, concise and readable textual language to specify executable models in the context of UML. Alf can be attached to any place that a UML behaviour can be. For instance, Alf sentences can be used directly to specify the behaviours of the transitions on a statechart diagram, the method of an operation or the classifier behaviour of an active class.

Further, Alf also provides an extended notation that may be used to specify structural modeling elements. Therefore, it is possible to specify a UML model entirely using Alf, though Alf syntax only directly covers the limited subset of UML structural modeling available in the fUML subset. However, in this thesis we use UML instead of Alf to represent the structural part of a model. This is because of: (1) we believe a graphical notion of the structural model is more intuitive to understand; and (2) the fUML subset (nor Alf) does not allow to define integrity constraints associated to the class diagram, an element that the methods presented in this thesis take into consideration.

But, why do we need Alf? We can wonder why we cannot use Java, C++ or another programming language. Programming languages are not designed to manipulate the elements of a model. They do not provide the facilities that we need to be able to express the actions in a model in a clear and precise, yet abstract, manner. However, programming languages allow the developer to manipulate all sorts of implementation-specific features that are wholly inappropriate in a PIM (Platform Independent Model). For instance, it is commonplace in modelling to want to navigate across an association (i.e. finding the associated object/s at the other end of an association). With a programming language we would need to know how the association is going to be implemented, for instance with pointers, and therefore navigate the association using pointers. This immediately makes the model implementation platform specific. However, Alf allows to navigate the association simply and concisely, without restricting the ways in which associations can be implemented.

Then, Alf is a platform independent language that works at the same semantic level as the rest of the UML model. This means that actions allow to directly manipulating the elements of the PIM (no assumptions are made about middleware, implementation language or software design policy) and they are capable of being translated into different implementations for different platforms and languages.

Syntactically, Alf is based on several key design principles:

- Alf has a largely C-legacy ("Java like") syntax, since that is most familiar to the community that programs detailed behaviours. Nevertheless, Alf allows UML textual syntax when it exists (e.g., colon syntax for typing, double colon syntax for name qualification, etc.).
- Alf provides a naming system that is based on UML namespaces for referencing elements outside of an activity but also provides for the consistent use of local names to reference flows of values within an activity.
- Alf does not require graphical models to change in order to accommodate use of the action language (e.g., special characters are allowed in names, arbitrary names are allowed for constructors, etc.). Further, while Alf maps to the fUML subset in order to provide its execution semantics, it is usable in the context of models not limited to the fUML subset.
- Alf uses an implicit type system that allows but does not require the explicit declaration of typing within an activity, always providing for static type checking, based at least on typing declared in the structural model elements.
- Alf has the expressivity of OCL in the use and manipulation of sequences of values. These sequence expressions are fully executable in terms of fUML expansion regions, allowing the simple and natural specification of highly concurrent computations.

Before the adoption of Alf, several action languages emerged (some of them are proprietary). As an example, some existing action languages precursors of Alf are: Object Action Language (OAL) [147], Shlaer-Mellor Action Language (SMALL) [111], Action Specification Language

(ASL) [96], That Action Language (plus an extra L) (TALL) [108], Starr's Concise Relational Action Language (SCRALL), Platform-independent Action Language (PAL) and PathMATE Action Language (PAL). Alf is also such a language, but one that is an OMG standard that can be consistently implemented across a number of tools, promoting the same sort of interoperability for textual behavioural specification that the UML standard already does for graphical modeling. So this is the reason why in this thesis we focus on Alf language. However, the ideas presented in this document would be adapted to any of the above action languages.

Even though the framework proposed in this thesis is fully-fUML/Alf compliant, in this document we focus on the write actions (actions that modify the system state) of Figure 2.6 since they are the ones that can compromise the correctness properties we present in the next chapters.

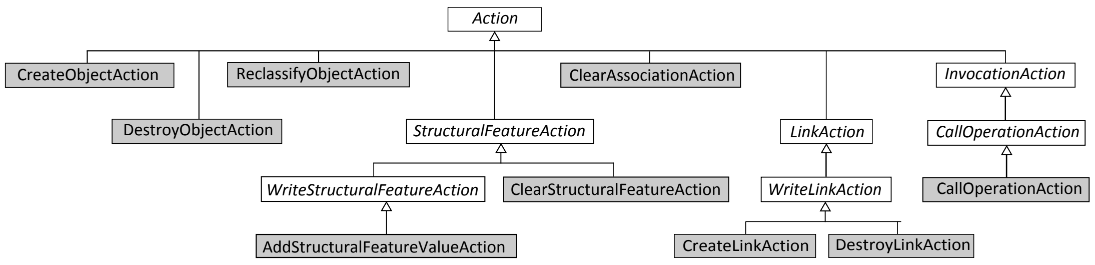


Figure 2.6. Extract of the fUML metamodel (Actions Package).

In addition to these actions, Alf also provides additional actions (to read values, declare variables, etc.) that are not explicitly described in this document since they do not impact on the correctness properties that we deal with in this thesis. As any programming language, Alf also includes a set of statements to coordinate the basic actions in action sequences, conditional blocks or loops. For more information about the Alf action language, please refer to [124].

In order to illustrate the use of the above actions, in the following we show three operations[2], which describe an excerpt of the behaviour of our restaurant chain running example (see the class diagram in Figure 2.4).

Example 4 Operation `newCourse` (in the context of class `Course`) creates a new course in the system.

```
activity newCourse(in _description:String,
in _substitutingCourses:Course[*]) {
  Course c = new Course();
  c.description = _description;
  for ( i in 1.._substitutingCourses→size() ) {
   CanBeSubstitutedBy.createLink(replaced=>c,
   replacement=>_substitutingCourses[i]);
  }
 }
```

[2]Operation methods are specified as UML activities in Alf.

Example 5 Operation addMenu (in the context of class Menu) adds a new menu to the system.

```
activity addMenu(in _name:String, in _price:Real, in
_courses:Course[3..*]) {
  if ( !Menu.allInstances()→exists(m|m.name=_name) ) {
   Menu m = new Menu();
   m.name = _name;
   m.price = _price;
   for ( i in 1.._courses→size() ) {
    IsComposedOf.createLink(menu=>m,course=>_courses[i]);
   }
  }
}
```

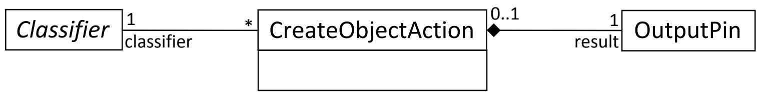
Example 6 Operation classifyAsSpecialMenu (in the context of class Menu) classifies a menu as a special menu.

```
activity classifyAsSpecialMenu(in _discount:Real) {
  if ( _discount ≥ 10 ) {
   classify self to SpecialMenu;
   self.discount = _discount;
  }
}
```

In the following we show the complete description of each Alf action we deal with in this thesis. Tables 2.2 to 2.10 show the description of each concrete action of Figure 2.6. Each table is divided into two parts:

1. *fUML*: Provides information about the action according to the fUML standard [125]. It is divided into several rows:
   - *Description*: Provides a general description of the action.
   - *Abstract syntax*: Provides an extract of the fUML metamodel showing the relevant attributes and associations of the action.
2. *Alf*: Provides information about the action according to the Alf standard [124]. It is divided into several rows:
   - *Concrete syntax*: Provides a concrete textual syntax for the action.
   - *Concrete semantics*: Describes the semantics of the action. Note that the semantics provided by Alf is more precise than the semantics provided by fUML. Besides, in some cases, both semantics are not entirely equivalent. Since our operations are based on Alf statements, in the rest of the document we assume the precise semantics described by the Alf standard.
   - *Example of use*: Provides an example of use and their meaning.
   - *Considerations*: Optionally, describes some considerations which will be applied in the rest of this document.

Table 2.2. CreateObjectAction.

fUML	*Description*	Instantiates a classifier, i.e. creates an object that conforms to a statically specified classifier and puts it on an output pin at runtime.
	Abstract syntax	 Classifier 1 classifier * CreateObjectAction 0..1 1 result OutputPin
Alf	*Concrete syntax*	`<object> = new <type>([<tuple>])`, where: `type` must resolve to a class or a constructor operation (but not both at a time) and `tuple` is an optional list of expressions used to provide the arguments for the invocation.
	Concrete semantics	The statement creates and returns an object. If `tuple` is not empty, it is used to specify values for the attributes of the new object value. If a named tuple is used, then the names must correspond to the names of the attributes of the class. Arguments are matched with attributes of the named class, with the attributes being considered as in parameters. Each argument expression must be to the corresponding attribute.

	Example of use	`c = new City();` //creates on object of the class `City` without initializing any of its attributes
		`c = new City::transferred(cityInfo);` //creates an object of the class `City` and calls the constructor `transferred` on that object with the argument `cityInfo`
		`c = new City(name=>‘‘Barcelona’’);` //creates a object of the class `City` and initializes its attribute `name`

	Considerations	Action `<object> = new <type>([<tuple>])` (when `type` is a class) can be decomposed into two actions: (1) a `<object> = new <type>()` action; and (2) a `<object>.<attribute> = value` (see Table 2.5) action for each attribute of the `tuple`. Similarly, the action `<object> = new <type>([<tuple>])` (when `type` is a constructor operation) can be decomposed into two actions: (1) a `<object> = new <class>()` action (where `class` is the class which owns the constructor operation); and (2) a `<object>.<attribute> = value` (see Table 2.5) action for each attribute of the `tuple`. In the rest of this document, we assume this composed action is always expressed in its extended version.

Table 2.3. DestroyObjectAction.

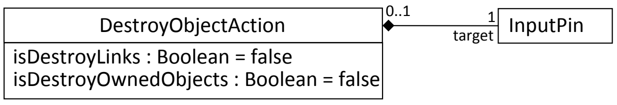
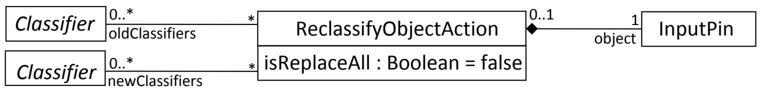
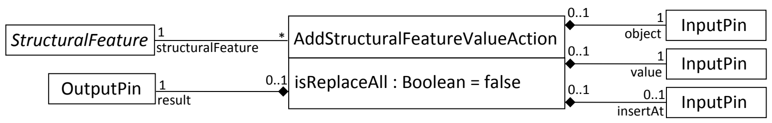
fUML	*Description*	Destroys the object on its input pin at runtime. The object may be a link object, in which case the semantics of *DestroyLinkAction* also applies.
	Abstract syntax	 DestroyObjectAction isDestroyLinks : Boolean = false isDestroyOwnedObjects : Boolean = false 0..1 1 target InputPin
Alf	*Concrete syntax*	`<object>.`**`destroy`**`()`
	Concrete semantics	Destroys an object from its class and any immediate superclasses (if such exists). Unlike fUML, in Alf object destruction is always done with *isDestroyLinks*=true and *isDestroyOwnedObjects*=true, because this is the expected high-level behaviour for object destruction. This means that links in which the `object` participates and the objects owned be the `object` are destroyed along with the `object`.
	Example of use	`c.destroy();` //destroys the object `c`

Table 2.4. ReclassifyObjectAction.

fUML	*Description*	Changes the type(s) of an object by adding given classifiers to the object and removing given classifiers from that object. Multiple classifiers may be added and removed at a time.
	Abstract syntax	 Classifier 0..* oldClassifiers * Classifier 0..* newClassifiers * ReclassifyObjectAction isReplaceAll : Boolean = false 0..1 1 object InputPin
Alf	*Concrete syntax*	**`classify`** `<object> [`**`from`** `<oldCl>] [`**`to`** `<newCl>]`
	Concrete semantics	Dynamically reclassifies an already existing object. The statement identifies an already existing `object` (which must have a class as its static type) and the classes from which (`oldCl`) and/or to which (`newCl`) the identified object is to be reclassified. All qualified names listed in the `oldCl` and `newCl` lists must resolve to classes, must be subclasses of the static type of the `object` and none of them may have a common superclass that is a subclass of the static type of `object` (that is, they must be disjoint subclasses). If the `from` list is given as "*", then all the current classes of the identified object are removed and replaced with the classes in the `to` list. In this case, the `to` list must not be empty.

	Example of use	`classify m from SpecialMenu;` //Removes `m` from `SpecialMenu`
		`classify m to SpecialMenu;` //Adds `m` to `SpecialMenu`

Table 2.5. AddStructuralFeatureValueAction.

fUML	*Description*	Adds values of the input pin to a structural feature of a given object.
	Abstract syntax	 StructuralFeature 1 structuralFeature * AddStructuralFeatureValueAction isReplaceAll : Boolean = false OutputPin 1 result 0..1 0..1 1 object InputPin 0..1 1 value InputPin 0..1 0..1 insertAt InputPin

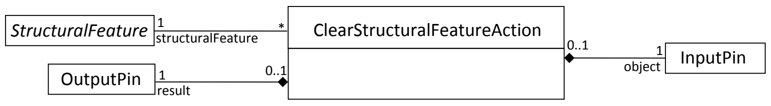
Alf	*Concrete syntax*	`<object>.<attribute> = <value>`
		`<object>.<attribute>->add([<position>],<value>)`

	Concrete semantics	Adds `value` as a new value for the `attribute` of the `object`. If `position` is not empty, it is used to specify the position where the `value` is added. Unlike fUML, in Alf add structural feature value is done with *isReplaceAll*=true, because this is the expected high-level behaviour. This means that existing value/s of the `attribute` of the `object` is/are removed before adding the new value.

	Example of use	`restaurantBranch.phone = ``111111111'';` //Sets `` ``111111111'' `` as the new value for the attribute `phone` of object `restaurantBranch`
		`restaurantBranch.phone->add(2,``999999999'');` //Adds `` ``999999999'' `` as a new value in the second position for the attribute `phone` of object `restaurantBranch`

	Considera-tions	In order to homogenize the document, in the following chapters we use the simple version of the action, i.e. `<object>.<attribute> = <value>`. However, all the conclusions over this action could be also applied to the alternative version of this action.

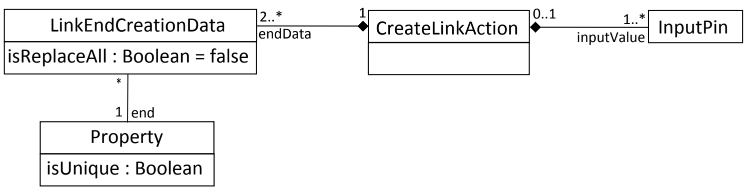
Table 2.6. ClearStructuralFeatureAction.

fUML	*Description*	Removes all values of a structural feature.
	Abstract syntax	 StructuralFeature 1 structuralFeature * ClearStructuralFeatureAction OutputPin 1 result 0..1 0..1 1 object InputPin
Alf	*Concrete syntax*	`<object>.<attribute>[[<position>]] = null`
	Concrete semantics	If `position` is not empty, removes the `position`-th value of the `attribute` from `object`. If `position` is empty, removes all values of the `attribute` from object `object`. Note that, since "null" represents the empty collection, not a value itself, the statement removes the values, it does not assign some "null" value.

	Example of use	`restaurantBranch.phone = null;` //Removes all values of `phone` from `restaurantBranch`
		`restaurantBranch.phone[2] = null;` // Removes the second value of the collection `restaurantBranch.phone`

	Considerations	In order to homogenize the document, in the following chapters we use the simple version of the action (without specifying a position), i.e. `<object>.<attribute> = null`. However, all the conclusions over this action could be also applied to the second version of this action.

Table 2.7. CreateLinkAction.

fUML	*Description*	Creates a link between two or more objects.
	Abstract syntax	 LinkEndCreationData isReplaceAll : Boolean = false 2..* endData 1 CreateLinkAction 0..1 1..* inputValue InputPin * 1 end Property isUnique : Boolean

Alf	*Concrete syntax*	`<association>`**`.createLink`**
		`([<role1>[<position1>]=>]<object1>,`
		`[<role2>[<position2>]=>]<object2>)`

	Concrete semantics	Creates a new link (i.e. an instance of an association) in the `association` with end values `object1` (with `role1`) and `object2` (with `role2`). `association` must resolve to an existing association and must not be abstract. If some association end is ordered, then the `position` of a link for the end can be indicated using an index. If an index is not given for an ordered end, then the default is *, which indicates adding the link at the end.

	Example of use	`IsLocatedIn.createLink(restaurantBranch=>rb,`
		`city=>c);` //Creates a link of the association `IsLocatedIn` between the restaurant branch `rb` and the city `c`
		`IsLocatedIn.createLink(rb,c);` //Creates the same link as before
		`IsLocatedIn.createLink(restaurantBranch[1]=>rb,`
		`city=>c);` //Inserts the `rb` at the beginning of the list of restaurant branches for the city `c`

	Considerations	As is usual in the conceptual modelling community [112], we assume between two objects there may exist at most one link, i.e. *isUnique = true*. Then, in order to homogenize the document, in the following chapters we use the simple version of the action (without specifying a position), i.e. `<association>.createLink([<role1>=>]<object1>,` `[<role2>=>]<object2>)`. However, all the conclusions over this action could be also applied to the complete version of this action.

Table 2.8. DestroyLinkAction.

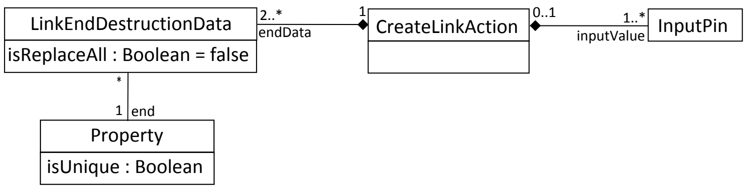
fUML	*Description*	Destroys a link between two or more objects.
	Abstract syntax	 LinkEndDestructionData isReplaceAll : Boolean = false 2..* endData 1 CreateLinkAction 0..1 1..* inputValue InputPin * 1 end Property isUnique : Boolean
Alf	*Concrete syntax*	`<association>.`**`destroyLink`** `([<role1>[<position1>]=>]<object1>, [<role2>[<position2>]=>]<object2>)`
	Concrete semantics	Destroys the link (i.e. an instance of an association) in the `association` with end values `object1` (with `role1`) and `object2` (with `role2`). If some association end is ordered, then the `position` of a link for the end can be indicated using an index. If an index is not given for an ordered end, then the default is *, which indicates destroying the link from the end.
	Example of use	`IsLocatedIn.destroyLink(rb,c);` //Destroys the link of the association `IsLocatedIn` between the objects `rb` and `c`
	Considerations	As is usual in the conceptual modelling community [112], we assume between two objects there may exist at most one link, i.e. *isUnique = true.* Then, in order to homogenize the document, in the following chapters we use the simple version of the action (without specifying a position), i.e. `<association>.destroyLink([<role1>=>]<object1>, [<role2>=>]<object2>)`. However, all the conclusions over this action could be also applied to the complete version of this action.

Table 2.9. ClearAssociation.

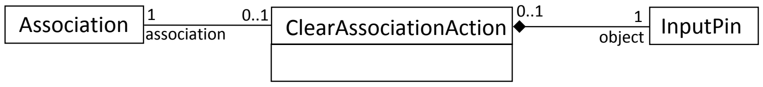
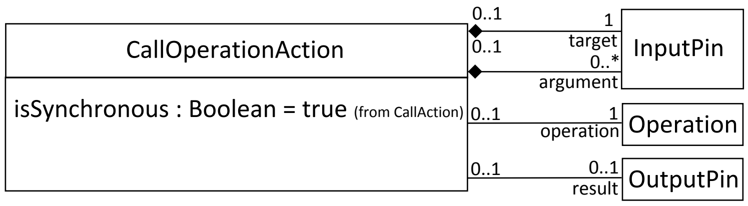
fUML	*Description*	Destroys all links of an association in which a particular object participates.
	Abstract syntax	 Association 1 association 0..1 ClearAssociationAction 0..1 1 object InputPin
Alf	*Concrete syntax*	`<association>.`**`clearAssoc`**`(<object>)`
	Concrete semantics	Destroys all links of the named `association` that have at least one end with value `object`.
	Example of use	`IsLocatedIn.clearAssoc(c);` //Destroys all links from association `IsLocatedIn` in which the city `c` participates

Table 2.10. CallOperationAction.

fUML	*Description*	Transmits an operation call request to the target object, where it may cause the invocation of associated behaviour.
	Abstract syntax	 CallOperationAction isSynchronous : Boolean = true (from CallAction) 0..1 1 target 0..1 0..* argument InputPin 0..1 1 operation Operation 0..1 0..1 result OutputPin
Alf	*Concrete syntax*	`[<result>]=<object>.<operation>([<arguments>])`
	Concrete semantics	Dispatches the `operation` in the context of the `object`. If the operation has `arguments`, they are matched to parameters of the `operation` by order. If the operation has a return parameter, then the output pin of the call operation action corresponding to that parameter is the `result` source element for the feature invocation action. Otherwise it has no result source element.
	Example of use	`menu.classifyAsSpecialMenu(_discount);` //Invokes the operation `classifyAsSpecialMenu(in r: Real)` in the context of class `Menu` with the argument `_discount`

2.3 Summary

Executable models are models with a behavioural specification detailed enough so that they can be systematically implemented or executed in the production environment. Executable models are now increasing its popularity given the benefits they can contribute: increase the productivity, reduce costs and improve the quality of software systems.

Traditional UML diagrams are not specified precisely enough to be executed. In order to overcome this issue, the OMG has recently added two standards focused on making the models executable: (1) fUML [125], an executable subset of UML that provides an abstract syntax for actions; and (2) Alf [124], a concrete syntax of fUML abstract actions.

Table 2.11 summarizes the main modification actions provided by fUML (see its complete description in Section 2.2.2), the corresponding concrete syntax provided by Alf and a brief description of the update each action performs. In addition to the actions shown in Table 2.11, Alf also provides additional actions (to read values, declare variables, etc.) that are not shown above since they do not compromise the correctness properties we deal with in this thesis. As any programming language, Alf also includes a set of statements to coordinate the basic actions in action sequences, conditional blocks or loops.

Alf allows specifying detailed behaviours at a higher level of abstraction. Such behaviours are specified more at the level of *what* is to be done, rather than *how* it is to be done in a specific implementation platform. Indeed, the real power of executable modeling going forward relies on keeping the entire behavioural specification at such a higher level of abstraction.

Executable modelling originates a new developing paradigm, aligned to MDD, where models act just like code: programming in UML. This is the next grand challenge for the Software Engineering community [161].

Table 2.11. Modification actions provided by fUML and concrete syntax in Alf.

fUML Action	*Alf Syntax*	*Description*
Create Object Action	`<object> = `**`new`**` <type>()`	Creates and returns a new `object` of class `type`.
Destroy Object Action	`<object>.`**`destroy`**`()`	Destroys the object `object`.
Reclassify Object Action	**`classify`**` <object> [`**`from`**` <oldCl>] [`**`to`**` <newCl>]`	Removes `object` from classes in `oldCl` and adds it as a new instance of classes in `newCl`.
Add Structural Feature Value Action	`<object>.<attribute> = <value>`	Sets `value` as the new value for the attribute `attribute` of object `object`.
Clear Structural Feature Action	`<object>.<attribute> = null`	Removes all values of the `attribute` from the `object`.
Create Link Action	`<association>.`**`createLink`**` ([<role1>=>] <object1>, [<role2>=>] <object2>)`	Creates a new link (i.e. association instance) in the binary association `association` between `object1` (with `role1`) and `object2` (with `role2`).
Destroy Link Action	`<association>.`**`destroyLink`**` ([<role1>=>] <object1>, [<role2>=>] <object2>)`	Destroys the link (i.e. association instance) in the binary association `association` between `object1` (with `role1`) and `object2` (with `role2`).
Clear Association Action	`<association>.`**`clearAssoc`**` (<object>)`	Destroys all links of the named `association` that have at least one end with value object.
Call Operation Action	`[<result>]=<object>. <operation>([<arguments>])`	Invokes the `operation` in the context of the `object` with `arguments`.

Quality is never an accident. It is always the result of intelligent effort.

John Ruskin

3 Quality of Models

As we introduced in Chapter 1, *quality* is a complex concept the meaning of which has been widely discussed. In the context of modelling, the **quality of a model** is the degree to which a set of *internal properties* is present.

In the context of MDD, where models are the basis of the whole development process, the quality of the models has a high impact on the final quality of software systems derived from them [121]. It means that a quality model will lead to a higher quality information system. Hence, models may directly affect both the *efficiency* (time, cost, effort) and the *effectiveness* (quality of the results) of information systems development.

The aim of this chapter is to give a global view of the state of the art in quality assessment. It is divided into three sections: Section 3.1 reviews the main quality frameworks proposed in the literature; Section 3.2 outlines a representative set of existing methods that can be used to ensure the quality of an input specification; and finally, Section 3.3 summarizes and concludes the chapter.

3.1 Frameworks for evaluating the quality of models

Most approaches to quality evaluation decompose the concept of quality into a set of lower level quality properties (also called "goals") which may be precisely measured.

At the level of **software quality**, for instance, since 2001 there is an international standard (ISO/IEC 9126 [88]) for evaluating the quality of software systems. This standard decomposes the concept of software quality into six quality properties, which are further divided into 24 quality sub-properties, measured by 113 quality metrics. Previously (in 1976), in the same line, Boehm [18] and Dromey (in 1995) [51] provided simplified alternatives consisting of a single level of properties.

Even though an international standard for evaluating the quality of software systems exists [88], no equivalent standard for evaluating specific software quality categories (such as *data*, *code*, *model*, *process*, etc.) has so far been proposed. Although there is no generally accepted guidelines, there is a broad set of research proposals devoted to evaluating the quality of *data* [134, 180, 181], *code* [166] and *processes* [119], among others.

In the context of **model quality**, which is the topic of this thesis, also relevant frameworks has been proposed. In [115] the most relevant proposals are reviewed. Some of them focus on *entity relationship (ER) models* [114, 168] or *object oriented (OO) models* [12], among others. Regarding *conceptual models*, one of the earliest and widely referred framework is the Lindland et al. framework [106]. This framework relies on four main concepts (see the boxes of Figure 3.1):

- **Language** (L): All the statements that can be made according to a specific syntax (described by means of an alphabet plus its grammar).
- **Domain** (D): All possible statements that would be correct and relevant for solving the problem.
- **Model** (M): The set of statements actually made.
- **Audience interpretation** (I): The set of statements that the audience thinks the model contains.

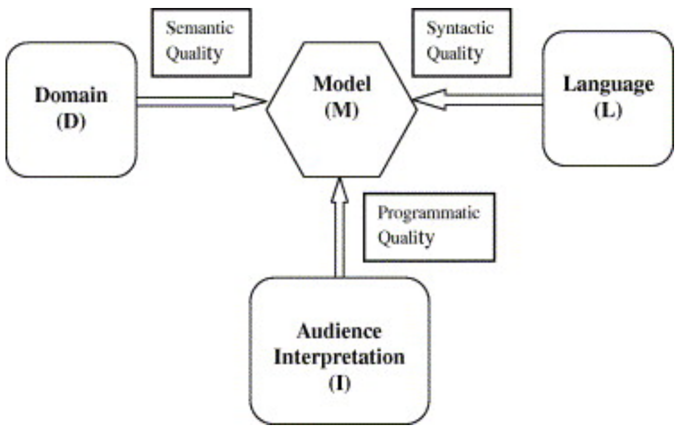


Figure 3.1. Lindland's framework main concepts (boxes) and relationships (edges).

Based on these concepts, Lindland defines three relationships (i.e. *model quality goals*) among them (see the edges of Figure 3.1):

- **Syntactic quality**: The more closely the model adheres to the language rules, the higher the syntactic quality.
- **Semantic quality**: The more similar the model and domain, the better semantic quality, the more different, the worse semantic quality.
- **Pragmatic quality**: The more similar the model and the audience interpretation, the better pragmatic quality, the more different, the worse pragmatic quality.

The Lindland framework has been later extended in several works. In 2006 Krogstie et al. [99] added other quality goals such as *organizational quality* (i.e. whether a model fulfills the goals of modelling and that all the goals of modelling are addressed through the model) and *technical pragmatic quality* (i.e. being interpretable by tools). Additionally, Soldheim et al. [164] defined two new quality goals relevant for MDD: *transformability* and *maintainability*. In the same line, in 2007, Nelson et al. [118] added *perceptual*, *descriptive* and *inferential* quality as quality goals.

Other works redefine the Lindland framework. For instance, in 2005, Unhelkar et al. [175] replaced the *pragmatic quality* by *aesthetics quality* (with focus on symmetry and consistency in order to improve the look and to help understanding).

As we will see in the next chapter, for the purpose of contextualizing the quality properties considered in this thesis, we adopt the well-known Lindland's quality framework.

3.2 Practices to improve the quality of models

In order to assess whether a model meets the above quality goals, several methods can be employed. All these methods aim to validate and verify (V&V) the model.

The IEEE [86] defines **validation** as the "confirmation by examination and provisions of objective evidence that the particular requirements for a specific intended use are fulfilled". On the other hand, the same standard defines **verification** as the "confirmation by examination and provisions of objective evidence that specified requirements have been fulfilled".

Applying the above definitions in the modelling context, **validation** is an activity that answers the question: "Are we developing the right model?", that is, whether all the knowledge in the model is sufficiently correct and relevant to the problem domain. On the other hand, **verification** is an activity that answers the question "Are we developing the model right?", that is, whether the model satisfies quality properties such as consistency. According to the ISO/IEC 9126 [88] classification, validation aims to check the *external quality* and *verification* aims to check the *internal quality*.

In this thesis, we focus on the verification of the internal quality of executable models.

The methods applicable to V&V are called **analytical methods**. Analytical methods are suitable for software V&V in general, and for model (requirements, design specifications, executable models, code, test cases, project plans, etc.) V&V in particular.

In this thesis, we classify analytical methods regarding two perspectives: (1) the mode how the analysis is done; and (2) the level of formalization. This classification is an oversimplification for the purpose of this thesis.

First, regarding the *mode how the analysis is done*, we classify analytical methods in two categories:

- **Static methods**. Static methods examine a model and reason over all the possible behaviours that might arise at run time [58]. It means that the model is read by humans, or pursued by a computer, but not executed as a program. Hence, static methods work at "compile time".
- **Dynamic methods**. Dynamic methods operate by executing a program (in our case, a model) and observing its executions [58]. It means that the model is run (or executed) by means of a computer. Hence, dynamic methods work at "run time".

Second, regarding their *level of formalization*, we classify analytical methods in two categories:

- **Formal methods**. Wing [184] describes formal methods as "mathematically based techniques for describing system properties. Such formal methods provide frameworks within which people can specify, develop and verify systems in a systematic, rather than ad-hoc manner". In [38] Clarke reports that "in the past, the use of formal methods in practice seemed hopeless. The notations were too obscure, the techniques did not scale, and the tool support was inadequate or too hard to use. There were only a few non-trivial case studies and together they still were not convincing enough to the practicing software or hardware engineer. Few people had the training to use them effectively on the job". However, in the last decades we have begun to see a more promising picture of formal methods because of the advent of formal notations (such as the Z notation [165]), the use of formal methods on industrial case studies [48] (which increases the confidence in using formal methods) and the commercial tools for supporting formal analysis. Plate et al. [144] argue that the main benefit of using formal methods is that "formal notations can be unambiguously interpreted and provide extensive means to express abstraction. These notations can be used to formally verify characteristics of the design". Formal methods has been mainly applied in hardware design [97]. However, they cut across almost all areas in computer science and engineering, so they are also suitable for software design.
- **Non-formal methods**. Unlike formal methods, non-formal methods do not try to follow a rigorous approach but to use informal techniques. Non-formal methods has the advantage that the user does not need be an expert in understanding mathematical models. They are easy to illustrate and can be used to V&V models written in natural language increasing the participation from non-technical stakeholders. As a drawback, given its non-formality, they can be ambiguous and can provide a non-precise result.

In the following, we briefly review some of the existing analytical methods, classifying them into the above categories (see Figure 3.2). Note that, again, our classification is an oversimplification which only includes a subset of the many existing methods devoted to V&V. Note also that, although we classify model checking as a dynamic method, some types of model checking may be also considered as a static method [72].

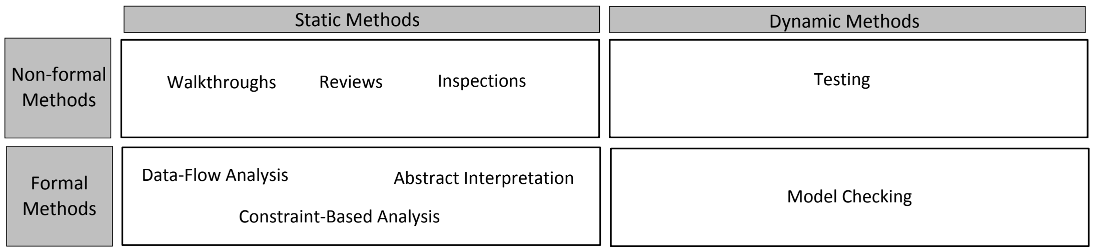


Figure 3.2. Verification&Validation relevant methods.

Walkthrough

The IEEE [85] standard defines a *walkthrough* as “a static analysis technique in which a designer or a programmer leads members of the development team and other interested parties through a *software product*, and the participants ask questions and make comments about possible errors, violation of development standards, and other problems”. A *software product* might be a software design document or program source code, but use cases, business process definitions, test case specifications, and a variety of other technical documentation may also be walked-through.

In general, a walkthrough has one or two broad objectives: (1) to gain feedback about the technical quality or content of the software product analyzed; and/or (2) to familiarize the audience with the content.

Review

The IEEE [85] standard defines a *review* as “a process or meeting during which a software product is presented to project personnel, managers, users, customers, user representatives, or other interested parties for comment or approval”. The IEEE [85] also defines a *technical review* as “a systematic evaluation of a software product by a team of qualified personnel that examines the suitability of the software product for its intended use and identifies discrepancies from specifications and standards. Technical reviews may also provide recommendations of alternatives and examination of various alternatives”. Technical reviews differ from *walkthroughs* in its specific focus on the technical quality of the product reviewed.

The purpose of a technical review is to arrive at a technically superior version of the work product reviewed, whether by correction of defects or by recommendation or introduction of alternative approaches.

Inspection

Inspection is the key method for static and non-formal analysis. The IEEE [85] standard defines an *inspection* as “a visual examination of a software product to detect and identify software anomalies, including errors and deviations from standards and specifications. Inspections are peer examinations led by impartial facilitators who are trained in inspection techniques. Determination of remedial or investigative action for an anomaly is a mandatory element of a software inspection, although the solution should not be determined in the inspection meeting”.

Compared to the *technical reviews* and *walkthoughs*, inspections are more structured. The IEEE standard [85] states that inspections should be done according to the project plan. It is usually not just done on demand. An inspection is therefore regarded as a proper examination activity rather than an activity to evaluate a work product for suitability.

The inspection process was developed by Fagan [60] in the mid-1970s and it has later been extended and modified. The process should have entry criteria that determine if the inspection process is ready to begin. This prevents unfinished work products from entering the inspection process. The entry criteria might be a checklist including items such as “The document has been spell-checked”.

In the early 1970s, walkthroughs, reviews and inspections began to be applied to analyze the source code of programs. Myers [117] argues that “for many years, most of us in the programming community worked under the assumptions that programs are written solely for machine execution and are not intended for people to read, and that the only way to test a program is to execute it on a machine. This attitude began to change through the efforts of program developers who first saw the value in reading code as part of a comprehensive testing and debugging regimen”.

Since then, the above techniques has been mainly applied to analyze source code [65, 145]. However, these techniques can also be applied in earlier phases of the software development such as requirements specification [146] or design [60, 171].

Data-Flow Analysis

Data-Flow Analysis derives information about the dynamic behaviour of a program by only examining the static code [120]. It gathers information about possible set of values calculated at various program points. The *Control Flow Graph* (CFG), which represents all alternatives of control flow in a program, is utilized for determining where to propagate the values between different program points.

There are several Data-Flow Analysis that compute various properties, such as the *liveness analysis* (to determine which variables are referenced beyond a particular program point) or *reaching definitions analysis* (to statically determine which definitions may reach a given point in the code).

Data-Flow Analysis was first used as a data structure in compilers [116] for code optimization purposes. Later, this method was adopted extensively in the software engineering community, in particular by the software testing community (e.g. [15, 34, 78]). Besides, there have been works applying Data Flow Analysis on models, mainly based on UML Sequence Diagrams [67].

Constraint-Based Analysis

Constraint-Based Analysis is a technique for inferring implementation types (or representation/concrete types) [5]. Implementation types are sets of classes. In contrast to interface types (or abstract/principal types), they deliver the low-level information that is typically needed by compilers and other tools.

Constraint-Based Analysis consists of two phases [7]: *constraint generation* and *constraint resolution*. The first phase produces constraints from a program text that give a declarative specification of the desired information about the program. The second phase computes this desired information by finding the sets of values that satisfy the constraints. A solution is an assignment from set variables in the constraints to the finite descriptions of such sets of values.

So far, Constraint-Based Analysis has been mainly applied to analyze source code of programs [79].

Abstract Interpretation

Abstract Interpretation [47] is a theory of discrete approximation which can be applied to the semantics of (specification or programming) languages. Abstract Interpretation formalizes the idea that a semantics can be more or less precise according to the considered observation level [46].

Given a programming or specification language, Abstract Interpretation consists of giving several semantics linked by relations of abstraction. A semantics is a mathematical characterization of a possible behaviour of the program. The most precise semantics, describing very closely the actual execution of the program, are called the *concrete semantics*. For instance, the concrete semantics of an imperative programming language may associate to each program the set of execution traces it may produce, where an *execution trace* being a sequence of possible consecutive states of the execution of the program; a *state* typically consists of the value of the program counter and the memory locations (globals, stack and heap). More abstract semantics are then derived; for instance, one may consider only the set of reachable states in the executions (which amounts to considering the last states in finite traces).

Abstract Interpretation has been successfully used in different areas for the verification and optimization of systems [45].

Testing

Testing is probably the most popular method used for the dynamic verification of a program or system. It does this by running a discrete set of test cases, where a test case consists of input values and their expected output. The test cases are suitably selected from a finite, but very large, input domain. During testing the actual behaviour is compared with the intended or expected behaviour. The emphasis of software testing is to validate and to verify the design and the initial construction. Unlike to many other engineering products, where the emphasis is on testing the correct reproduction, software testing is part of the development steps, not the manufacturing process.

Agarwal [4] states a number of rules that can serve well as testing objectives: (1) Testing is a process of executing a program with the intent of finding an error; (2) A good test case is one that has a high probability of finding an undiscovered error; and (3) A successful test is one that uncovers an as-yet undiscovered error. Then, the major testing objective is to design tests that systematically uncover types of errors with minimum time and effort.

Testing techniques has been extensively used for testing source code of programs. Various code-driven testing frameworks have come to be known collectively as *xUnit*. These frameworks allow testing of different elements (units) of software, such as functions and classes. The main advantage of xUnit frameworks is that they provide an automated solution with no need to write the same tests many times, and no need to remember what should be the result of each test. Such frameworks are based on a design by Beck [14], originally implemented for Smalltalk as *SUnit*. From there, the framework was ported to other languages such as JUnit (for Java), CppUnit (for C++), NUnit (for .NET). They are all referred to as *xUnit* and are usually free, open source software. They are now available for many programming languages and development platforms.

In the context of MDD, testing techniques has also been applied for testing models. In these approaches the artifact under test is a model instead of source code. Some examples are [133], an approach for testing UML design models to uncover inconsistencies; [50] an Eclipse plug-in for animating and testing UML models; and [169] a method which applies the principles of TDD (Test-Driven Development) to conceptual modeling.

Model Checking

Model checking [39] is an automatic technique for verifying systems. It was originally developed in 1981 by Clarke and Emerson [36, 37]. The essential concept behind model checking is to (mathematically) prove whether a given model satisfies a certain specification property (such as *liveness* (reachability of path), *safety* (deadlock freeness), etc.) by generating and analyzing all the potential executions at run-time and evaluating if for each (or some) execution the given property is satisfied. Sometimes a more formal definition can be found: Given a model M and a formula ϕ, model checking is the problem of verifying whether or not ϕ is true in M.

As shown in Figure 3.3, traditional model checking is composed of three major steps:

1. Define a formal model of the system that is subject to verification by creating a model of the system in a propositional temporal logic language that fits the model checker's input language. Those modeling languages are usually tight coupled to the model checker itself, like the PROMELA language of the SPIN model checker [84].

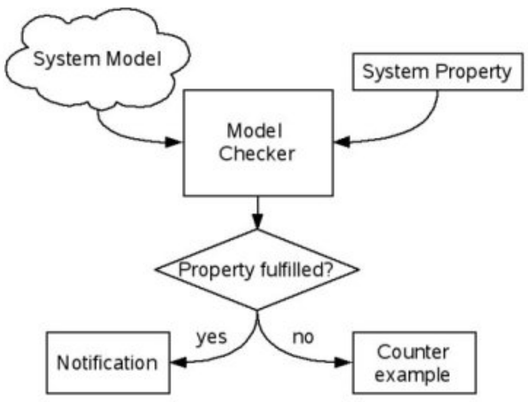


Figure 3.3. Model checking overview.

2. Provide a particular system property that should be proved. In other words, a question about the system's behaviour is formulated that should be answered by the model checker.

3. Invoke the model checking tool and receive a notification whether the given system property was fulfilled or not. In case the system property could not be verified, a counterexample execution is generated to finger-point to the source of error in the input model.

Model checking methods suffer from the state-explosion problem [177], (i.e. the number of potential executions to analyze grows exponentially in terms of the size of the model, the domains of the parameters, etc) even though a number of optimizations are available in order to alleviate the problem by using a subset or an abstraction of the set of states. Unfortunately, their use tends to restrict the set of analysis or verification questions that can be answered, making it impossible to discuss the methods without some taxonomy of the questions.

Model checking has been applied so far mainly to hardware circuit designs [28, 109] and communication protocols [55]. In the last decade, model checking has also been applied to software models verification [104, 157] (see more details in Chapter 11).

3.2.1 The emergence of Lightweight Formal Methods

Formal methods have offered great benefits, but often at a heavy price. Formal methods can provide guarantees of correctness, but, except in safety-critical work, the cost of full verification is prohibitive and early detection of errors is a more realistic goal. Heitmeyer [80] argues that "a significant barrier in applying formal methods is the widespread perception among software developers that formal notations and formal analysis techniques are difficult to understand and apply. Moreover, software developers often express serious doubts about the scalability and cost-effectiveness of formal methods". Hence, for everyday software development, in which the pressures of the market don't allow full-scale formal methods to be applied, a more lightweight approach is called for.

The term "lightweight formal methods" was popularized over fifteen years ago, following the publication of a round-table article "An Invitation to Formal Methods" by Saiedian [156]. Two contributions came under the heading "Formal Methods Light".

In one article [93] Jones argued for a *rigorous* approach with a formal basis, but suggested using a less-than-completely formal approach (rather than a fully formal approach) in most development cases. According to Jones, the use of formal methods in a lighter way is both a key to using them on larger-scale applications and a way of penetrating fields outside the safety-critical area.

On the other hand, Jackson and Wing [91] suggested that, "to make analysis economically feasible, the cost of specification must be dramatically reduced". Hence, they advocated a lightweight approach to formal methods in which cost-effectiveness is achieved by developing partial specifications (with a focused application) than can be analyzed automatically. According to them, the elements of a lightweight approach are the following:

- *Partiality in language.* Until now, specification languages have been judged primarily on their expressiveness, with little attention paid to tractability. Some languages were from the start designed with tool support in mind, but they are the exception. Tools designed as an after-thought can provide only weak analysis, such as type checking. The tendency (in Z [165] especially) to see a specification language as a general mathematical notation is surely a mistake, since such generality can only come at the expense of analysis (and, moreover, at the expense of the language's suitability for its most common applications).

- *Partiality in modelling.* Since a complete formalization of the properties of a large system is infeasible, the question is not whether specifications should focus on some details at the expense of others, but rather which details merit the cost of formalization. The naive presumption that formalization is useful in its own right must be dropped. There can be no point embarking on the construction of a specification until it is known exactly what the specification is for; which risks it is intended to mitigate; and in which respects it will inevitably prove inadequate.

- *Partial analysis.* A sufficiently expressive language, even if designed for tractability, cannot be decidable, so a sound and complete analysis is impossible. Most specifications contain errors, and so it makes more sense to sacrifice the ability to find proofs than the ability to detect errors reliably. A common objection to this approach is that it reduces analysis to testing, since one can no longer infer from the omission of reported errors the absence of actual errors. But this much-touted weakness of testing is not its major flaw. The problem with testing is not that it cannot show the absence of bugs, but that it fails to show their presence. A model checker that exhausts an enormous state space finds bugs much more reliably than conventional testing techniques, which sample only a minute proportion of cases.

- *Partiality in composition.* For a large system, a single partial specification will not suffice, and it will be necessary to compose many partial specifications, at the very least to support some analysis of consistency. How to compose different views of a system is not well understood, and has only minimal support from specification languages, since it does not fit the standard pattern of "whole-and-part" composition.

Both articles [91, 93], in different ways, were calling on the formal methods community to develop methods and tools that provide at least some of the benefits of formalism, without requiring the wholesale application of highly specialized technology.

Later publications suggest to work on the same direction. For instance, Heitmeyer [80] argues the need for *practical* formal methods and suggests a number of guidelines for making formal methods lightweight (and thus more accessible to software developers), such as: (1) minimize effort and expertise needed to apply the method (i.e. offer a language that software developers find easy to use and easy to understand, make formal analysis as automatic as possible, and provide good feedback); (2) provide a suite of analysis tools; (3) integrate the method into the user's development process; and (4) provide a powerful and customizable simulation capability. In the same line, Larsen et al. [103] advocate for the usage of formal methods in an agile context. They argue that both agile and formal methods are just that: sets of techniques that should be combined to suit the needs of the product and the character of the development team.

As an example, a popular lightweight formal method is the Alloy analyzer [89], a tool which can be used to analyze (partial) models written in the Alloy specification language. The Alloy analyzer can generate instances of model invariants, simulate the execution of operations defined as part of the model, and check user-specified properties of a model. It can perform incremental analysis of (partial) models as they are constructed, and provide immediate feedback to users.

In this thesis we follow the principles of lightweight formal methods promoted by the above authors. We believe that, in order to software practitioners can benefit from formal methods, these methods need to be user-friendly, robust and powerful. It means that the designer should not need to have advanced mathematical training nor need he have theorem proving skills.

As we will see in the next chapters, in order to make formal methods more accessible to software developers, in this thesis we propose a set of methods aligned to the fundamentals of lightweight formal methods.

3.3 Summary

Even though an international standard for evaluating the quality of a software system exists [88], no equivalent standard for evaluating the quality of a model has so far been proposed. However, there have been several proposals for evaluating it. For the purpose of contextualizing the quality properties considered in this thesis, we adopt the well-known Lindland's quality framework [106] (see Section 3.1).

In order to assess whether a model meets the desired quality goals, several methods can be employed. In this chapter, we have reviewed and classified a subset of the most relevant analytical methods that have been used in several fields of computer science, both in hardware and software (mainly to source code) verification:

- *Static and non-formal methods*: Walkthroughs, reviews and inspections.
- *Static and formal methods*: Data-Flow Analysis, Constraint-Based Analysis and Abstract Interpretation.
- *Dynamic and non-formal methods*: Testing.
- *Dynamic and formal methods*: Model Checking.

Additionally to the above methods, in the last years, the emergence of the lightweight formal methods (see Section 3.2.1) has gained attention. The term “lightweight” is used to indicate that the methods can be used to perform partial analysis on partial specifications, without a commitment to developing and base-lining complete, consistent and formal specifications [52].

As we explain in the next chapters, in this thesis we develop several lightweight and static methods for verifying internal correctness properties of executable models.

Part III

Contributions

Nothing is particularly hard if you divide it into small jobs.

Henry Ford

4 Framework Overview

As we introduced in the previous chapters, the aim of this thesis is to provide a set of lightweight methods to help the designers improve the internal quality of their executable models. The methods provided as part of this thesis are organized in a *verification framework*.

This chapter provides an overall picture of the verification framework we propose in this thesis. It is divided into five sections: Section 4.1 provides a broad overview of the three axis that compose our framework; Sections 4.2 to 4.4 describe each axis in detail; and finally, Section 4.5 summarizes and concludes the chapter.

4.1 Presentation

According to the vision of Weber et al. [182], we understand a framework as a holistic and concise description of *concepts* and *methods* relating to a specific *domain*. In this thesis, as we introduced in Chapter 1, we propose a framework to help the designers improve the internal quality of their executable models. Then, our framework is based on three axis (see Figure 4.1): (1) the *executable models* (domain); (2) the *correctness properties* (concepts) that executable models should accomplish; and (3) the *verification methods* (methods) that can be used to determine whether the correctness properties are fulfilled by the input executable model. In the next sections we review each of these axis.

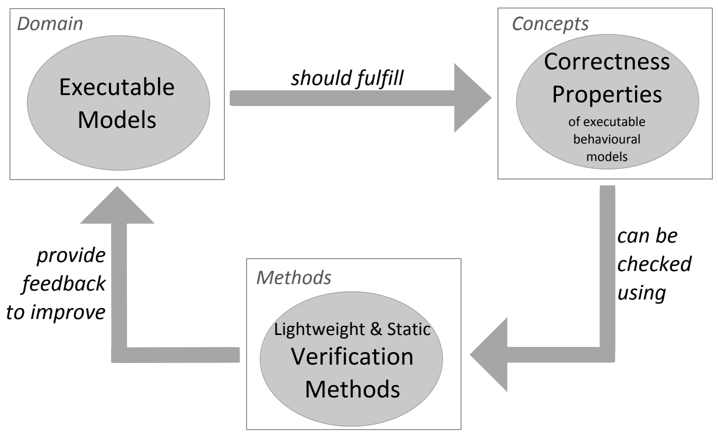

Domain
Executable Models
should fulfill
Concepts
Correctness Properties
of executable behavioural models
can be checked using
Methods
Lightweight & Static Verification Methods
provide feedback to improve

Figure 4.1. Framework overview.

4.2 Executable Models

The first axis of our framework consists in the executable models. As we described in Chapter 2, an executable model is a model with a behavioural specification detailed enough so that it can be systematically implemented or executed in the production environment.

Executable models are the domain in which the correctness properties are defined. They are also the domain in which our verification methods are applied.

As we introduced in Chapter 2 an executable model (see Figure 4.2) is composed by a *structural model* and a *behavioural model* (in this thesis, we assume the behavioural model is described as a set of low-level operations detailed enough using an action language). Besides, when not otherwise indicated, we assume executable models are written using the following OMG standards: (1) **UML** (Unified Modeling Language) [126], which is used to specify the structural model; (2) **OCL** (Object Constraint Language) [123], which is used to specify the integrity constraints that cannot be graphically expressed using UML; and (3) **Alf** [124] (Action Language for Foundational UML), which is used to specify the behaviour of low-level operations in the context of the structural model.

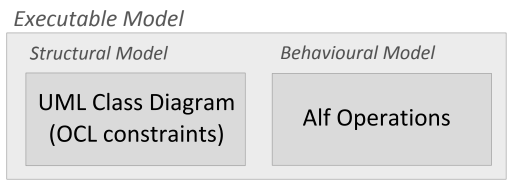


Figure 4.2. Executable Models overview.

However, as we explain in Chapter 8, the ideas presented in this document can be adapted to models specified by means of other languages such as model transformations languages.

4.3 Correctness Properties

The second axis of our framework consists in the correctness properties that we believe all executable models should accomplish. One of the contributions of this thesis has been to identify and precisely define several correctness properties over executable models, in particular, over the behavioural model.

In the following we briefly describe the properties addressed in our framework and contextualize them in the Lindland et al. framework for evaluating the quality of models [106]. A deeper and more formal definition of each correctness property may be found on the next chapters.

Syntactic correctness

An action-based operation is *syntactically correct* if all the statements in the operation conform to the syntax of the language in which it is described (i.e. Alf action language [124]).

Example 7 Consider the excerpt of the class diagram shown in Figure 4.3 and the operation `addCourseToMenu` (in the context of class `Menu`) to add a course to the `self` menu.

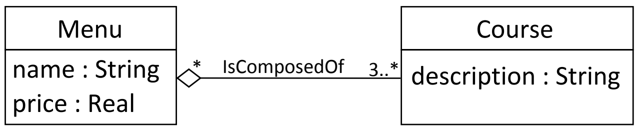


Figure 4.3. Excerpt of the structural model.

```
activity addCourseToMenu(in _course:  Course) {
  IsComposedOf.createLink(course=>_course);
}
```

The above operation is not syntactically correct because of one parameter (the menu) is missing in the create link action. Then, this action does not conform to the concrete syntax for the create link action we introduced in Chapter 2.

Note that we focus on the syntactic correctness of the operation, i.e. we assume that the UML class diagram is syntactically correct (e.g. associations are defined between at least two classes, multiplicities are coherent, and so on).

Executability

The *executability* of an action-based operation is its ability to be executed without breaking the integrity constraints defined in the structural model. Executability can be studied regarding two levels of correctness. On the one hand, an action-based operation is *weakly executable* when there is a *chance* that a user may successfully execute the operation, i.e. when there is at least an initial state of the system for which the execution of the actions included in the operation evolves this state to a new system state that satisfies all the integrity constraints of the structural model. On the other hand, an action-based operation is *strongly executable* when it is *always* successfully executed, i.e. when every time a user executes the operation, the effect of the actions included in it evolves the initial state of the system to a new system state that satisfies all the integrity constraints of the structural model.

Example 8 Consider the excerpt of the class diagram shown in Figure 4.3 and the operation `addMenu` (in the context of class `Menu`) to add a new menu to the system.

```
activity addMenu(in _name: String, in _price: Real) {
  Menu m = new Menu();
  m.name = _name;
  m.price = _price;
}
```

Since we consider that operations are executed in an atomic transaction, the above operation is not weakly neither strongly executable because of, after its execution, we always reach a system state in which the new menu is not linked to any course, a situation forbidden by the minimum 3 multiplicity of association `IsComposedOf` at the role `course` (Cmin(`IsComposedOf,course`)=3).

We consider the two levels of correctness are important. On the one hand, weak executability guarantees a basic level of correctness that could probably be achieved directly by the designers without using verification tools. On the other hand, strongly executability guarantees a full level of correctness that is more difficult to be achieved directly by the designers (see the results of our experiment in Chapter 10). In any case, and taking into account that weak executability is easier to verify than strong executability, the designer should choose which level of correctness she wants to guarantee.

Completeness

A set of action-based operations is *complete* when all possible changes on the system state can be performed through the execution of these operations. Otherwise, there will be parts of the system that users will not be able to modify since no available behaviour addresses their modification.

Example 9 Consider the excerpt of the class diagram shown in Figure 4.3 and a behavioural model composed by the operations `addCourse` (in the context of class `Course`) to add a new course to the system, and `deleteMenu` (in the context of class `Menu`) to remove an existing menu from the system.

```
activity addCourse(in _description:  String, in _category:
CourseCategory) {
  Course c = new Course();
  c.description = _description;
}
```

```
activity deleteMenu() {
  self.destroy();
}
```

The above behavioural model is incomplete since, for instance, actions to remove courses or to create menus are not specified, forbidding users to perform such kind of changes on the data.

Considerations

The above correctness properties may be contextualized in the Lindland et al. framework for evaluating the quality of models [106] we introduced in Chapter 3.

According to this framework, our correctness properties may be classified in two of the three model quality goals (see Figure 4.4):

- *Syntactic quality.* Since **syntactic correctness** relates the model with the language rules, we consider this property works at the syntactic level of the model's correctness. Then, ensuring syntactic correctness improves the syntactic quality of the executable model we are analyzing.
- *Semantic quality.* Since **executability** and **completeness** relate the model with the correct interpretation of the domain, we consider these properties work at semantic level of the model's correctness. Then, ensuring executability and completeness improve the semantic quality of the executable model we are analyzing.

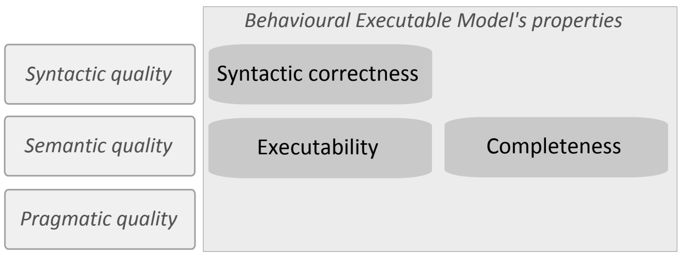


Figure 4.4. Contextualization of our correctness properties.

The more lower the quality goal, the more fundamental the correctness property is. It means that the syntactic quality ensures a $fundamental$ quality level that all executable models should guarantee; the semantic quality ensures a $significant$ quality level that all executable models should also guarantee; and the pragmatic quality ensures a *desirable* quality level that all executable models could guarantee. According to this classification, we believe that *syntactic correctness*, *executability* and *completeness* are mandatory properties that all correct executable models should fulfill.

The above correctness properties can be checked independently. However, we recommend to check them in the following order (see Figure 4.5):

1. Firstly, we suggest to verify the **syntactic correctness** property of each operation of the behavioural model. Syntactic correctness is a requisite to guarantee the rest of the properties, then, it should be checked at the beginning.
2. Secondly, we suggest to verify the **executability** and **completeness** properties in parallel. Any change for correcting a non-executable operation or a non-completeness behavioural model (for instance, adding an action to an operation) may influence both properties. Then, we recommend to check both properties iteratively until we reach a correct model.

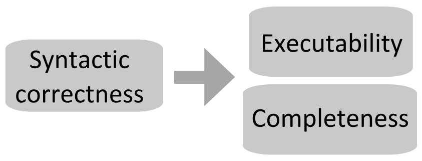


Figure 4.5. Suggested order to verify our correctness properties.

4.4 Verification Methods

All the above correctness properties aim to improve the internal quality of models, so they can be checked using analytical methods. Then, the last axis of our framework consists in the verification methods that can be used to check whether an executable model satisfies the mentioned correctness properties. As part of this thesis, we have developed a set of methods for verifying such correctness properties. Each of them verifies a specific property over an input executable model.

All the verification methods proposed in this thesis are based on an static analysis of the input executable model, i.e. on examining the model (without executing it) and reason over all possible behaviours that might arise at run time. On the other hand, the verification methods proposed in this thesis are lightweight. It means that they are formal (since they reason over a model formalized in a specific language) but they are not too costly to be applied (since, for instance, they do not require translate the input model into a more precise language in order to perform the verification).

Each of our verification methods (see Figure 4.6) takes as input an executable model and return either a positive answer, meaning that the behavioural model fulfills the desired property, or a corrective feedback expressed in the same language used to model the behaviour. In

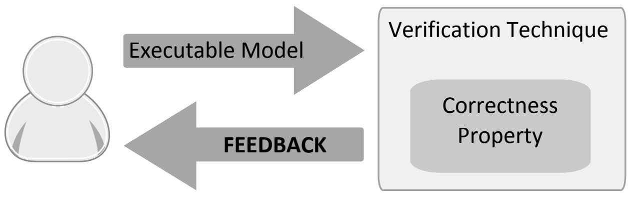


Figure 4.6. Verification methods overview.

the following chapters the several methods provided as part of our verification framework are described in detail.

At this point it is important to note that syntactic correctness has been widely studied in the context of programming languages and also in the context of modelling languages. Then, in Chapter 5 we do not focus on the algorithm to check the syntactic compliance but we focus on the syntactic rules that must be checked during the verification. On the other hand, executability (see Chapter 6) and completeness (see Chapter 7) have been less studied so far. Besides, given its complexity, we consider the study of these properties is the main contribution of this thesis. This is the reason why in this thesis we put special emphasis on these two properties.

If we consider the verification methods as a black box, all the methods proposed in this thesis follow the guidelines proposed by Heitmeyer [80] to producing practical formal methods. In particular, all of them hold the following features:

- Use of an easy language. Our methods reason over a model formalized in the same language used by the designers. The input model and the feedback provided are expressed in the same language used by the reasoning engine, facilitating the understandability of the whole method.

- Automatization. All our methods are "pushbutton", meaning that they can work with autonomy. Only when verifying the *strongly executability* (see Chapter 6), the user intervention may be required in order to make the results of the reasoning more accurate.

- Feedback. The designer is provided with easy-to-understand feedback useful in correcting the possible detected errors. Instead of a counterexample, our methods provide the source of the errors and one or more possible reparations to solve them.

- Suite of tools. Our tools are integrated in a verification framework to work together.

- Integration into the development process. CASE tools can benefit from our verification methods if, once the designer has defined the executable model, the CASE tool integrates our methods in order to verify the correctness properties proposed in this thesis. We believe that, in order to be integrated into the development process, all the methods should provide a response in a cost-effective time. In this sense, our verification methods provide feedback in a reasonable time since they do not require to execute the model.

Since all the methods proposed as part of our framework are lightweight, we believe they are useful for the software engineering community.

As a trade-off, the methods proposed in this thesis can only work over executable models with limited expressiveness. Regarding the structural model, our method does not take into account all possible integrity constraints but only those that conform with one of the types described in Section 2.2.1 (see Chapter 2). Regarding the behavioural model, our method is not able to verify operations that contain recursive invocations (since recursive invocations, as we will see later, generate infinite paths). Nevertheless, recursive operations could be transformed into their imperative counterparts before the application of our methods.

4.5 Summary

In order to organize the contributions of this thesis, we propose a verification framework to help the designers improve the internal quality of their executable models. Our framework is based on three axis:

1. **Executable models**: Executable models are the subject of our analysis.
2. **Correctness properties**: There are several correctness properties that executable models should accomplish. We identify and define three fundamental correctness properties over behavioural executable models (i.e. action-based operations): *syntactic correctness*, *executability* and *completeness*.
3. **Verification methods**: Several verification methods that can be used to determine whether the above correctness properties are satisfied by the input executable model.

In the next chapters we study in depth each of the proposed properties as well as the verification methods we have developed to check them.

Language is a process of free creation; its laws and principles are fixed, but the manner in which the principles of generation are used is free and infinitely varied. Even the interpretation and use of words involves a process of free creation.

Noam Chomsky

5 Syntactic correctness

The aim of this chapter is to precisely define the notion of *syntactic correctness* and to describe the fundamentals to check whether an action-based operation satisfies this property.

This chapter is divided into three sections: Section 5.1 precisely defines the *syntactic correctness* property; Section 5.2 provides the fundamentals to determine whether an action-based operation satisfies this property; and finally, Section 5.3 summarizes and concludes the chapter.

5.1 Syntactic Correctness Definition

All languages have a syntax, i.e. a set of rules about how elements of the language can be combined together meaningfully in that language. For instance, Latin has their own syntax, and so do Java, XML and UML. Then, specifications written in a specific language must comply with the syntax imposed by the language in which they are defined. This relationship between the specification and the language in which it is described is known as *conformance*.

As an example, in the context of UML, a UML model must conform to the UML metamodel (which defines the abstract syntax all the models should satisfy). In the same way, the UML metamodel must conform to the MOF metametamodel (which, in its turn, must conform to itself). Figure 5.1 shows these relationships between models. To verify the syntactic correctness of a model (in our case, a set of action-based operations), we focus on the *conformsTo* relationship between the model (i.e. the operations) and its metamodel (i.e. the UML/fUML metamodel).

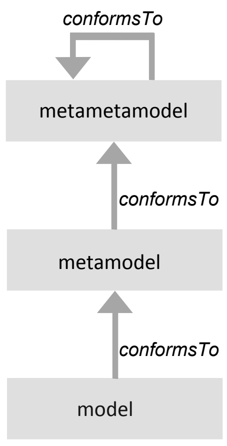


Figure 5.1. Conformance relationship.

In order to precisely define the conformance rules, the UML [126] and fUML [125] metamodels include a set of constraints - also called Well-Formedness Rules (WFR) - that restrict the possible set of valid (or well-formed) models.

At this point it is important to note that the UML and fUML metamodels are not completely consistent regarding its WFRs. On the one hand, the fUML metamodel includes some WFR which are not defined on the UML metamodel. Besides, fUML does not explicitly include most of the WFR contained in the UML metamodel. However, since fUML is a subset of UML, we assume all fUML models must also satisfy the WFR defined in the UML metamodel. This is the reason why, in the rest of the chapter, we mention both UML and fUML. Note also that Alf does not define additional WFR, since it only defines a concrete syntax conforming to the fUML abstract syntax. This means that the syntactic correctness is defined with respect to the abstract syntax.

The UML/fUML metamodels include WFRs for each element of the metamodel. In this thesis we only focus on such WFRs that affect action elements, since they are the ones aimed at preventing syntactic errors in action-based operations. As an example, the UML metamodel includes a WFR to ensure that, when the value of an attribute is modified (using the action $AddStructuralFeatureValueAction$), the type of the new value is the same as the type of the attribute.

In the following we show how this WFR may be formally expressed in OCL. Check Figure 5.2 to see the metaclasses which participate in this WFR.

WFR: The type of the value input pin is the same as the type of the structural feature.

```
context WriteStructuralFeatureAction inv:  self.value->notEmpty()
implies self.value.type = self.structuralFeature.type
```

The above WFR expresses that, if the value of the input pin is not empty, then the type of this value (`self.value.type`) must be equal to the type of the attribute - i.e. the structural feature - to be modified (`self.structuralFeature.type`). This WFR belongs to

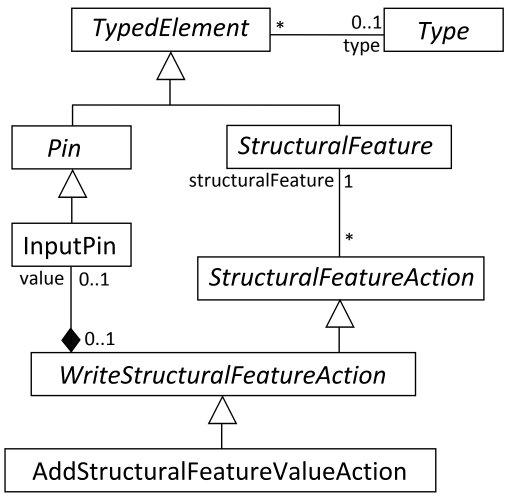


Figure 5.2. Excerpt of the UML metamodel.

the *WriteStructuralFeatureAction* metaclass. Since *AddStructuralFeatureValueAction* specializes such metaclass, the action *AddStructuralFeatureValueAction* must also satisfy this WFR.

Then, we consider an Alf-based operation is syntactically correct when all the actions included in the operation satisfy all the WFRs defined in the UML/fUML metamodels. Note that each rule only affects a unique action, i.e. there are no WFRs that affect the combination of several actions.

We denote by *IsSyntacticallyCorrect(a,op,SM)* the proposition that represent the action *a* in the operation *op* is syntactically correct with respect to the structural model SM, i.e. it satisfies all the WFRs defined in the UML/fUML metamodels.

Then, more formally:

Let ExM = ⟨SM,BM⟩ be an executable model, an operation op ∈ BM is **syntactically correct** iff ∀ a ∈ Actions(op) IsSyntacticallyCorrect(a,op,SM).

Where *Actions(op)* returns a list of all actions included in the operation *op*.

Example 10 Consider the excerpt of the class diagram shown in Figure 5.3 and the operation `setCategory` (in the context of class `Course`) to modify the category of the `self` course.

```
activity setCategory(in _newCategory:  Integer) {
  self.category = _newCategory;
}
```

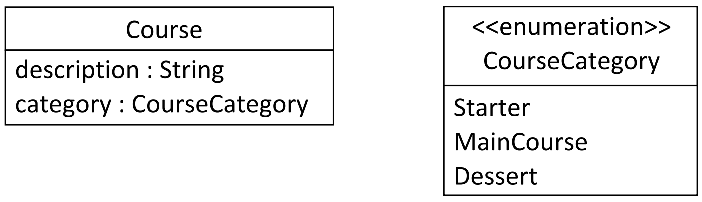


Figure 5.3. Excerpt of the structural model.

The above operation is not syntactically correct because of the type of the `_newCategory` parameter (`Integer`) is not the same as the type of the attribute `cateogry` (`CourseCategory`). Then, in order to correct this syntactic error, the designer should change the type of the `_newCategory` parameter to a `CourseCategory`. The repaired operation is:

```
activity setCategory(in _newCategory:  CourseCategory) {
  self.category = _newCategory;
}
```

Note that, in the context of programming languages, the term *syntactic correctness* simply refers to the conformity to a concrete syntax. In this thesis, we extend this meaning to the conformity to the abstract syntax, which in the programming languages context is usually referred as *semantic correctness*.

5.2 Verifying the syntactic correctness

In order to verify whether an action a is syntactically correct, there is the need to check whether a conforms to all the WFR that may impact on their syntactic correctness.

Nowadays, existing UML editors do enforce syntactic correctness, checking the compliance of the edited model regarding (a subset of) the WRFs defined in the UML specification. This is the reason why in this thesis we do not focus on the algorithm to check this compliance. Instead, we list the WFRs that must be checked when defining action-based operations, including some corrections and extensions to the rules provided by the UML and fUML standards.

In the following subsections, we review the rules that must be taken into consideration when verifying the syntactic correctness of an action-based operation. For each action type we show a table with the following information:

- First row (*Id*) shows the identifier for the WFR.
- Second row (*description*) describes (textually and formally in OCL) the WFR.
- Third row (*Source*) points out the source of the rule. Some rules directly proceed from UML/fUML metamodels, while others are new rules that we believe they should be added

to these metamodels to be considered during a syntactic analysis. It is also important to note that, in some cases, the formal description provided by the OMG metamodels contains errors. In this chapter, we also correct them.

5.2.1 CreateObjectAction

Table 5.1 shows the WFRs that must be taken in consideration for each *CreateObjectAction* (`<object> := new <cl>`) appearing in an action-based operation.

Roughly, these rules check that: (1) the input classifier `<cl>` is able to be instantiated (rules 1 to 4); (2) the types of `<object>` and `<cl>` are consistent (rule 5); and (3) the multiplicity of the returned object is (1,1) (rule 6).

Figure 5.4 shows an excerpt from the UML metamodel concerning *CreateObjectAction* action to help understanding the OCL expressions mentioned in Table 5.1.

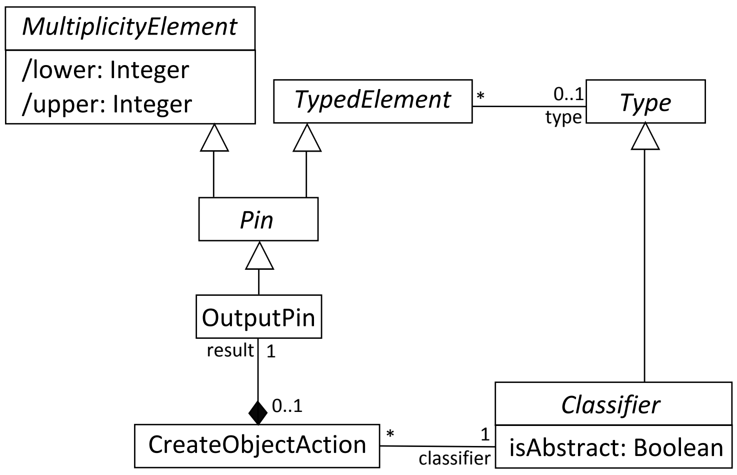


Figure 5.4. Excerpt from the UML metamodel concerning *CreateObjectAction* action.

Table 5.1. Well-Formedness Rules for *CreateObjectAction* action.

Id	*Well-Formedness Rule*		*Source*
1	*Description*	The classifier must be a class.	fUML
	Formalization	**`context`** `CreateObjectAction` **`inv:`** `self.classifier.oclIsKindOf(Class)`	
2	*Description*	The classifier cannot be abstract.	UML
	Formalization	**`context`** `CreateObjectAction` **`inv:`** `not (self.classifier.isAbstract = true)`	
3	*Description*	The classifier cannot be an association class.	UML
	Formalization	**`context`** `CreateObjectAction` **`inv:`** `not (self.classifier.oclIsKindOf (AssociationClass))`	
4	*Description*	The classifier cannot be the supertype of a covering generalization set (in a covering generalization, instances of the supertype cannot be directly created).	Added
	Formalization	**`context`** `CreateObjectAction` **`inv:`** `not (isSupertypeOfAcoveringGeneralization (self.classifier))`, where `isSupertypeOfAcoveringGeneralization(cl)` returns `true` if *cl* is supertype of a covering generalization set.	
5	*Description*	The type of the result pin must be the same as the classifier of the action.	UML
	Formalization	**`context`** `CreateObjectAction` **`inv:`** `self.result.type = self.classifier`	
6	*Description*	The multiplicity of the output pin is 1..1.	UML (OCL corrected)
	Formalization	**`context`** `CreateObjectAction` **`inv:`** `self.result.lower = 1 and self.result.upper = 1`	

5.2.2 DestroyObjectAction

Table 5.2 shows the WFRs that must be taken in consideration for each *DestroyObjectAction* (`<object>.destroy()`) appearing in an action-based operation.

Roughly, these rules check that: (1) the multiplicity of the `<object>` is (1,1) (rule 1); and (2) the `<object>` has no type.

Figure 5.5 shows an excerpt from the UML metamodel concerning *DestroyObjectAction* action to help understanding the OCL expressions mentioned in Table 5.2.

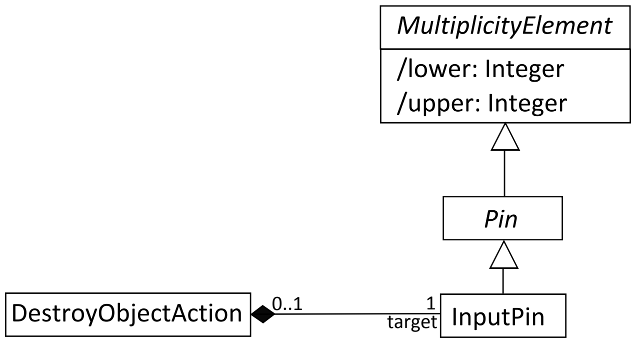


Figure 5.5. Excerpt from the UML metamodel concerning *DestroyObjectAction* action.

Table 5.2. Well-Formedness Rules for *DestroyObjectAction* action.

Id	*Well-Formedness Rule*		*Source*
1	*Description*	The multiplicity of the input pin is 1..1.	UML (OCL corrected)
	Formalization	**`context`** `DestroyObjectAction` **`inv:`** `self.target.lower = 1 and self.target.upper = 1`	
1	*Description*	The input pin has no type.	UML
	Formalization	**`context`** `DestroyObjectAction` **`inv:`** `self.target.type->size() = 0`	

5.2.3 ReclassifyObjectAction

Table 5.3 shows the WFRs that must be taken in consideration for each *ReclassifyObjectAction* (`classify <object> [from <oldCl>] [to <newCl>]`) appearing in an action-based operation.

Roughly, these rules check that: (1) old (`<oldCl>`) and new (`<newCl>`) classifiers are able to be removed/added as an old/new classifiers for the `<object>` (rules 1 to 3); and (2) the `<object>` may be reclassified (rules 4 and 5).

Figure 5.6 shows an excerpt from the UML metamodel concerning *ReclassifyObjectAction* action to help understanding the OCL expressions mentioned in Table 5.3.

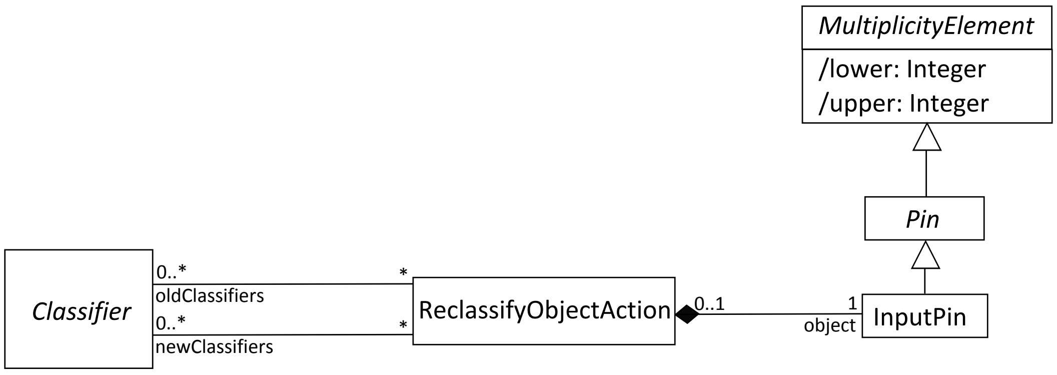


Figure 5.6. Excerpt from the UML metamodel concerning *ReclassifyObjectAction* action.

Table 5.3. Well-Formedness Rules for *ReclassifyObjectAction* action.

Id	*Well-Formedness Rule*		*Source*
1	*Description*	All the old and new classifiers must be classes.	fUML
	Formalization	`context ReclassifyObjectAction inv: self.oldClassifiers->forAll(oclIsKindOf (Class)) and self.newClassifiers->forAll (oclIsKindOf(Class))`	
2	*Description*	None of the new classifiers may be abstract.	UML
	Formalization	`context ReclassifyObjectAction inv: not (self.newClassifiers->exists (isAbstract = true))`	
3	*Description*	None of the new classifiers may be the supertype of a covering generalization set (in a covering generalization, instances of the supertype cannot be directly created).	Added
	Formalization	`context ReclassifyObjectAction inv: not (self.newClassifiers->exists( c | isSupertypeOfAcoveringGeneralization(c) = true)),` where `isSupertypeOfAcoveringGeneralization(cl)` returns `true` if *cl* is supertype of a covering generalization set.	
4	*Description*	The multiplicity of the input pin is 1..1.	UML (OCL corrected)
	Formalization	`context ReclassifyObjectAction inv: self.object.lower = 1 and self.object.upper = 1`	
5	*Description*	The input pin has no type.	UML (OCL corrected)
	Formalization	`context ReclassifyObjectAction inv: self.object.type->size() = 0`	

5.2.4 AddStructuralFeatureValueAction

Table 5.4 shows the WFRs that must be taken in consideration for each *AddStructuralFeatureValueAction* (`<object>.<attribute> = <value>`) appearing in an action-based operation.

Roughly, these rules check the consistency with respect to: (1) the `<value>` (rules 1 to 3); (2) the result `<object>.<attribute>` (rules 4 and 5); (3) the `<object>` (rule 6); and (4) the `<attribute>` (rules 7 to 10).

Figure 5.7 shows an excerpt from the UML metamodel concerning *AddStructuralFeature ValueAction* action to help understanding the OCL expressions mentioned in Table 5.4.

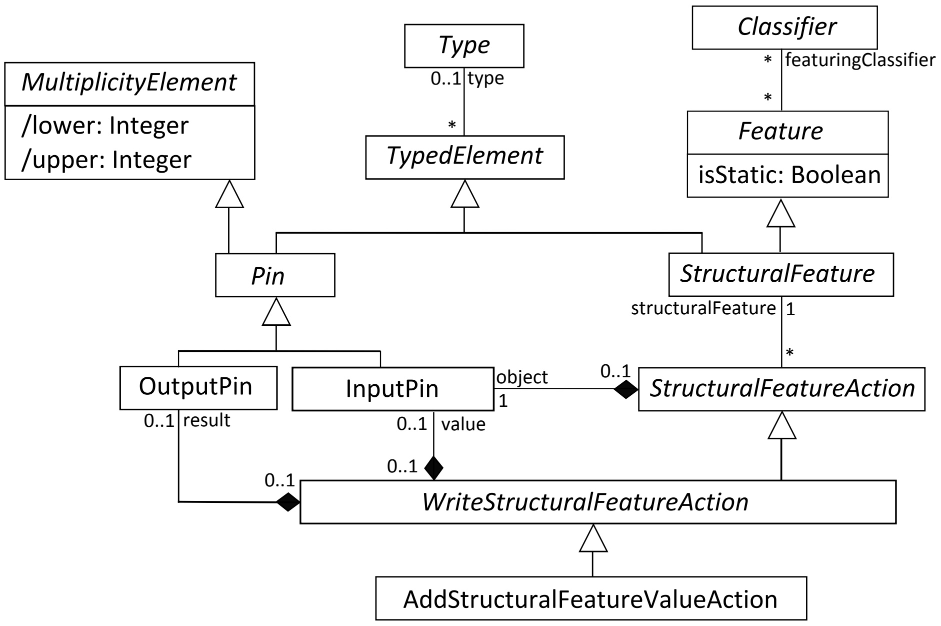


Figure 5.7. Excerpt from the UML metamodel concerning *AddStructuralFeatureValueAction* action.

Table 5.4. Well-Formedness Rules for *AddStructuralFeatureValueAction* action.

Id	*Well-Formedness Rule*		*Source*
1	*Description*	A value input pin is required.	UML
	Formalization	**context** `AddStructuralFeatureValueAction` **inv:** `self.value->notEmpty()`	
2	*Description*	The multiplicity of the value input pin is 1..1.	UML (OCL corrected)
	Formalization	**context** `WriteStructuralFeatureAction` **inv:** `self.value.lower = 1 and self.value.upper = 1`	

Continued on next page

Table 5.4 – continued from previous page

Id	*Well-Formedness Rule*		*Source*
3	*Description*	The type of the value input pin is the same as the type of the structural feature.	UML
	Formalization	**`context`** `WriteStructuralFeatureAction` **`inv:`** `self.value->notEmpty() implies self.value.type = self.structuralFeature.type`	
4	*Description*	The type of the result output pin is the same as the type of the inherited object input pin.	UML
	Formalization	**`context`** `WriteStructuralFeatureAction` **`inv:`** `result->notEmpty() implies self.result.type = self.object.type`	
5	*Description*	The multiplicity of the result output pin must be 1..1.	UML (OCL cor- rected)
	Formalization	**`context`** `WriteStructuralFeatureAction` **`inv:`** `result->notEmpty() implies self.result.lower = 1 and self.result.upper = 1`	
6	*Description*	The multiplicity of the object input pin must be 1..1.	UML (OCL cor- rected)
	Formalization	**`context`** `StructuralFeatureAction` **`inv:`** `self.object.lower = 1 and self.object.upper = 1`	
7	*Description*	The structural feature must not be static.	UML
	Formalization	**`context`** `StructuralFeatureAction` **`inv:`** `self.structuralFeature.isStatic = false`	
8	*Description*	The structural feature must either be owned by the type of the object input pin, or it must be an owned end of a binary association with the type of the opposite end being the type of the object input pin.	UML
	Formalization	**`context`** `StructuralFeatureAction` **`inv:`** `self.structuralFeature. featuringClassifier.oclAsType(Type)-> includes(self.object.type) or self.structuralFeature.oclAsType (Property).opposite.type = self.object.type`	
9	*Description*	The visibility of the structural feature must allow access to the object performing the action.	UML

Continued on next page

Table 5.4 – continued from previous page

Id	*Well-Formedness Rule*		*Source*
	Formalization	**context** StructuralFeatureAction **inv:** let host : Classifier = self.context in self.structuralFeature.visibility = public or host = self.structuralFeature. featuringClassifier.type or (self.structuralFeature.visibility = protected and host.allSupertypes -> includes(self.structuralFeature. featuringClassifier.type)))	
10	*Description*	The structural feature has exactly one *featuringClassifier*.	UML
	Formalization	self.structuralFeature.featuringClassifier ->size() = 1	

5.2.5 ClearStructuralFeatureAction

Table 5.5 shows the WFRs that must be taken in consideration for each *ClearStructuralFeatureAction* (`<object>.<attribute> = null`) appearing in an action-based operation.

Roughly, these rules check the consistency with respect to: (1) the result of the action (`<object>.<attribute>`) (rules 1 and 2); (2) the `<object>` (rule 6 of Table 5.4); and (3) the `<attribute>` (rules 7 to 10 of Table 5.4).

Figure 5.8 shows an excerpt from the UML metamodel concerning *ClearStructuralFeatureAction* action to help understanding the OCL expressions mentioned in Table 5.5.

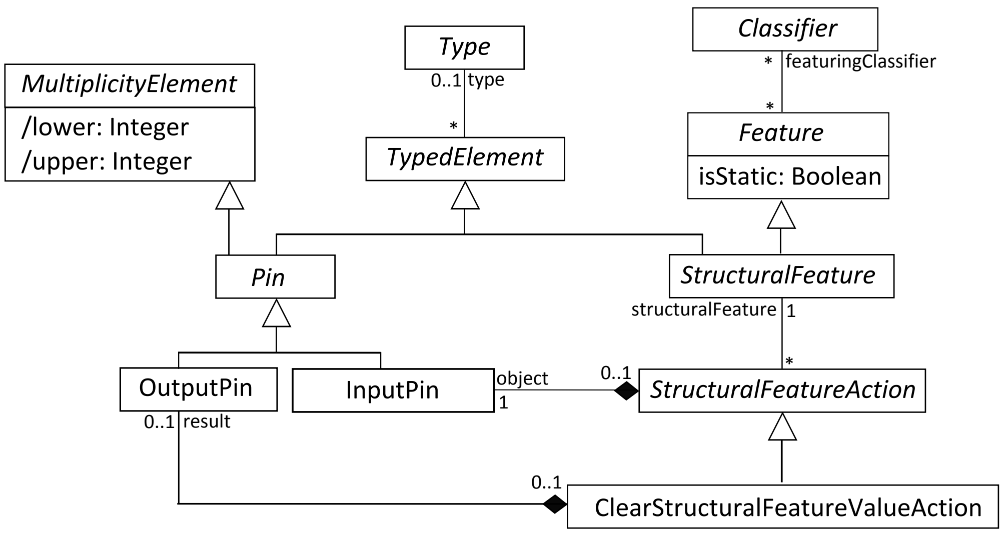


Figure 5.8. Excerpt from the UML metamodel concerning *ClearStructuralFeatureAction* action.

Table 5.5. Well-Formedness Rules for *ClearStructuralFeatureAction* action.

Id	*Well-Formedness Rule*		*Source*
1	*Description*	The type of the result output pin is the same as the type of the inherited object input pin.	UML
	Formalization	`context ClearStructuralFeatureAction inv: result->notEmpty() implies self.result.type = self.object.type`	
2	*Description*	The multiplicity of the result output pin must be 1..1.	UML (OCL corrected)
	Formalization	`context ClearStructuralFeatureAction inv: result->notEmpty() implies self.result.lower = 1 and self.result.lower = 1`	
See WFRs 6 to 10 of Table 5.4.			

5.2.6 CreateLinkAction

Table 5.6 shows the WFRs that must be taken in consideration for each *CreateLinkAction* (`<association>.createLink([<role1>=>]<obj1>,[<role2>=>]<obj2>)`) appearing in an action-based operation.

Roughly, these rules check the consistency with respect to: (1) the `<association>` (rule 1), (2) the end data (`<role1>` and `<role2>`) (rules 2 to 6); and (3) the input pins (`<obj1>` and `<obj2>`) (rules 7 and 8).

Figure 5.9 shows an excerpt from the UML metamodel concerning *CreateLinkAction* action to help understanding the OCL expressions mentioned in Table 5.6.

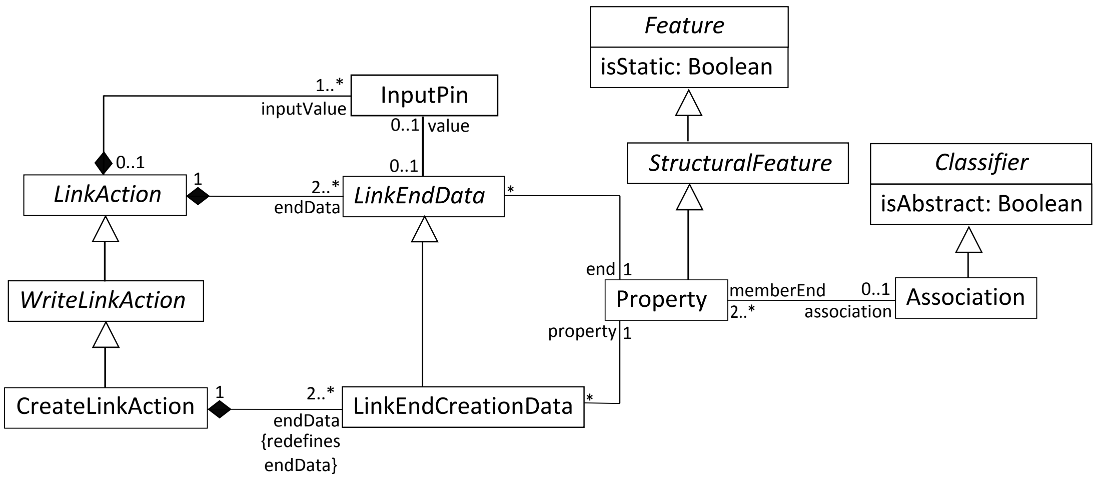


Figure 5.9. Excerpt from the UML metamodel concerning *CreateLinkAction* action.

Table 5.6. Well-Formedness Rules for *CreateLinkAction* action.

Id	*Well-Formedness Rule*		*Source*
1	*Description*	The association cannot be an abstract classifier.	UML
	Formalization	**context** `CreateLinkAction` **inv:** `not (self.association().isAbstract = true)` where `association()` returns the association of the action: `association = self.endData->asSequence().first().end.association`	
2	*Description*	All end data must have exactly one input object pin.	UML (OCL corrected)
	Formalization	**context** `WriteLinkAction` **inv:** `self.endData->forAll(value->size() = 1)`	
3	*Description*	All end data must belong to exactly association.	Added
	Formalization	**context** `WriteLinkAction` **inv:** `self.endData->forAll(end.association->size() = 1)`	
4	*Description*	The visibility of at least one end must allow access to the class using the action.	UML
5	*Description*	The association ends of the link end data must all be from the same association and include all and only the association ends of that association.	UML
	Formalization	**context** `LinkAction` **inv:** `self.endData->collect(end) = self.association()->collect(memberEnd)`	(OCL corrected)
6	*Description*	The association ends of the link end data must not be static.	UML
	Formalization	**context** `LinkAction` **inv:** `self.endData->forAll(end.oclisKindOf (NavigableEnd) implies end.isStatic = false)`	
7	*Description*	The action must have at least two input pins.	Added
	Formalization	**context** `LinkAction` **inv:** `self.inputValue-> size() >= 2`	
8	*Description*	The input pins of the action are the same as the pins of the link end data and insertion pins.	UML
	Formalization	**context** `LinkAction` **inv:** `self.inputValue-> asSet() = let ledpins : Set = self.endData-> collect(value) in if self.oclIsKindOf(LinkEndCreationData) then ledpins->union(self.endData.oclAsType (LinkEndCreationData).insertAt) else ledpins`	

5.2.7 DestroyLinkAction

Table 5.7 shows the WFRs that must be taken in consideration for each *DestroyLinkAction* (`<association>.destroyLink([<role1>=>]<obj>,[<role2>=>]<obj2>)`) appearing in an action-based operation.

Roughly, these rules check the consistency over: (1) the end data (`<role1>` and `<role2>`) (rules 2 to 6 of Table 5.6); and (2) the input pins (`<obj1>` and `<obj2>`) (rules 7 and 8 of Table 5.6).

Figure 5.10 shows an excerpt from the UML metamodel concerning *DestroyLinkAction* action to help understanding the OCL expressions mentioned in Table 5.7.

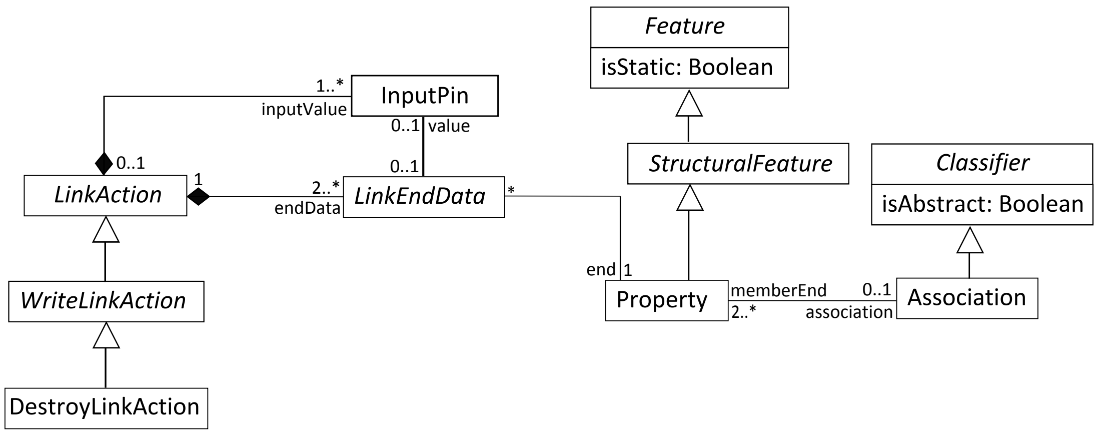


Figure 5.10. Excerpt from the UML metamodel concerning *DestroyLinkAction* action.

Table 5.7. Well-Formedness Rules for *DestroyLinkAction* action.

Id	*Well-Formedness Rule*	*Source*
Same WFRs 2 to 8 of Table 5.6.		

5.2.8 ClearAssociationAction

Table 5.8 shows the WFRs that must be taken in consideration for each *ClearAssociationAction* (`<association>.clearAssoc(<object>)`) appearing in an action-based operation.

Roughly, these rules check that: (1) the type of the `<object>` is the same as the type of one association ends of the `<association>` (rule 1); and (2) the multiplicity of the `<object>` is (1,1) (rule 2).

Figure 5.11 shows an excerpt from the UML metamodel concerning *ClearAssociationAction* action to help understanding the OCL expressions mentioned in Table 5.8.

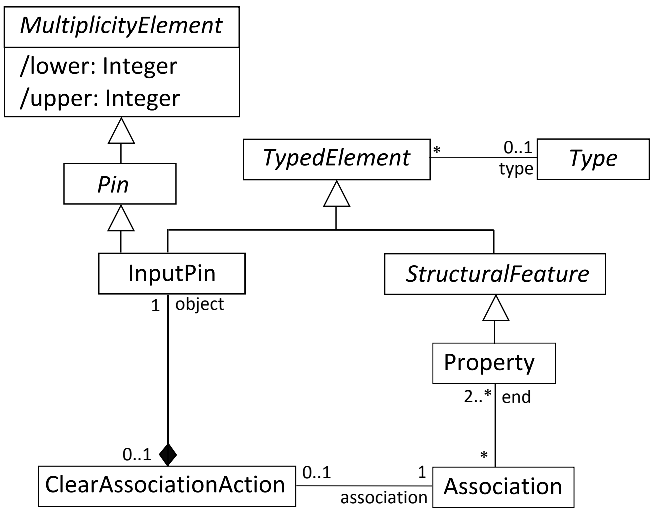


Figure 5.11. Excerpt from the UML metamodel concerning *ClearAssociationAction* action.

Table 5.8. Well-Formedness Rules for *ClearAssociationAction* action.

Id	*Well-Formedness Rule*		*Source*
1	*Description*	The type of the input pin must be the same as the type of at least one of the association ends of the association.	UML
	Formalization	`**context** ClearAssociationAction **inv:** self.association->exists(end.type = self.object.type)`	
2	*Description*	The multiplicity of the input pin is 1..1.	UML (OCL corrected)
	Formalization	`**context** ClearAssociationAction **inv:** self.object.lower = 1 and self.object.upper = 1`	

5.2.9 CallOperationAction

Table 5.9 shows the WFRs that must be taken in consideration for each *CallOperationAction* (`[<result>]=<object>.<operation>([<arguments>])`) appearing in an action-based operation.

Roughly, these rules check that: (1) the arguments of the invocation are consistent with the parameters of the operation (rules 1 to 3); and (2) the type of the `<object>` is consistent with the class that holds the `<operation>` (rule 4).

Figure 5.12 shows an excerpt from the UML metamodel concerning *CallOperationAction* action to help understanding the OCL expressions mentioned in Table 5.9.

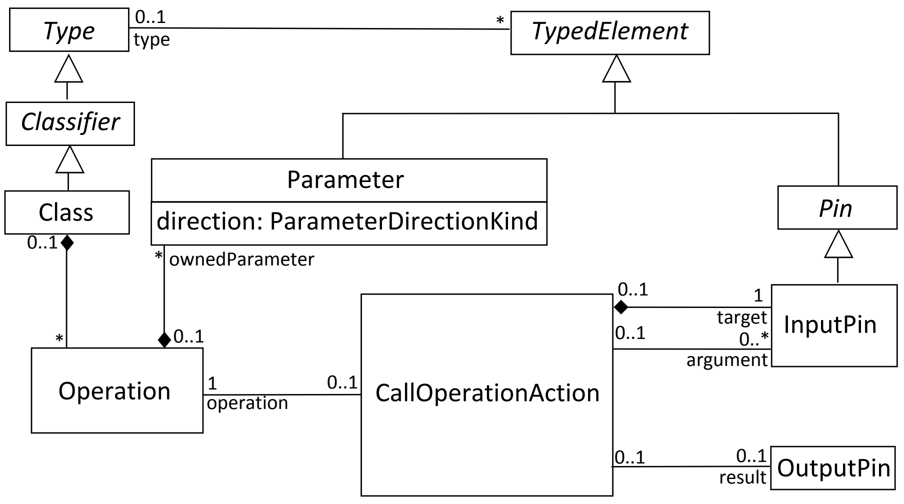


Figure 5.12. Excerpt from the UML metamodel concerning *CallOperationAction* action.

Table 5.9. Well-Formedness Rules for *CallOperationAction* action.

Id	*Well-Formedness Rule*		*Source*
1	*Description*	The number of argument pins and the number of owned parameters of the operation of type in and inout must be equal.	fUML
	Formalization	**`context`** `CallOperationAction` **`inv:`** `self.argument->size() = self.operation.ownedParameter-> select(p | p.direction = in or p.direction = inout)->size()`	(OCL added)
2	*Description*	The number of result pins and the number of owned parameters of the operation of type return, out, and inout must be equal.	fUML
	Formalization	**`context`** `CallOperationAction` **`inv:`** `self.result->size() = self.operation.ownedParameter-> select(p | p.direction = return or p.direction = out OR p.direction = inout)->size()`	(OCL added)
3	*Description*	The type, ordering, and multiplicity of an argument or result pin is derived from the corresponding owned parameter of the operation.	fUML
4	*Description*	The type of the target pin must be the same as the type that owns the operation.	fUML
	Formalization	**`context`** `CallOperationAction` **`inv:`** `self.target.type = self.operation.class`	(OCL added)

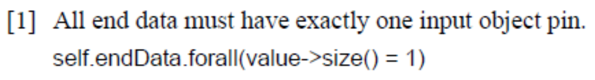
Then, in order to verify whether an action-based operation is syntactically correct, we suggest using an existing modelling tool able to check the compliance of the model regarding the above reviewed WFRs.

5.2.10 Flaws and lacks in UML/fUML metamodels

In order to conclude this chapter, we would like to remark that our analysis of the Action packages of UML (v2.4.1) [126] and fUML (v1.0) [125] metamodels has detected several lacks and flaws that compromise the correctness of these specifications. Most of them has been fixed in the tables of the previous section. In the following we summarize these deficits and suggest possible corrections for them:

Flaws

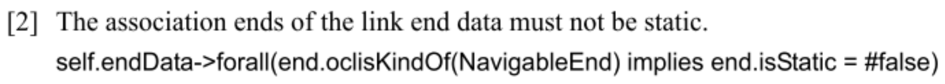
1. UML 2.4.1 contains references to "*forall*" OCL operation (see Figure 5.13) instead of "*forAll*". In some cases, the use of this operation is incorrect (they invoke it using ".forAll" instead of "->forAll"). It also contains references to "*oclisKindOf*" (see Figure 5.14) OCL operation instead of "*oclIsKindOf*".



[1] All end data must have exactly one input object pin.

```
self.endData.forall(value->size() = 1)
```

Figure 5.13. Excerpt from the UML metamodel concerning *WriteLinkAction* metaclass description (page 318 of [126]).

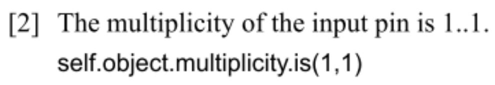


[2] The association ends of the link end data must not be static.

```
self.endData->forall(end.oclisKindOf(NavigableEnd) implies end.isStatic = #false)
```

Figure 5.14. Excerpt from the UML metamodel concerning *LinkAction* metaclass description (page 282 of [126]).

2. In UML 2.4.1, constraints restricting the multiplicity of input and output pins refer to a *multiplicity* attribute (see Figure 5.15) that does not longer exist in this version of the UML metamodel (it exists in the version 1.5 of the UML metamodel). From UML 2.0, pins are subtypes of *MultiplicityElement* and thus we should use the *upper* and *lower* attributes instead.

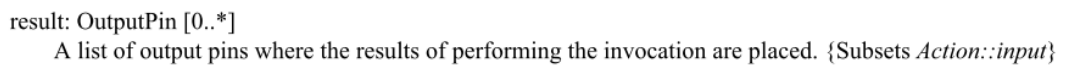


[2] The multiplicity of the input pin is 1..1.

```
self.object.multiplicity.is(1,1)
```

Figure 5.15. Excerpt from the UML metamodel concerning *ClearAssociationAction* metaclass description (page 272 of [126]).

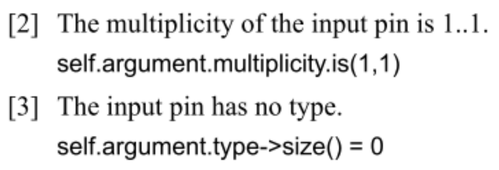
3. UML 2.4.1 defines the *result* member end from *CallAction* metaclass as a subset of *Action* :: *input* association (see Figure 5.16). It should be a subset of *Action* :: *output* association.



result: OutputPin [0..*]
A list of output pins where the results of performing the invocation are placed. {Subsets *Action::input*}

Figure 5.16. Excerpt from the UML metamodel concerning *CallAction* metaclass description (page 268 of [126]).

4. UML 2.4.1 references the *argument* attribute in the constraints of the *ReclassifyObjectAction* metaclass (see Figure 5.17). This attribute does not exist in this version of the metamodel. Instead, the attribute *object* should be used.

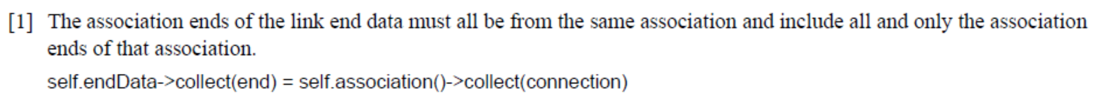


[2] The multiplicity of the input pin is 1..1.
self.argument.multiplicity.is(1,1)

[3] The input pin has no type.
self.argument.type->size() = 0

Figure 5.17. Excerpt from the UML metamodel concerning *ReclassifyObjectAction* metaclass description (page 301 of [126]).

5. UML 2.4.1 references the *connection* member end in the constraints of the *LinkAction* metaclass (see Figure 5.18). This member end does not exist in this version of the metamodel. Instead, the member end *memberEnd* should be used.



[1] The association ends of the link end data must all be from the same association and include all and only the association ends of that association.
self.endData->collect(end) = self.association()->collect(connection)

Figure 5.18. Excerpt from the UML metamodel concerning *LinkAction* metaclass description (page 282 of [126]).

Lacks

1. In UML 2.4.1, it is not clear the relationship between the *InstanceSpecification*, *ValueSpecification* and *Pin* metaclasses. Since input and output pins must hold *InstanceSpecification* (e.g. in the *CreateObjectAction* action) and *ValueSpecification* (e.g. *WriteStructuralFeature* actions) values, both kind of values need to be converted to instances of the *Pin* metaclass which is not possible with the current metamodel structure.

2. In fUML 1.0, the semantic of attributes neither associations is not specified. Besides, as we have mentioned, this specification does not include most of the constraints included

in the UML 2.4.1 metamodel. We assume fUML models must satisfy both the UML and fUML WFRs.

3. The fUML specification is not complete. For instance, regarding the *CreateObjectAction* action, fUML does not specify whether attributes or links are initialized when creating an object. Fortunately, Alf specification clarifies this lack of precision.

5.3 Summary

In this chapter we have reviewed the syntactic correctness property of action-based operations.

We consider an action-based operation is **syntactically correct** when all the actions included in the operation satisfy the Well-Formedness Rules defined in the fUML/UML metamodels.

In this chapter we have reviewed and corrected the WFR proposed in the UML/fUML metamodels. Besides, we have added new rules that we believe they should be considered when verifying the syntactic correctness of an action-based operation.

The execution of any thing considerable implies in the first place previous persevering meditation.

William Godwin

6 Executability

The aim of this chapter is to precisely define the notion of *executability* regarding two levels of correctness (*weak* and *strong*) and to describe a lightweight and static method we propose to check whether an Alf-based operation satisfies this property. In order to define the notion of *executability*, we need to previously introduce the notion of *execution paths* of an operation, which refers to all the possible sequences of actions that may be followed during the operation execution.

This chapter is divided into five sections: Section 6.1 precisely defines the notion of *execution path* and explains how the execution paths of an operation may be computed; Section 6.2 precisely defines the executability properties we address in this chapter; Section 6.3 describes the method we propose to verify the executability of an action-based operation; then, since executability is the more complex property and the more deeply studied along this thesis, Section 6.4 discusses about the pros and cons of our method; and finally, Section 6.5 summarizes and concludes the chapter.

6.1 Execution Paths

The execution of an action-based operation may be described in terms of the sequence of changes the operation applies on the system state. The correctness property addressed in this chapter is based on an analysis of the possible *execution paths* allowed by the actions that define the operation effect, that is, the possible sequences of actions that may be followed during the operation execution.

In the next subsections we describe a way where operations can be represented in order to obtain its execution paths (Section 6.1.1); and how execution paths may be computed from this previous representation (Section 6.1.2).

6.1.1 Constructing the Model-Based Control Flow Graph

In order to determine the possible execution paths of an operation (i.e. the sequences of changes it may induce), we propose to draw each operation as a *Model-Based Control Flow Graph* (MBCFG). A MBCFG is a directed graph which extrapolates the traditional ideas of *Control Flow Graph* (CFG)[3], a directed digraph which represents the program code, to represent the information of a model (in our case, operations as part of an executable behavioural model).

Formally, a *directed graph* (or *digraph*) [20] $G = (V, A)$ consists of a finite set V of *vertices* (or *nodes*) and a set $A \subseteq V \times V$ of *arcs*. An arc (v_1, v_2) has source v_1 and target v_2 and is said to go from vertex v_1 to vertex v_2. Note that if $v_1 = v_2$ it is a $self-loop$.

In order to represent Alf operations as MBCFGs, we consider that each operation is an instance of the *Activity* metaclass from fUML (see the fUML metamodel excerpt in Figure 6.1). An *Activity* contains several $ActivityNode$s ($ActivityNode$ generalizes the metaclass $ExecutableNode$, which in turn generalizes the metaclass *Action*). Then, each action can be either one of the actions from the Actions package (see an excerpt of them at Section 2.2.2), a $ConditionalNode$ (which represents an exclusive choice among some number of alternatives) or a $LoopNode$ (which represents a loop with setup, test, and body sections). We also use two $fake$ nodes: an *initial node* (representing the first instruction in the operation) and a *final node* (representing the last one). These two nodes do not change the operation effect but help in simplifying the presentation of our MBCFG.

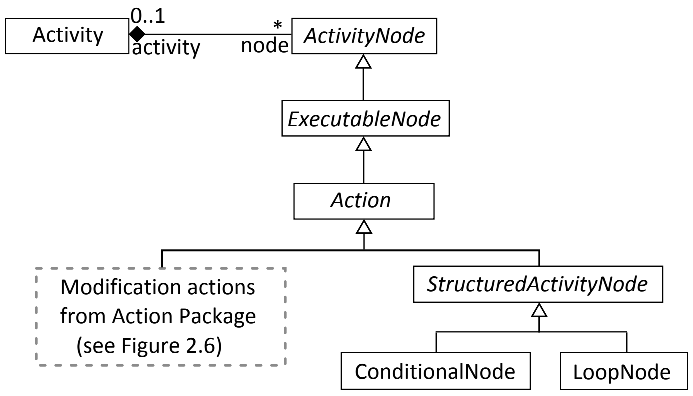


Figure 6.1. Fragment of the fUML metamodel Actions Package.

A Model-Based Control Flow Graph (MBCFG) for an operation *op* is a 2-tuple (V_{op}, A_{op}). The corresponding vertices (V_{op}) and arcs (A_{op}) are obtained applying the following rules:

- Every activity node (i.e. action) in *op* is a vertex in V_{op}. In order to simplify the MBCFG, we only consider actions that may modify the system state (i.e. modification actions) and

[3]As we introduced in Chapter 3, CFGs are used on Data-Flow Analysis [120].

structured actions (i.e. conditionals or loops). It means that we skip other types of actions (as actions to read values or to declare and initialize variables) since they do not affect the result of our analysis.

- An arc from a vertex v_1 to v_2 is created in A_{op} if v_1 immediately precedes v_2 in an ordered sequence of nodes.
- A vertex v representing a conditional node n is linked by an arc to the vertices $v_1, \ldots, v_n$ representing the first activity node for each *clause* (i.e. the *then* clause, the *else* clause, ...) in n. All vertices of each clause are englobed into a dashed line box. The last vertex in each clause is linked to the vertex v_{next} immediately following n in the sequence of executable activities. If n does not include an *else* clause, an arc between v and v_{next} is also added to A_{op}.
- Each arc from a conditional node to its first clause vertex is labelled with the condition of the conditional structure. Each arc to an *else* clause (or the arc between the conditional node and the the v_{next} if there is not an *else* clause) is labelled with the negation of the above condition.
- A vertex v representing a loop node n, is linked by an arc to the vertex representing the first activity node for *n.bodyPart* (returning the list of actions in the body of the loop) and the vertex v_{next} immediately following n in the activity. The last vertex in *n.bodyPart* is linked back to v (to represent the loop behaviour).
- Each arc from a loop node to its first vertex is labelled with the condition fulfilled in the first execution of the loop followed by the times the loop is executed.
- A vertex representing an *OperationCall* action is replaced by the sub-digraph corresponding to the called operation op' as follows: (1) the initial vertex of op' is connected with the vertex that precedes the *OperationCall* activity node in the main operation; (2) the final vertex of op' is connected with the vertex/ces that follow the *OperationCall*; and (3) the parameters of op' are replaced by the arguments in the call.

During the construction of our MBCFG we assume several facts. In the following we explain and justify these assumptions:

- We assume the body of all conditional and loop structures is reachable (given the proper input values). This means that the condition of all conditional and loop structures may be satisfied (i.e. it may evaluate to true) and then the body of these structures may be executed. This assumption is made because of, as we will see in the next sections, our analysis takes into account all the actions which are part of an execution path. On the other hand, this assumption is based on the accepted criteria that unreachable code should be eliminated [16, 42].
- We assume that operations to be analyzed do not include recursive invocations (i.e. invocations to herself). This assumption is made because of recursive invocations generate infinite paths (given that the recursive invocation is replaced by the sub-diagraph corresponding to the operation itself, then, this replacement process never finishes) that are not able to be addressed by the techniques proposed in this thesis. Nevertheless, recursive operations could be transformed into their iterative counterparts before the application of our techniques [9]. On the other hand, iterative invocations always generate finite paths since the body of a loop contains a finite number of actions and then it always generate a finite set of nodes.

Example 11 Consider the excerpt of the class diagram shown in Figure 6.2 and the following operations: `addCourse` (in the context of class `Course`) which adds a new course to the system); `setCourseDescription` (in the context of class `Course`) which modifies the description of a course; `addSubstituteCourse` (in the context of class `Course`) which adds a substituting course to the self course; and `addDessertsMenu` (in the context of class `Menu`) which adds a menu composed only of desserts. Suppose operation `checkCategory(c:Course)` is already defined. To simplify the graphs, we do not show the body of this operation.

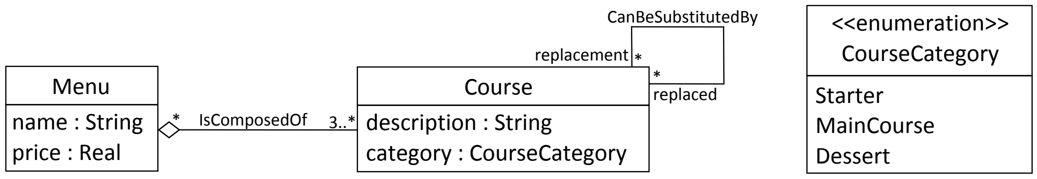


Figure 6.2. Excerpt of the structural model.

```
activity addCourse(in _description:  String, in _category:
CourseCategory, in _substitutingCourses:  Course[*]) {
  Course c = new Course();
  c.description = _description;
  c.category = _category;
  for ( i in 1.._substitutingCourses→size() ) {
   CanBeSubstitutedBy.createLink(replaced=>c,
   replacement=>_substitutingCourses[i]);
  }
}
```

```
activity setCourseDescription(in _newDescription:  String) {
  self.description = _newDescription;
}
```

```
activity addSubstitutedCourse(in _course:  Course) {
  if (self.category == _course.category) {
   CanBeSubstitutedBy.createLink(replaced=>self,
   replacement=>_course;
  }
}
```

```
activity addDessertsMenu(in _name:  String, in _price:  Real, in
_courses:  Course[*]) {
  Menu m = new Menu();
  m.name = _name;
  m.price = _price;
  for ( i in 1.._courses->size() ) {
   if ( _courses[i].category == Dessert )
    IsComposedOf.createLink(menu=>m,course=>_courses[i]);
   else checkCategory(_courses[i]);
  }
}
```

Figures 6.3 to 6.6 show the MBCFGs for the operations `addCourse`, `setCourseDescription`, `addSubstitutedCourse` and `addDessertsMenu` respectively.

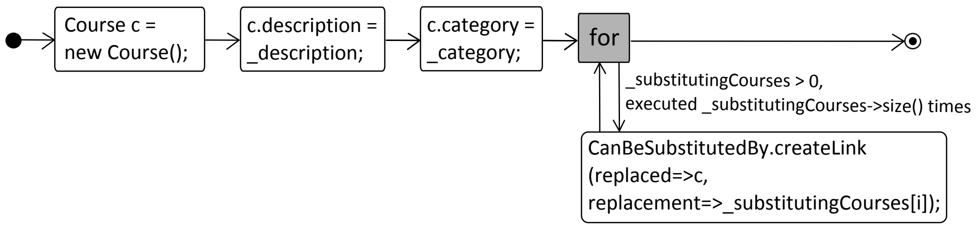

Course c =
new Course();
c.description =
_description;
c.category =
_category;
for
_substitutingCourses > 0,
executed _substitutingCourses->size() times
CanBeSubstitutedBy.createLink
(replaced=>c,
replacement=>_substitutingCourses[i]);

Figure 6.3. MBCFG of `addCourse` operation.

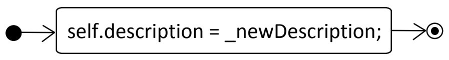


Figure 6.4. MBCFG of `setCourseDescription` operation.

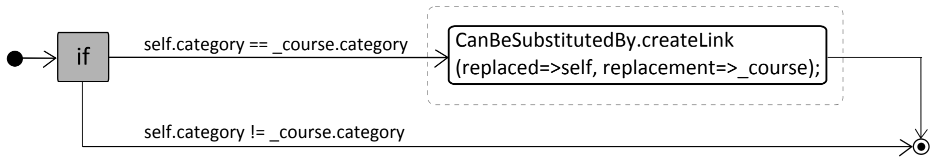

if
self.category == _course.category
CanBeSubstitutedBy.createLink
(replaced=>self, replacement=>_course);
self.category != _course.category

Figure 6.5. MBCFG of `addSubstitutedCourse` operation.

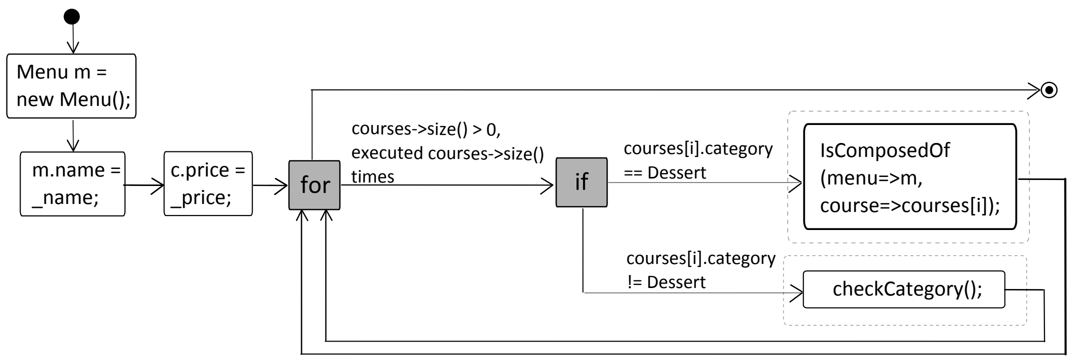


Figure 6.6. MBCFG of `addDessertsMenu` operation.

6.1.2 Computing the Execution Paths from the MBCFG

An **execution path** of an operation *op* is a finite and not empty (we discard the possible empty paths) sequence of actions starting at the initial vertex and ending at the final vertex in the $MBCFG$, that is, a sequence of actions that may be followed during the operation execution. For trivial operations (e.g. with neither conditional nor loop nodes) there is a single execution path but, in general, several ones will exist.

More formally, an execution path is a sequence of arcs (v_{ini}, v_1), (v_1, v_2), ..., (v_{i-1}, v_i), (v_i, v_{end}) where the target of the arc (v_{i-1}, v_i) equals the source of the arc (v_i, v_{i+1}).

We represent an execution path p as a set of *terms* and *guards* {*terms*, *guards*}, where:

$$terms = \{\underbrace{< t_1, multiplicity_1, action_1 >}_{term_1}, \ldots, \underbrace{< t_n, multiplicity_n, action_n >}_{term_n}\}$$

- t_i is a unique identifier for the term and specifies the position of the $action_i$ inside the path.
- $action_i$ is the ith-action of the path.
- $multiplicity_i$ indicates the number of times that $action_i$ is executed. The $multiplicity_i$ of the term is equal to "1" when $action_i$ is not included in any loop structure. For actions within loops, *multiplicity* is either the number of times the loop is executed (if we can deduce the concrete number of iterations) or simply an abstract variable ("N"[4]) otherwise.

[4]If an action is included into two loops, its multiplicity will be "N_1*N_2" and so forth.

$$guards = \{\underbrace{< g_1, condition_1, t_{ini_1}, t_{end_1} >}_{guard_1}, \ldots, \underbrace{< g_m, condition_m, t_{ini_m}, t_{end_m} >}_{guard_m}\}$$

- g_i is an unique identifier of the guard.
- $condition_i$ is a boolean expression over the parameters and variables of the path that evaluates to true immediately before the term t_{ini_i} is executed. The $condition_i$ is the condition of the conditional or loop structure (or its negation, when dealing with *else* branches).
- t_{ini_i} and t_{end_i} ($t_{ini_i} \leq t_{end_i}$) identify the first and the last terms englobed by the $condition_i$.

Given a $MBCFG_{op}$ graph for an operation *op*, the set of execution paths ($Paths(op)$) for *op* is defined as $Paths(op) = allPaths(MBCFG_{op})$, where $allPaths(MBCFG_{op})$ returns the set of all paths in $MBCFG_{op}$ that start at the initial vertex (the vertex corresponding to the initial node), end at the final vertex and do not include repeated arcs.

For those MBCFGs in which appear one or several conditional structures within a loop, all the branches of the conditional are concatenated in the same execution path, each with a multiplicity equal or lower to the loop multiplicity (see Example 15). This performance allows to take into account that some iterations of an execution may follow one branch of the conditional while others may follow another branch.

To illustrate the algorithm for computing the execution paths given a MBCFG, in the following we show the execution paths of the operations introduced in the previous subsection.

Example 12 Operation `addCourse` has two execution paths (see Figure 6.7): $p1_{addCourse}$ is the sequence of actions executed when the `_substitutingCourses` array does not contain any course; and $p2_{addCourse}$ is the sequence of actions executed otherwise. Note that the first path has three terms (t_1, t_2 and t_3) but it does not have any guard. On the other hand, the second path has four terms and one guard (g_1).

Execution paths for operation `addCourse`:
$p1_{addCourse} = \{$
{ <t_1, 1, `Course c = new Course()`>,
<t_2, 1, `c.description = _description`>,
<t_3, 1, `c.category = _category`>}, {} }
$p2_{addCourse} = \{$
{ <t_1, 1, `Course c = new Course()`>,
<t_2, 1, `c.description = _description`>,
<t_3, 1, `c.category = _category`>,
<t_4, `_substitutingCourses→size()`,`CanBeSubstitutedBy.createLink(replaced=>c,replacement=>_substitutingCourses[i])`> },
{<g_1, `_substitutingCourses→size()>0`, t_4, t_4>} }

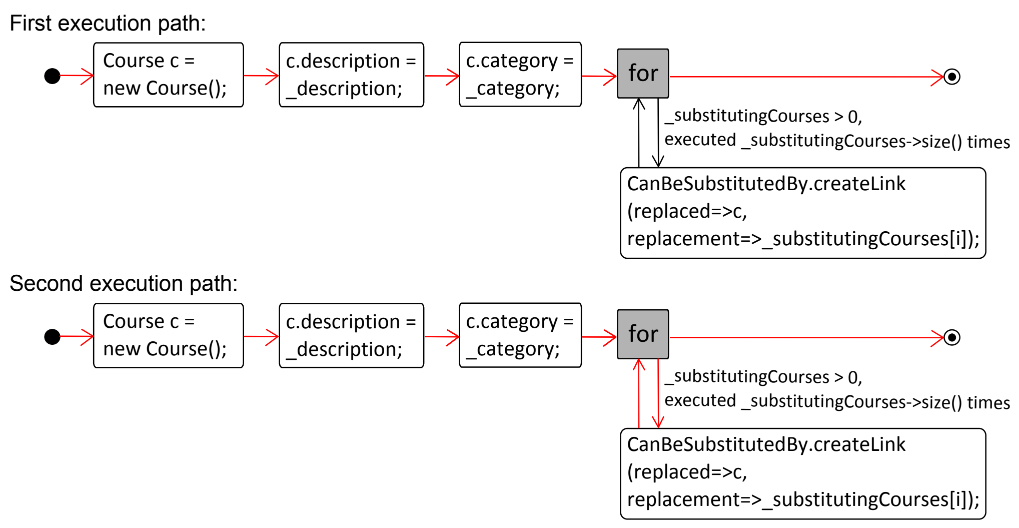

First execution path:
Course c = new Course();
c.description = _description;
c.category = _category;
for
_substitutingCourses > 0, executed _substitutingCourses->size() times
CanBeSubstitutedBy.createLink (replaced=>c, replacement=>_substitutingCourses[i]);
Second execution path:
Course c = new Course();
c.description = _description;
c.category = _category;
for
_substitutingCourses > 0, executed _substitutingCourses->size() times
CanBeSubstitutedBy.createLink (replaced=>c, replacement=>_substitutingCourses[i]);

Figure 6.7. Execution paths for the `addMenu` operation.

Example 13 Operation `setCourseDescription` has a single execution path (see Figure 6.8).

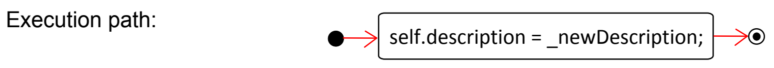


Figure 6.8. Execution paths for the `setCourseDescription` operation.

Execution path for operation `setCourseDescription`:
$p_{setCourseDescription} = \{$
$\{$ <t_1, 1, `self.description = _newDescription`>$\}$, $\{\}$ $\}$

Example 14 Operation `addSubstitutedCourse` has also a single execution path (see Figure 6.9), (note that we discard the empty path): $p_{addSubstitutedCourse}$ is the sequence of actions executed when `self.category == _course.category` (see the guard g_1 from $p_{addSubstitutedCourse}$).

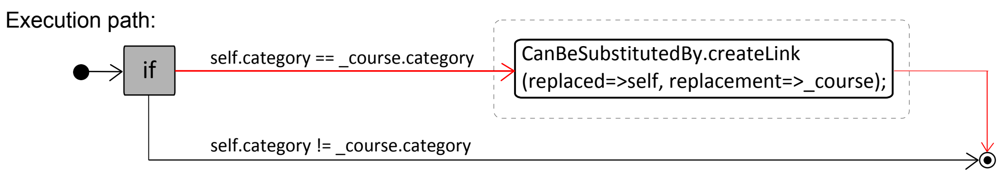
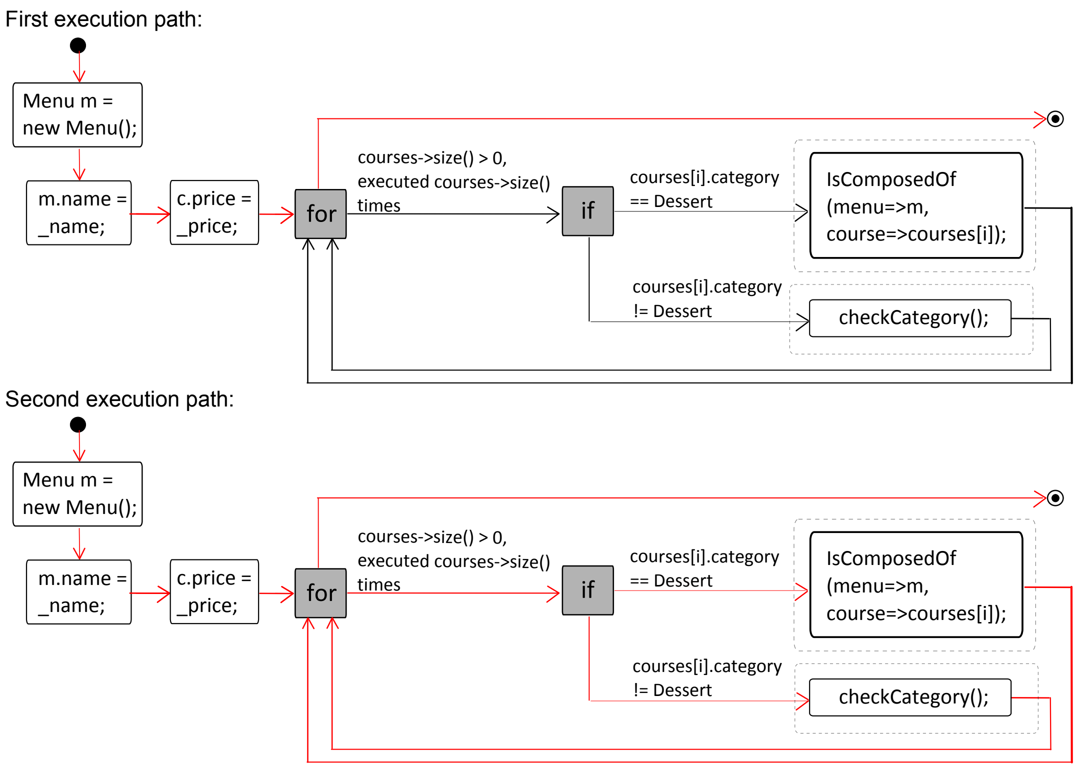

Execution path:
if
self.category == _course.category
CanBeSubstitutedBy.createLink (replaced=>self, replacement=>_course);
self.category != _course.category

Figure 6.9. Execution paths for the `addSubstitutedCourse` operation.

Execution path for operation `addSubstitutedCourse`:
$p_{addSubstitutedCourse} = \{$
  { <t_1, 1, `CanBeSubstitutedBy.createLink(replaced=>c,replacement=>_substitutingCourses[i])`> },
 {<g_1, `self.category == _course.category`, t_1, t_1>} }

Example 15 Finally, operation `addDessertsMenu` has two execution paths (see Figure 6.10), (note that we discard the empty path): $p1_{addDessertsMenu}$ is the sequence of actions executed when `_courses->size()==0`; and $p2_{addDessertsMenu}$ is the sequence of actions executed otherwise (see the guard g_1 from $p2_{addDessertsMenu}$). Note that both branches of the conditional structure which appear inside the loop are concatenated in the same path (see $p2_{addDessertsMenu}$) with multiplicity lower or equal to the total multiplicity of the loop. This allows to take into account that some iterations of the loop may follow the *if* branch while others may follow the *else* branch.



First execution path:
Menu m = new Menu();
m.name = _name;
c.price = _price;
for
courses->size() > 0, executed courses->size() times
if
courses[i].category == Dessert
IsComposedOf (menu=>m, course=>courses[i]);
courses[i].category != Dessert
checkCategory();
Second execution path:
Menu m = new Menu();
m.name = _name;
c.price = _price;
for
courses->size() > 0, executed courses->size() times
if
courses[i].category == Dessert
IsComposedOf (menu=>m, course=>courses[i]);
courses[i].category != Dessert
checkCategory();

Figure 6.10. Execution paths for the `addDessertsMenu` operation.

Execution paths for operation `addDessertsMenu`:

$p1_{addDessertsMenu}$ = {
{ <t_1, 1, `Menu m = new Menu()`>,
<t_2, 1, `m.name = _name`>,
<t_3, 1, `m.price = _price`> } , { } }

$p2_{addDessertsMenu}$ = {
{ <t_1, 1, `Menu m = new Menu()`>,
<t_2, 1, `m.name = _name`>,
<t_3, 1, `m.price = _price`>,
<t_4, ≤`_courses->size()`, `IsComposedOf.createLink(menu=>m,course=> _courses[i])`>,
<t_5, ≤`_courses->size()`, `checkCategory(_courses[i])`> },
{ <g_1, `_courses->size()>0`, t_4, t_5>,
<g_2, `_courses[i].category==Dessert`, t_4, t_4>,
<g_3, `_courses[i].category!=Dessert`, t_5, t_5> } }

6.2 Executability Definition

Prior to define the executability property, we need to describe the meaning of *execution* of an execution path of an operation. The execution of an execution path p over a system state s, generates a new state s' where the changes described in p have been applied to s.

We denote by *AllExecutions(p,s)* $= \{s'_1,\ldots,s'_n\}$ all the possible (potentially infinite) executions of the execution path p over a system state s, that is, all the possible states $(s'_1,\ldots,s'_n)$ that may be reached (depending on the input arguments used to call the operation) by applying p in s.

Example 16 All executions of the single path of the operation `newCity` (defined in the context of class `City`) when it is applied over an empty system state (i.e. a state in which no object exists) are *AllExecutions*($p_{newCity}$,*emptyState*) $= \{state_1,\ldots,state_n\}$, where $state_1, \ldots, state_n$ are states where a city with a specific name (given by the input argument `_name`) has been created.

```
activity newCity(in _name:  String){
  City c = new City();
  c.name = _name;
}
```

The following subsections describe the types of executability regarding the states of the system that can be reached after applying an operation.

6.2.1 Weakly Executable operations

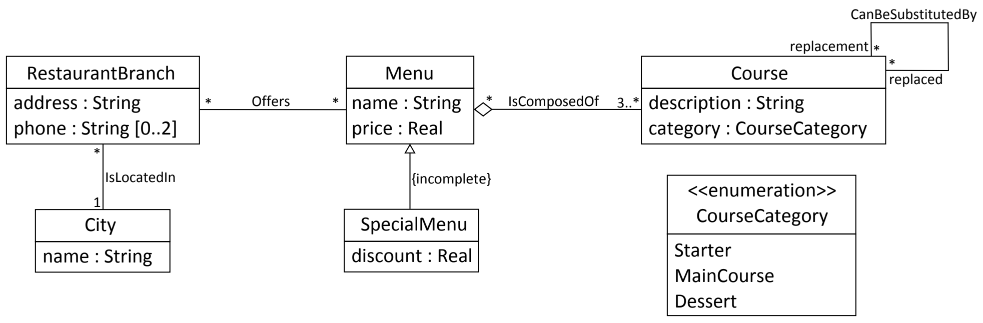
We consider an operation is *weakly executable* (WE) if it may generate a consistent state, but it is not guaranteed to do so: at least one of the many possible executions of the operation during the life span of the system will be successfully executed but probably not all of them (e.g. depending on the input arguments).

In other words, an operation is WE if there is a chance that a user may successfully execute it, i.e. if exists at least an initial state and a set of arguments for the operation parameters for which the execution of the actions included in the operation evolves the initial state of the system to a new state that satisfies all the integrity constraints of the structural model. Note that *weak executability* does not require all executions of the operation to be successful.

More formally:

Let ExM = ⟨SM,BM⟩ be an executable model, an operation op ∈ BM is **weakly executable** (WE) iff ∃ p ∈ Paths(op) ∧ ∃ s IsConsistent(s,SM) ∧ ∃ s' ∈ AllExecutions(p,s) | IsConsistent(s',SM).

Where *Paths(op)* returns the execution paths of the operation *op*; and, as we stated in Chapter 2, *IsConsistent(s,SM)* states whether the state *s* is consistent regarding the structural model *SM*.



context Menu **inv** menuPrimaryKey: Menu.allInstances()->isUnique(name)	ID(name,Menu)
context SpecialMenu **inv** atMost3SpecialMenus: SpecialMenu.allInstances()->size()<=3	Cmax(SpecialMenu)=3
context SpecialMenu **inv** validDiscount: self.discount >= 10	ValueComp(self.discount,>=,10)
context Course **inv** symmetricAssociation: self.replaced -> forAll(c|c.replaced -> includes(self))	Sym(CanBeSubstitutedBy)

Figure 6.11. Excerpt of a restaurant chain class diagram.

Example 17 Operation `newCourse` (defined in the context of class `Course`) is not WE since it never may generate a consistent system state regarding the structural model we used along this thesis (see Figure 6.11). Every time we try to create a new course `c` but we do not assign any category for it, we reach an inconsistent system state where `c` has no category, a situation forbidden by the structural model that defines the attribute `category` as mandatory (Mand(`category`,`Course`)), i.e. it must have at least one value.

```
activity newCourse(in _description:  String,
in _substitutingCourses:Course[*]) {
  Course c = new Course();
  c.description = _description;
  for ( i in 1.._substitutingCourses→size() ) {
   CanBeSubstitutedBy.createLink(replaced=>c,
   replacement=>_substitutingCourses[i]);
  }
 }
```

Example 18 Instead, operation `classifyAsSpecialMenu` (defined in the context of class `Menu`) is WE since we are able to find an execution scenario (a system state that contains less than three special menus) where the menu can be successfully subtyped.

```
activity classifyAsSpecialMenu(in _discount:  Real) {
  if ( _discount ≥ 10 ) {
   classify self to SpecialMenu;
   self.discount = _discount;
  }
}
```

Note that classifying an operation as WE does not mean that every time this operation is executed the new system state will be consistent with the integrity constraints. For instance, if the system state where we apply the above operation already contains three special menus, the operation will fail because of the integrity constraint Cmax(`SpecialMenu`)=3, which defines the class `SpecialMenu` may have at most three instances.

6.2.2 Strongly Executable operations

We consider an operation is *strongly executable* (SE) if it is guaranteed to always generate a consistent state: all the executions of the operation (regardless of the input values provided to the operation and the initial system state where the operation is applied over) reach a consistent state with respect to the structural model and their integrity constraints.

In other words, an operation is SE if it is always successfully executed, i.e. if every time we execute the operation (whatever values are given as arguments for its parameters), the effect of the actions included in the operation evolves the initial state of the system to a new state that satisfies all integrity constraints of the structural model. Note that, unlike *weak executability*, *strong executability* requires all executions of the operation to be successful.

More formally:

Let ExM = ⟨SM,BM⟩ be an executable model, an operation op ∈ BM is **strongly executable** (SE) iff ∀ p ∈ Paths(op) ∧ ∀ s IsConsistent(s,SM) ∧ ∀ s' ∈ AllExecutions(p,s) IsConsistent(s',SM).

Example 19 As we have seen, operation `classifyAsSpecialMenu` is not SE since after its execution we may violate the maximum cardinality integrity constraint of class `SpecialMenu` (Cmax(`SpecialMenu`)=3), in particular, when the system state where the operation is applied already contains three special menus.

Example 20 Instead, operation `addMenu` (defined in the context of class `Menu`) is SE since we may guarantee it will never violate any integrity constraint of the structural model after their execution.

```
activity addMenu(in _name:  String, in _price:  Real, in _courses:
Course[3..*]) {
  if ( !Menu.allInstances()→exists(m|m.name=_name) ) {
   Menu m = new Menu();
   m.name = _name;
   m.price = _price;
   for ( i in 1.._courses→size() ) {
    IsComposedOf.createLink(menu=>m,course=>_courses[i]);
   }
  }
}
```

Note that, in this case, the operation `addMenu` includes a guard (i.e. a precondition) to guarantee the changes this operation performs will only be executed in a safe context.

6.2.3 Non Executable operations

Operations that are not WE (nor SE), are non executable. *Non executable* operations never generate a consistent system state. After their execution, they always reach a system state that violates some integrity constraints of the structural model (e.g. some cardinality constraints).

More formally:

Let ExM = ⟨SM,BM⟩ be an executable model, an operation op ∈ BM is **non executable** iff ∀ p ∈ Paths(op) ∧ ∀ s IsConsistent(s,SM) ∧ ∀ s' ∈ AllExecutions(p,s) ¬IsConsistent(s',SM).

Example 21 Since `newCourse` is not WE, it is non executable.

6.2.4 Comparative relation

The previous properties can also be characterized in terms of probabilities. In [11], A. J. Ayer established that "a proposition is said to be verifiable, in the strong sense of the term, if and only if, its truth could be conclusively established in experience. But it is verifiable in the weak sense, if it is possible for experience to render it probable". Then, we can say that non executable operations generate a consistent state with probability 0, weakly executable operations generate a consistent state with probability strictly greater than 0 and strongly executable operations generate a consistent state with probability exactly 1 (see Figure 6.12).

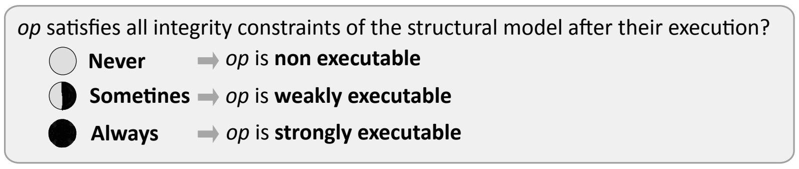

op satisfies all integrity constraints of the structural model after their execution?
Never ⇒ *op* is **non executable**
Sometines ⇒ *op* is **weakly executable**
Always ⇒ *op* is **strongly executable**

Figure 6.12. Types of executability.

Making sure that all operations are weakly executable ensures a first basic level of correctness. Besides, making sure that all operations are strongly executable ensures a full level of correctness, by facilitating a lot the development (and run-time efficiency) of the system: since we know that operations always leave the system in a consistent state, we can avoid checking at the end of each operation execution whether all constraints are satisfied which improves the efficiency of the system. Building (and executing) such integrity checking mechanism is an error-prone and time-consuming process that can be avoided when using our method to guarantee the correctness of all operations.

We consider the two levels of correctness are important. On the one hand, weak executability could probably be achieved directly by the designers (without using code generation neither verification tools) and it is less costly to verify. On the other hand, strongly executability is more difficult to be achieved directly by the designers (see the results of our experiment in Chapter 10) but it is also more costly to verify. In any case, the designer should choose which level of correctness wants to guarantee.

On the other hand, non executable operations are completely useless since every time a user tries to execute them (regardless the provided input values) an error arises because some integrity constraints become violated.

As a final remark, notice also that weak executability is a necessary (but not sufficient) condition for strong executability. Then SE operations are a subset of WE operations (see Figure 6.13).

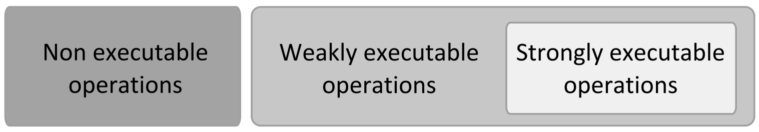

Non executable operations
Weakly executable operations
Strongly executable operations

Figure 6.13. Executability classification.

6.3 Verifying the Executability of Alf operations

The lightweight method we have developed for verifying the executability property (see Figure 6.14) takes as input an executable model composed by a structural model (a UML class diagram) and an Alf-based operation (as part of the behavioural model). We consider all the operations of the behavioural model are syntactically correct. Then, our method returns either a positive answer, meaning that the operation is WE/SE or a corrective feedback (see Section 6.3.4), consisting in a set of actions and guards that should be added to the operation in order to make it WE/SE. Note that, extending the operation with the provided feedback is a necessary condition but not a sufficient one to immediately guarantee the WE/SE of the operation since the added actions may in its turn induce other constraint violations. Therefore, the extended operation must be iteratively reanalyzed with our method until we reach a WE/SE status.

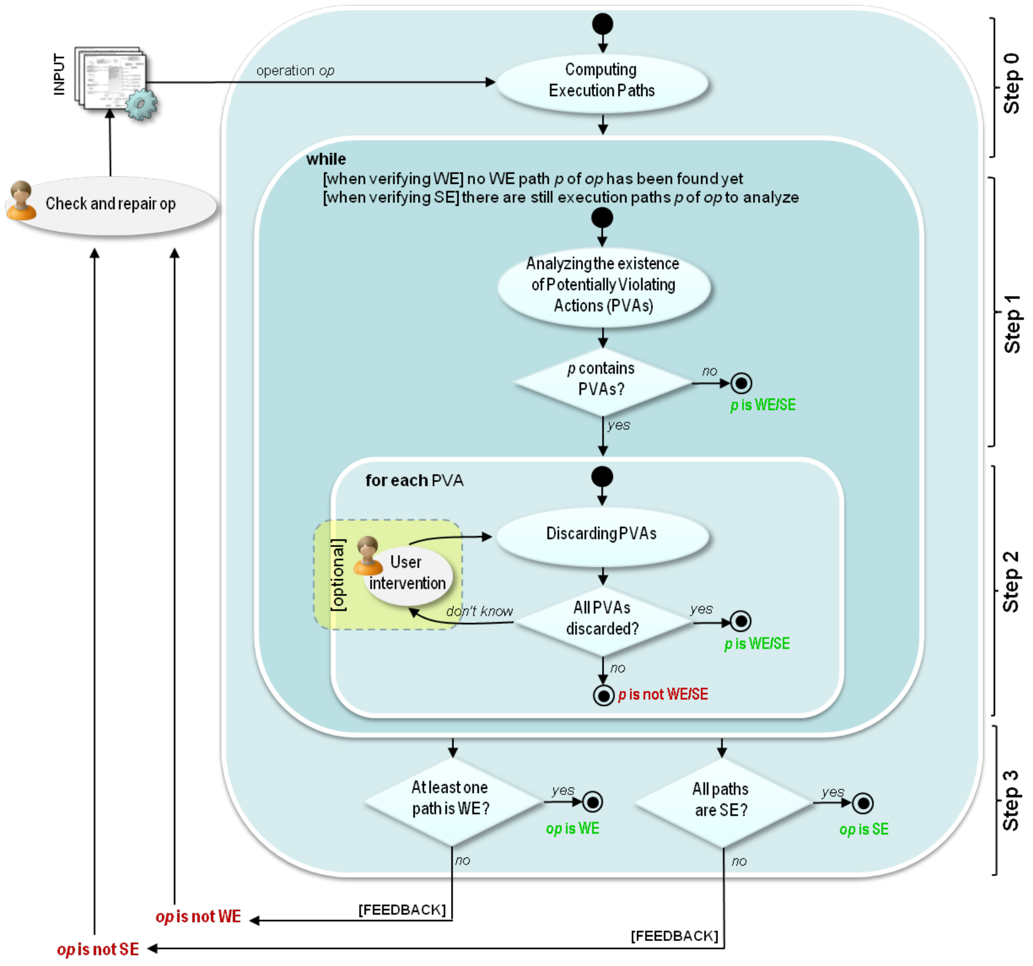


Figure 6.14. Executability method overview.

When analyzing the WE/SE of an operation we must take into account all the possible *execution paths* (see Section 6.1): an operation is WE iff at least one of its execution paths is WE; and it is SE iff all its executions paths are SE. Therefore, prior to checking the weak/strong executability of an operation, our method performs a pre-processing step to compute its execution paths (Step 0). Once the execution paths have been computed, Steps 1 and 2 of the method are applied on each path p until we recognize a WE path (in case of verifying weak executability) or until we check all paths are SE (in case of verifying strong executability). First, Step 1 (see Section 6.3.1) individually analyzes each action in the path p to see whether it may violate some integrity constraints of the structural model. Then, Step 2 (see Section 6.3.2) performs a contextual analysis of each potentially violating action to see whether other actions or conditions in p compensate or complement its effect to ensure that we sometimes/always reach a consistent state at the end of the operation execution. If all potentially violating actions can be discarded we can conclude that p is WE/SE. Finally, Step 3 (see Section 6.3.3) classifies the operation depending on the results obtained in the previous step. If at least one of the execution paths of the operation is WE, the operation is classified as WE. If all its execution paths are SE, the operation is classified as SE. Otherwise, the operation is non executable.

Our method performs an over-approximation analysis. Over-approximation is due to the lack of exhaustiveness in the comparison of conditions in the operation to favor the efficiency of the process. This implies that our method may return false positives, that is, it may return as a non-WE/SE an operation which is actually WE/SE. On the other hand, the method does not return false negatives (in our opinion, more critical than the above), that is, when it states that an operation is WE/SE, this statement is always true[5]. This over-approximation may be manually eliminated by the designer if she decides to intervene during the second step of the method in order to disambiguate the situations that cannot be computed in a lightweight manner. We believe this is a reasonable trade-off for the verification method we propose. Section 6.4 discusses more about this question.

In the following subsections we describe in detail the three main steps of our verification method (Sections 6.3.1, 6.3.2 and 6.3.3) and the feedback it provides (Section 6.3.4). Finally, Section 6.3.5 illustrates the use of our method by using our running operations.

6.3.1 Step 1: Analyzing the existence of Potentially Violating Actions

First step of our verification method analyzes each term[6] in the path to see whether the effect of the action included in the term can change the system state in a way that some integrity constraints of the structural model become violated. If so, this action is declared as *Potentially Violating Action* (PVA) and we refer to the constraints the PVA can violate as *Susceptible Violated Constraints* (SVC). If the path has no PVAs, it is WE/SE (and if we are checking if the operation is WE, we can directly confirm it at this step). Otherwise, we need to continue the analysis with the next step.

In order to detect the PVAs we have defined a set of rules that automatically determine the actions that may violate each integrity constraint of the structural model. Table 6.1 shows these rules. First column (*Susceptible Violated Constraint (SVC)*) shows each constraint our method supports (see them at Section 2.2.1) and second column (*Potentially Violation Actions (PVAs)*) determines the modification actions each constraint may violate. Several subrows for the same integrity constraint indicate several actions that may violate this constraint. Sharp sign (#) represents irrelevant variables and consecutive letters (`x`, `y`,...) represent free variables that

[5]This statement is true as long as the assumptions we discuss in Section 6.4 are fulfilled.

may be bound to any value in the term. Note that, when the minimum cardinality constraint of a class (Cmin(`cl`)) or of an association (Cmin(`as`,`role`)) is not restricted (i.e. it is equal to zero), then no action may violate this constraint. Similarly, when the maximum cardinality constraint of a class (Cmax(`cl`)) or of an association (Cmax(`as`,`role`)) is not restricted (i.e. it is equal to "*"), then no action may violate this constraint.

Table 6.1. Rules to determine the actions that may violate each integrity constraint.

	Susceptible Violated Constraint (SVC)	*Potentially Violating Actions (PVAs)*
1	Cmin(`cl`)≠0	`o.destroy`, where `o` is an instance of class `cl` or of a subclass of `cl`
		`classify x from oldCl`, where `oldCl` includes the class `cl` or one of its subclasses
2	Cmax(`cl`)≠*	`x = new cl()`
		`x = new cl'()`, where `cl'` is a subclass of `cl`
		`classify x to [newCl]`, where `newCl` includes the class `cl` (only applies when `cl` is child of a generalization)
3	Mand(`attr`,`cl`)	`x = new cl()`
		`x = new cl'()`, where `cl'` is a subclass of `cl`
		`classify x to newCl`, where `newCl` includes the class `cl` (only applies when `cl` is child of a generalization)
		`x.attr = null`, where `x.oclIsTypeOf(cl)` or `x.oclIsTypeOf(cl')`, where `cl'` is a subclass of `cl`
4	Cmin(`as`,`r`)≠0	`x = new cl()`, where `cl` (or one of its superclasses) participates on the association `as` with role `r'` (`r'` is the opposite role to `r` in `as`)
		`classify x to newCl`, where `newCl` includes the class `cl` and `cl` (or one of its superclasses) participates on the association `as` with role `r'` (`r'` is the opposite role to `r` in `as`) (only applies when `cl` is child of a generalization)
		`as.destroyLink(r=>x,r'=>y)`, where `r'` is the opposite role to `r` in `as`
		`as.clearAssoc(o)`, where `o` participates on the association `as` with role `r'` (`r'` is the opposite role to `r` in `as`)
5	Cmax(`as`,`r`)≠*	`as.createLink(r=>x,r'=>y)`, where `r'` is the opposite role to `r` in `as`
6	Disj(`cl`,{cl_1,..., cl_n})	`classify x to newCl`, where `newCl` includes one cl_i
7	Cov(`cl`,{cl_1,..., cl_n})	`classify x from oldCl`, where `oldCl` includes one cl_i
8	ID(`attr`,`cl`)	`o.attr = #`, where `o` is an instance of the class `cl` or of a subclass of `cl`

Continued on next page

Table 6.1 – continued from previous page

	Susceptible Violated Constraint (SVC)	*Potentially Violating Actions (PVAs)*
9	Sym(`as`)	`as.createLink(r=>x,r'=>y)`, where `r` and `r'` are the roles which participate in `as`
		`as.destroyLink(r=>x,r'=>y)`, where `r` and `r'` are the roles which participate in `as`
10	Asym(`as`)	`as.createLink(r=>x,r'=>y)`, where `r` and `r'` are the roles which participate in `as`
11	Irrefl(`as`)	`as.createLink(r=>x,r'=>x)`, where `r` and `r'` are the roles which participate in `as`
12	ValueComp (`attr`,op,v)	`o.attr = #`, where `o` is an instance of the class which owns `attr` or of one of its subclasses
13	Referential(`cl`,`as`)	`classify o from oldCl`, where `o.oclIsTypeOf(cl)` (before classifying `o`), `oldCl` includes the class `cl` and `cl` participates on the association `as` (only applies when `cl` is child of a generalization)

In the following we discuss each row of Table 6.1:

1. First row determines the actions that may violate a minimum cardinality constraint of a class `cl` when it is different to zero (Cmin(`cl`)≠0). This constraint may be violated when we destroy an object of class `cl` or of a subclass of `cl`, that is, when the number of instances of `cl` is decreased (first subrow); or when we take out an object from class `cl` or from one of its subclasses (second subrow).

2. Second row determines the actions that may violate a maximum cardinality constraint of a class `cl` when it is different to "*" (Cmax(`cl`)≠*). This constraint may be violated when we create an object of class `cl` (first subrow) or of a subclass of `cl` (second subrow); or when we classify an object to class `cl` (third subrow).

3. Third row determines the actions that may violate a mandatory attribute constraint (Mand(`attr`,`cl`)). This constraint may be violated when we create an object of class `cl` (first subrow) or of a subclass of `cl` (second subrow); when we classify an object to class `cl` (third subrow); or when we remove the value of the attribute (fourth subrow).

4. Fourth row determines the actions that may violate a minimum cardinality constraint of an association `as` in the role `r` when it is different to zero (Cmin(`as`,`r`)≠0). This constraint may be violated when we create a new object of class `cl` (where `cl` or one of its superclasses participates on the association `as` with role `r'`, and `r'` is the opposite role to `r` in `as`) (first subrow); or when we classify an object to `cl` (second subrow); or when we destroy a link of `as` (third subrow); or when we clear the association `as` (fourth subrow).

5. Fifth row determines the action that may violate a maximum cardinality constraint of an association `as` in the role `r` when it is different to "*" (Cmax(`as`,`r`)$\neq$*). This constraint may be violated when we create a link of `as`.

6. Sixth row determines the action that may violate a disjointness of a generalization constraint (Disj(`cl`,cl_1,...,cl_n), where `cl` generalizes cl_1,...,cl_n). This constraint may be violated when we classify an object to one subclass of `cl`.

7. Seventh row determines the action that may violate a covering of a generalization constraint (Cov(`cl`,cl_1,...,cl_n), where `cl` generalizes cl_1,...,cl_n). This constraint may be violated when we take off an object from one subclass of `cl`.

8. Eight row determines the action that may violate a identifier of an attribute constraint (ID(`attr`,`cl`)). This constraint may be violated when we assign a new value to the attribute `attr` of an object of type `cl`.

9. Ninth row determines the actions that may violate a symmetric constraint of a recursive association `as` (Sym(`as`)). This constraint may be violated when we create (first subrow) or destroy (second subrow) a link of `as`.

10. Tenth row determines the action that may violate an asymmetric constraint of a recursive association `as` (Asym(`as`)). This constraint may be violated when we create a link of `as`.

11. Eleventh row determines the action that may violate a irreflexive constraint of a recursive association `as` (Irrefl(`as`)). Similarly, this constraint may be violated when we create a link of `as`.

12. Twelfth row determines the action that may violate a value comparison constraint over an attribute (ValueComp(`attr`,op,v)). This constraint may be violated when we assign a new value to the attribute `attr`.

13. Finally, thirteenth row determines the action that may violate a referential constraint over an association (Referential(`cl`,`as`)). This constraint may be violated when we take off an object from the class which participates on the association `as`. Note that this constraint is not violated when we destroy an object of type `cl`, because the Alf semantics for the action *DestroyObject* ensures the destruction of all links in which the destroyed object participates.

In this first step, the rules of Table 6.1 are applied over all the integrity constraints that appear in the input structural model. As a result, we obtain the set of potentially violating actions that may violate each integrity constraint of the structural model. Then, we may determine whether a path p contains PVAs by comparing this set of actions with the set of actions which appear in p. All actions in the intersection of both sets are PVAs (see Figure 6.15).

In order to do this comparison a mapping between the PVAs obtained from Table 6.1 and the actions of the path has to be done. An action of the first set (cointaining generic PVAs) can be mapped onto an action of the second set (containing specific PVAs obtained from the operation paths) when the following conditions are satisfied: (1) both actions are from the same type

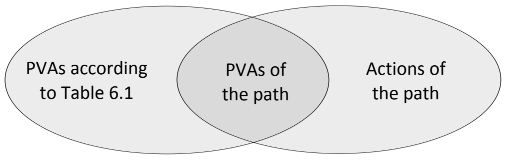


Figure 6.15. Potentially Violating Actions (PVAs).

(e.g. *CreateObject*, *ReclassifyObject*, etc.); (2) the model elements referenced by the actions coincide (e.g. both *CreateObject*s create objects of the same class); and (3) all instance-level parameters of the generic PVA (i.e. variables x, y,...) can be bound to the parameters of the specific PVA (irrelevant variables - i.e. # - may be bound to any parameter value in the specific PVA).

Example 22 In order to illustrate how this comparison is done, consider a simplified version of our initial class diagram (see Figure 6.16) and the operation addSpecialMenu (defined in the context of class SpecialMenu) to add a new special menu to the system.

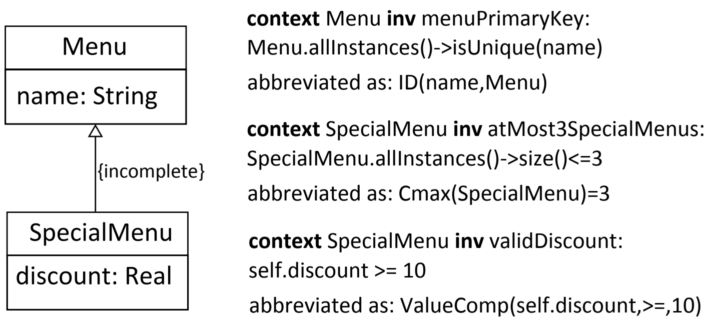

Menu
name: String
{incomplete}
SpecialMenu
discount: Real
context Menu inv menuPrimaryKey:
Menu.allInstances()->isUnique(name)
abbreviated as: ID(name,Menu)
context SpecialMenu inv atMost3SpecialMenus:
SpecialMenu.allInstances()->size()<=3
abbreviated as: Cmax(SpecialMenu)=3
context SpecialMenu inv validDiscount:
self.discount >= 10
abbreviated as: ValueComp(self.discount,>=,10)

Figure 6.16. Simplified class diagram to illustrate how PVAs are obtained.

```
activity addSpecialMenu(_name:  String, _discount:  Real) {
  SpecialMenu sm = new SpecialMenu();
  sm.name = _name;
  sm.discount = _discount;
}
```

Table 6.2 shows the PVAs (second column) and the SVCs they may violate (first column) according to Table 6.1.

Table 6.3 shows the actions of the single path of operation addSpecialMenu (first column) and points out which of them are PVAs (second column).

Then, the single execution path of the operation addSpecialmenu has three PVAs:

- First action (SpecialMenu sm = new SpecialMenu()) may violate three constraints: (1) Cmax(SpecialMenu)=3, (2) Mand(name,Menu) and (3) Mand(discount,SpecialMenu).
- Second action (sm.name = _name) may violate one constraint: ID(name,Menu).

Table 6.2. Computing the PVAs according to Table 6.1.

SVCs	*PVAs that may violate the SVC*
Cmax(`SpecialMenu`)=3	`x = new SpecialMenu()`
	`classify x to SpecialMenu`
Mand(`name`,`Menu`)	`x = new Menu()`
	`x = new SpecialMenu()`
	`o.name = null`, where `o` is an instance of `Menu` or of `SpecialMenu`
Mand(`discount`,`SpecialMenu`)	`x = new SpecialMenu()`
	`classify x to SpecialMenu`
	`o.discount = null`, where `o` is an instance of `SpecialMenu`
ID(`name`,`Menu`)	`o.name = #`, where `o` is an instance of `Menu` or of `SpecialMenu`
ValueComp(`discount`,`>=`,`10`)	`o.discount = #`, where `o` is an instance of `SpecialMenu`

Table 6.3. Obtaining PVAs from operation `addSpecialMenu`.

Actions of the operation `addSpecialMenu`	*Is PVA?*
`SpecialMenu sm = new SpecialMenu()`	Yes
`sm.name = _name`	Yes
`sm.discount = _discount`	Yes

- Last action (`sm.discount = _discount`) may also violate one constraint: ValueComp(`discount`,`>=`,`10`).

If the intersection of both sets is the empty set, there are no PVAs (and then, the path is WE and SE). Otherwise, we need to continue the analysis with the next step in order to determine whether the path is WE/SE or not.

6.3.2 Step 2: Discarding the Potentially Violating Actions

It may happen that the context in which a PVA is executed within the path guarantees that the effect of the PVA is not going to actually violate any of its SVCs. In these cases, the PVA may be discarded. Roughly, there are two ways to discard a PVA: (1) when the path contains a guard (i.e. a precondition) that ensures the PVA will only be executed in a safe context; and (2) when the path contains another action which counters or complements the effect of the PVA in order to maintain the integrity of the system after executing the operation.

In this second step, our method analyzes each PVA returned by the previous step and tries to discard them by analyzing the two possibilities commented above. If all PVAs that may compromise the WE/SE of the path can be discarded, then it is classified as WE/SE. If not, the path is marked as non-WE/SE and the corresponding corrective feedback is provided (see Section 6.3.4).

Note that, if a PVA may violate several SVCs, the PVA may be discarded iff it satisfies all the conditions to avoid violating each SVC.

Next subsections (see Tables 6.5 to 6.22) describe the conditions that the path must satisfy in order to discard a specific PVA when it may violate a specific SVC.

Before reading the tables

In order to facilitate the understanding of the following tables, first we show an empty table (see Table 6.4) describing the content of each cell.

Note that the three types of patterns (formal, Alf and textual description) are equivalent ways to express the conditions to discard a specific PVA when it may violate a specific SVC (both indicated in the caption of the table). Note also that several subrows indicate several different alternative conditions that can be used to discard that PVA; meaning that at least one of them must be satisfied by the path.

Note also that, when we are verifying whether an operation is WE, we have more alternatives to discard a PVA than when we are verifying whether it is SE. This is because, since we are verifying a weaker property, there may be more scenarios which satisfy the property. Consequently, the conditions to discard a PVA in order to make an operation SE is a subset of the conditions to discard a PVA in order to make the same operation WE.

As we introduced, the conditions in the tables are expressed as patterns that should be matched to the path. When trying to match guard expressions (for instance, conditional structures) we follow a syntactic approach, i.e. we do not try to formally prove the expression in the path implies the expression in the guard (which would be too costly for general expressions) but just to check whether a path expression matches one of the syntactic variation patterns predefined for the condition (even if tables only show one possible syntactic pattern, we have computed several variations for most of them). When the algorithm cannot conclude the implication it assumes that one does not imply the other. This is why the method over-approximates the results (as a necessary trade-off to foster the efficiency of the method) as commented in the introduction of this section. The designer could optionally participate in this step to manually identify those implications that were not found by the method using its syntactic approach.

PVA: CreateObject: o = new cl()

This subsection shows the conditions to discard a PVA of type **CreateObject** (term <t_i, 1, `o = new cl()`>) when it may violate a constraint (SVC) of type maximum cardinality of a class (see Table 6.5), mandatory attribute (see Table 6.6) or minimum cardinality of an association (see Table 6.7).

[7] n is the multiplicity of `o = new cl()` and `classify x to [cl]` in $t_j \ldots t_{i-1}$.

[8] m is the multiplicity of `o.destroy()` (`o.oclIsTypeOf(cl)`) and `classify x from [cl]` in $t_j \ldots t_{i-1}$.

[9] $nTerms$ is the number of terms of the path we are addressing.

[10] The method requires a different *DestroyObjectAction* (`destroy`) / *ReclassifyObjectAction* (`classify from`) action for each PVA.

Table 6.4. Example table. The caption of each table contains the formal description of the term which acts as PVA: $<t_i$, $multiplicity_i$, $action_i>$. The caption also indicates in what case the conditions of the table have to be applied, i.e. which specific constraint (SVC) is avoided to be violated when the PVA may be discarded.

	Property	*Conditions to discard the PVA*	
Row	States whether the condition must hold only when verifying weak executability (indicated as WE) or both weak and strong executability (indicated as WE/SE).	*Formal pattern*	Describes the discarding pattern expressed in reference of the *terms* and *guards* that the path should or should not include in order to discard the PVA (when it may violate the SVC) indicated in the caption of the table. Note that most of the formal patterns include references to i, j, k,... variables, where i makes reference to the position of the PVA inside the path (as can be seen in the caption of the table), and j, k,... make reference to the position of the rest of the actions wrt the PVA (some of them also appear in the footnotes of the table). Note also that the sharp sign (#) represents irrelevant variables.
		Alf pattern	Describes the discarding pattern expressed in reference of Alf statements that the path should or should not include in order to discard the PVA (when it may violate the SVC) indicated in the caption of the table. Note that some Alf patterns include the PVA - meaning that the PVA should be executed $before$ or $after$ some specific actions - while others do not include it - meaning that the position of the PVA is irrelevant inside the path -.
		Description	Describes the discarding pattern expressed using a textual *description* describing what the path should or should not include in order to discard the PVA (when it may violate the SVC) indicated in the caption of the table.

Table 6.5. Alternative conditions to discard a PVA of type **CreateObject** (term $<t_i$, 1, `o = new cl()`$>$) when it may violate a constraint (SVC) of type maximum cardinality of a class: **Cmax**(`cl`)$\neq$*.

	Property	*Conditions to discard the PVA*	
1	WE/SE	*Formal pattern*	$\exists <g_{\#}$, `cl.allInstances()`$\rightarrow$`size()`<Cmax(`cl`)-n[7]+m[8], t_j, t_k>, 1≤j≤i≤k≤nTerms[9]

		Alf pattern	`if ( cl.allInstances()→size()<Cmax(cl)-n`[7]`+m`[8]`{`
			` o = new cl(); //PVA`
			` ...`
			`}`

		Description	The path contains a guard that prevents the execution of the PVA when `cl` has already the maximum number of instances.
2	WE/SE	*Formal pattern*	$\exists$ <t_j, ≥1, `x.destroy()`[10]>, `x.oclIsTypeOf(cl)` and ∀ i,j
		Alf pattern	`x.destroy()`[10]`; //x.oclIsTypeOf(cl)`
		Description	The path includes an action to destroy an instance of `cl` that compensates the creation of the new object.
3	WE/SE	*Formal pattern*	$\exists$ <t_j, ≥1, `classify x from cl`[10]>, `x.oclIsTypeOf(cl)` and ∀ i,j
		Alf pattern	`classify x from cl`[10]`; //x.oclIsTypeOf(cl)`
		Description	The path includes an action to classify an object from `cl` to another class in order to compensate the creation of the new object. Note that this row only applies when `cl` is part (as a child) of a generalization set.
4	WE	*Formal pattern*	n[7]-m[8]<Cmax(`cl`)-Cmin(`cl`), $1\leq j \leq i$

		Alf pattern	x_1`.destroy(); //`x_1`.oclIsTypeOf(cl)`
			`...`
			x_m`.destroy(); //`x_m`.oclIsTypeOf(cl)`
			y_1` = new cl();`
			`...`
			y_n` = new cl();`
			o` = new cl(); //PVA`
			`//n`[7]`-m`[8]`<Cmax(cl)-Cmin(cl)`

		Description	The total number of creations minus deletions of objects of class `cl` before the PVA does not exceed the difference between the maximum and the minimum cardinalities of `cl`.

Example 23 In order to illustrate how Table 6.5 operates, consider a variant of our initial class diagram (see Figure 6.17) and the operation addSpecialMenu (defined in the context of class SpecialMenu) to add a new special menu to the system.

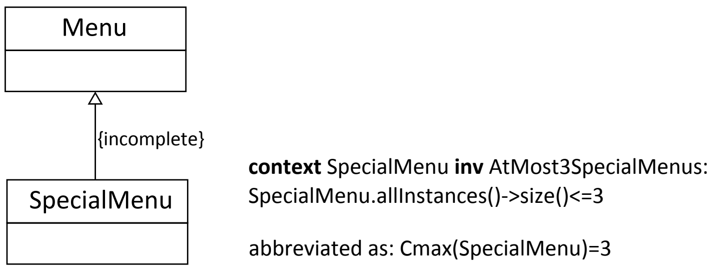

Menu
{incomplete}
SpecialMenu
context SpecialMenu **inv** AtMost3SpecialMenus:
SpecialMenu.allInstances()->size()<=3
abbreviated as: Cmax(SpecialMenu)=3

Figure 6.17. Simplified class diagram to illustrate the use of Table 6.5.

```
activity addSpecialMenu() {
  SpecialMenu sm = new SpecialMenu();
}
```

The operation addSpecialMenu is WE since it satisfies the 4th row of Table 6.5: the total number of creations (without considering the PVA) (n=0) minus deletions (m=0) does not exceed the difference between the maximum (Cmax(SpecialMenu)=3) and the minimum (Cmin(SpecialMenu)=0) cardinalities of class SpecialMenu, i.e. 0-0<3-0. It means that this operation will be successfully executed when we are not close to the maximum number of objects (i.e. at most two existing special menus).

However, this operation is not SE since it does not satisfy any of the necessary conditions for being SE (rows 1 to 3 of Table 6.5).

In order to become SE, the operation addSpecialMenu must satisfy one of the following conditions:

1. As row 1 of Table 6.5 states, the path contains a guard to ensure the PVA is only executed when there exist less than three special menus. The result of adding this guard to the operation is:

```
activity addSpecialMenu_row1Added() {
  if ( SpecialMenu.allInstances()→size() < 3 ){
   SpecialMenu sm = new SpecialMenu();
  }
}
```

2. As row 2 of Table 6.5 states, the path includes an action to destroy an existing special menu to compensate the creation of the menu sm. The result of adding this action to the operation is:

```
activity addSpecialMenu_row2Added() {
  SpecialMenu existingSm = getSpecialMenu();
  existingSm.destroy();
  SpecialMenu sm = new SpecialMenu();
}
```

Where the operation `getSpecialMenu()` returns an existing special menu.

3. As row 3 of Table 6.5 states, the path includes an action to take off an existing special menu from `SpecialMenu`. The result of adding this action to the operation is:

```
activity addSpecialMenu_row3Added() {
  SpecialMenu existingSm = getSpecialMenu();
  classify existingSm from SpecialMenu;
  SpecialMenu sm = new SpecialMenu();
}
```

Table 6.6. Conditions to discard a PVA of type **CreateObject** (term <t_i, 1, `o = new cl()`>) when it may violate a constraint (SVC) of type mandatory attribute: **Mand**(`attr`,`cl`).

	Property	*Conditions to discard the PVA*	
1	WE/SE	*Formal pattern*	$\exists$ <t_j, $\geq$1, `o.attr = #`>, i<j

		Alf pattern	`o = new cl(); //PVA`
			 `...`
			 `o.attr = #;`

		Description	The path includes, after the PVA, at least one action to initialize the attribute *attr*.

Example 24 In order to illustrate the how Table 6.6 operates, consider a variant of our initial class diagram (see Figure 6.18) and the operation `addRestaurantBranch` (defined in the context of class `RestaurantBranch`) to add a new restaurant branch to the system.

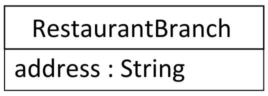


Figure 6.18. Simplified class diagram to illustrate the use of Table 6.6.

```
activity addRestaurantBranch() {
  RestaurantBranch rb = new RestaurantBranch();
}
```

The operation `addRestaurantBranch` is not WE nor SE since it does not satisfy the necessary condition for being WE/SE (row 1 of Table 6.6). Then, after the execution of this operation we always reach an inconsistent state of the system where the new restaurant branch has no address (a situation forbidden by the constraint Mand(`address`,`RestaurantBranch`)).

In order to become WE/SE, as row 1 of Table 6.6 states, the operation `addRestaurantBranch` should include, after the creation of the new restaurant branch, at least one action to initialize its attribute `address`. The result of adding this action to the operation is:

```
activity addRestaurantBranch_row1Added(in _address:  String) {
  RestaurantBranch rb = new RestaurantBranch();
  rb.address = _address;
}
```

Table 6.7. Conditions to discard a PVA of type **CreateObject** (term <t_i, 1, `o = new cl()`>) when it may violate a constraint (SVC) of type minimum cardinality of an association: **Cmin**(`as`,r_b)≠0. Note: r_b may be replaced for the opposite role of `as` (i.e. r_a).

	Property	*Conditions to discard the PVA*	
1	WE/SE	*Formal pattern*	∃ <t_j, ≥Cmin(`as`,r_b), `as.CreateLink(`r_a`=>o,`r_b`=>x)`>, i<j

		Alf pattern	`o = new cl(); //PVA`
			`...`
			`for ( i in 1..≥Cmin(as,`r_b`) ) {`
			`  ...`
			`  as.createLink(`r_a`=>o,`r_b`=>x);`
			`  ...`
			`}`

		Description	The path includes, after the PVA, at least Cmin(`as`,r_b) actions to create a link of `as` between the new object `o` (with role r_a) and another object (with role r_b).

Example 25 In order to illustrate how Table 6.7 operates, consider a variant of our initial class diagram (see Figure 6.19) and the operation `addMenu` (defined in the context of class `Menu`) to add a new menu to the system.

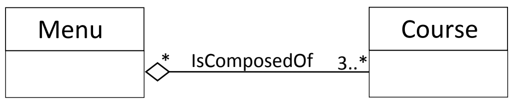


Figure 6.19. Simplified class diagram to illustrate the use of Table 6.7.

```
activity addMenu() {
  Menu m = new Menu();
}
```

The operation `addMenu` is not WE nor SE since it does not satisfy the necessary condition for being WE/SE (row 1 of Table 6.7). Then, after the execution of this operation we always reach an inconsistent state of the system where the new menu is not linked to any course (a situation forbidden by the constraint Cmin(`IsComposedOf`,`course`)=3).

In order to become WE/SE, as row 1 of Table 6.7 states, the operation addMenu should include, after the creation of the new menu, at least three actions to create a link of IsComposedOf between the new menu (m) and any course.

```
activity addMenu_row1Added(in _courses: Course[3..*]) {
  Menu m = new Menu();
  for ( i in 1..≥_courses→size() ){
    IsComposedOf.createLink(menu=>m,course=>_courses[i]);
  }
}
```

PVA: DestroyObject: o.destroy()

This subsection shows the conditions to discard a PVA of type **DestroyObject** (term <t_i, 1, o.destroy()>) when it may violate a constraint (SVC) of type minimum cardinality of a class (see Table 6.8).

Example 26 In order to illustrate how Table 6.8 operates, consider a variant of our initial class diagram (see Figure 6.20) and the operation destroySpecialMenu (defined in the context of class SpecialMenu) to destroy an existing special menu.

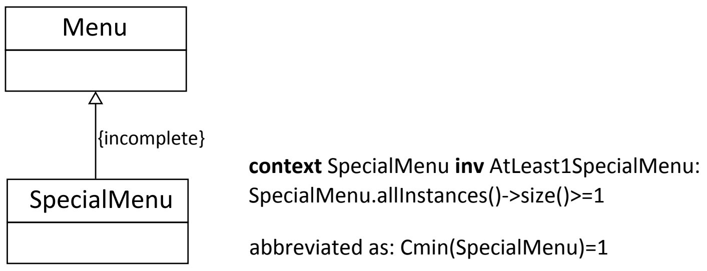

Menu
{incomplete}
SpecialMenu
context SpecialMenu inv AtLeast1SpecialMenu:
SpecialMenu.allInstances()->size()>=1
abbreviated as: Cmin(SpecialMenu)=1

Figure 6.20. Simplified class diagram to illustrate the use of Table 6.8.

```
activity destroySpecialMenu() {
  self.destroy();
}
```

The operation destroySpecialMenu is WE since it satisfies the 4th row of Table 6.8: the total number of deletions (without considering the PVA) (n=0) minus creations (m=0) does not exceed the difference between the maximum (Cmax(SpecialMenu)=*) and the minimum (Cmin(SpecialMenu)=1) cardinalities of class SpecialMenu, i.e. 0-0<*-1. It means that this operation will be successfully executed when we are not close to the minimum number of objects (i.e. at least two existing special menus). However, this operation is not SE since it does not satisfy any of the necessary conditions for being SE (rows 1 to 3 of Table 6.8).

In order to become SE, the operation destroySpecialMenu must satisfy one of the following conditions:

Table 6.8. Alternative conditions to discard a PVA of type **DestroyObject** (term $<t_i$, 1, `o.destroy()`$>$, where `o.oclIsTypeOf(cl)`) when it may violate a constraint (SVC) of type minimum cardinality of a class: **Cmin**(`cl`)≠0.

	Property	*Conditions to discard the PVA*	
1	WE/SE	*Formal pattern*	∃ $<g_{\#}$, `cl.allInstances()`→`size()`>Cmin(`cl`)+n[7]-m[8], t_j, t_k>, 1≤j≤i≤k≤nTerms[9]

		Alf pattern	`if ( cl.allInstances()→size()>Cmin(cl)+n`[7]`-m`[8]` ){`
			`  o.destroy(); //PVA`
			`  ...`
			`}`

		Description	The path contains a guard that prevents the execution of the PVA when `cl` has already the minimum number of instances.
2	WE/SE	*Formal pattern*	∃ $<t_j$, ≥1, x `= new cl()`[11]>, ∀ i,j
		Alf pattern	`x = new cl();`
		Description	The path includes at least one action to create an instance of `cl` that compensates the deletion of the object `o`.
3	WE/SE	*Formal pattern*	∃ $<t_j$, ≥1, `classify x to cl`[11]>, ∀ i,j
		Alf pattern	`classify x to cl`[10]`; //`x`.oclIsTypeOf(`cl`)`
		Description	The path includes at least one action to classify an object to `cl` in order to compensate the deletion of the object `o`. Note that this row only applies when `cl` is part (as a child) of a generalization set.
4	WE	*Formal pattern*	m[8]-n[7]<Cmax(`cl`)-Cmin(`cl`), 1≤j≤i

		Alf pattern	x_1 `= new cl();`
			
			`...`
			x_n `= new cl();`
			y_1`.destroy(); //`y_1`.oclIsTypeOf(cl)`
			`...`
			y_m`.destroy(); //`y_m`.oclIsTypeOf(cl)`
			o`.destroy(); //PVA`
			`//m`[8]`-n`[7]`<Cmax(cl)-Cmin(cl)`

		Description	The total number of deletions minus creations of objects of class `cl` before the PVA does not exceed the difference between the maximum and the minimum cardinalities of `cl`. Note that if Cmax(`cl`)=*, we consider it tends to infinite.

1. As row 1 of Table 6.8 states, the path contains a guard to ensure the PVA is only executed when exists more than one special menu. The result of adding this guard to the operation is:

```
activity destroySpecialMenu_row1Added() {
  if ( SpecialMenu.allInstances()→size() > 1 ) {
   self.destroy();
  }
}
```

2. As row 2 of Table 6.8 states, the path includes an action to create a new special menu to compensate the deletion of `self`. The result of adding this action to the operation is:

```
activity destroySpecialMenu_row2Added() {
  SpecialMenu sm = new SpecialMenu();
  self.destroy();
}
```

3. As row 3 of Table 6.8 states, the path includes an action to classify an existing special menu to `SpecialMenu`. The result of adding this action to the operation is:

```
activity destroySpecialMenu_row3Added() {
  Menu m = getMenu();
  classify m to SpecialMenu;
  self.destroy();
}
```

   Where the operation `getMenu()` returns an existing menu.

PVA: AddStructuralFeatureValue: o.attr = value

This subsection shows the conditions to discard a PVA of type **AddStructuralFeatureValue** (term <t_i, 1, `o.attr = value`>) when it may violate a constraint (SVC) of type value comparison (see Table 6.9) or identifier (see Table 6.10). Note that both constraints may only affect the strong executability of an operation (since we are always able to find the proper input values to make the operation WE).

Example 27 In order to illustrate how Table 6.9 operates, consider a variant of our initial class diagram (see Figure 6.21) and the operation `setDiscount` (defined in the context of class `SpecialMenu`) to modify the value of the attribute `discount`.

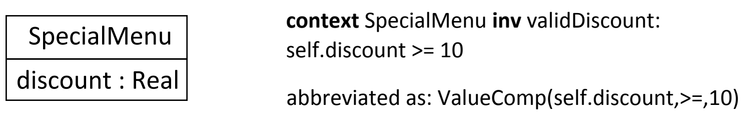

SpecialMenu
discount : Real
context SpecialMenu inv validDiscount:
self.discount >= 10
abbreviated as: ValueComp(self.discount,>=,10)

Figure 6.21. Simplified class diagram to illustrate the use of Table 6.9.

Table 6.9. Conditions to discard a PVA of type **AddStructuralFeatureValue** (term <t_i, 1, `o.attr = value`>, where `o.oclIsTypeOf(cl)`) when it may violate a constraint (SVC) of type value comparison: **ValueComp**(`attr`,`op`,`v`).

	Property	*Conditions to discard the PVA*	
1	SE	*Formal pattern*	∃ <$g_{\#}$, `value op v`, t_j, t_k>, 1≤j≤i≤k≤nTerms[9]

		Alf pattern	`if ( value op v ){`
			 `o.attr = value; //PVA`
			 `...`
			 `}`

		Description	The path contains a guard that prevents the execution of the PVA when the comparison stated in the constraint is not satisfied.

```
activity setDiscount(in _newDiscount: Real) {
  self.discount = _newDiscount;
}
```

The operation `setDiscount` is WE (since weak executability does not impose any condition in this case). It means that this operation will be successfully executed when the proper value (i.e. a value greater than 10) is given to the input argument `_newDiscount`.

However, this operation is not SE since it does not satisfy the condition for being SE (row 1 of Table 6.9). In order to become SE, the operation `setDiscount` should contain a guard to ensure the PVA is only executed when the comparison `_newDiscount>=10` is satisfied. The result of adding this action to the operation is:

```
activity setDiscount_row1Added(in _newDiscount: Real) {
  if ( _newDiscount >= 10 ) {
   self.discount = _newDiscount;
  }
}
```

Table 6.10. Alternative conditions to discard a PVA of type **AddStructuralFeatureValue** (term <t_i, 1, `o.attr = value`>, where `o.oclIsTypeOf(cl)`) when it may violate a constraint (SVC) of type identifier: **ID**(`attr`,`cl`).

	Property	*Conditions to discard the PVA*	
1	SE	*Formal pattern*	∃ <$g_{\#}$, `!cl.allInstances()→exists(x|x.attr = value)`, t_j, t_k> **AND** ∄ <t_l, ≥1, `y.attr = value`>, 1≤j≤l≤i≤k≤nTerms[9] and x≠y≠o

		Alf pattern	`if ( !cl.allInstances()→`
			 `exists(x|x.attr=value )){`
			 `//value is not assigned to other object`

Continued on next page

Table 6.10 – continued from previous page

	Property	*Conditions to discard the PVA*	

			`o.attr = value; //PVA`
			`...`
			`}`

		Description	The path contains a guard that guarantees there is no another object of type `cl` with the same value at the attribute `attr`.
2	SE	*Formal pattern*	∃ <$g_{\#}$, `x.attr = value`, t_j, t_k> **AND** ∃ <t_l, 1, `x.attr = value2`>, 1≤j≤l≤k≤nTerms[9], l<i and `value2`≠`value`

		Alf pattern	`if ( x.attr = value ){`
			
			`...`
			`x.attr = value2;`
			`...`
			`o.attr = value; //PVA`
			`...`
			`}`

		Description	The path contains an object `x` of class `cl` with value `value` for the attribute `attr`. The value of `attr` for `x` is changed to another value.
3	SE	*Formal pattern*	∃ <$g_{\#}$, `x.attr = value`, t_j, t_k> **AND** ∃ <t_l, 1, `x.destroy()`>, 1≤j≤l≤k≤nTerms[9], l<i

		Alf pattern	`if ( x.attr = value ){`
			
			`...`
			`x.destroy();`
			`...`
			`o.attr = value; //PVA`
			`...`
			`}`

		Description	The path contains an object `x` of class `cl` with value `value` for the attribute `attr`. The object `x` is destroyed.
4	SE	*Formal pattern*	∃ <$g_{\#}$, `x.attr = value`, t_j, t_k> **AND** ∃ <t_l, 1, `classify x from cl`>, 1≤j≤l≤k≤nTerms[9], l<i

		Alf pattern	`if ( x.attr = value ){`
			
			`...`
			`classify x from cl;`
			`...`
			`o.attr = value; //PVA`
			`...`
			`}`

Continued on next page

Table 6.10 – continued from previous page

	Property	*Conditions to discard the PVA*	
		Description	The path contains an object x of class cl with value value for the attribute attr. The object x is reclassified from cl to another class where attr does not belongs to. Note that this row only applies when cl is part (as a child) of a generalization set.

Example 28 In order to illustrate how Table 6.10 operates, consider a variant of our initial class diagram (see Figure 6.22) and the operation `setName` (defined in the context of class `Menu`) to modify the name of the `self` menu.

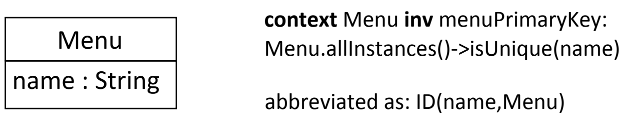

Menu
name : String
context Menu inv menuPrimaryKey:
Menu.allInstances()->isUnique(name)
abbreviated as: ID(name,Menu)

Figure 6.22. Simplified class diagram to illustrate the use of Table 6.10.

```
activity setName(in _newName:  String) {
  self.name = _newName;
}
```

The operation `setName` is WE (since weak executability does not impose any condition in this case). It means that this operation will be successfully executed when the proper value (i.e. a name different to the name of all existing menus) is given to the input argument `_newName`.

However, this operation is not SE since it does not satisfy any of the necessary conditions for being SE (rows 1 to 4 of Table 6.10).

In order to become SE, the operation `setName` must satisfy one of the following conditions:

1. As row 1 of Table 6.10 states, the path contains a guard to ensure the PVA is only executed when there is no other menu with the same name in the system. The result of adding this guard to the operation is:

   ```
   activity setName_row1Added(in _newName:  String) {
     if (!Menu.allInstances()→exists(x|x.name=_newName)){
      self.name = _newName;
     }
   }
   ```

2. As row 2 of Table 6.10 states, the path includes an action to change the name of an existing menu with name `_name`. The result of adding this action to the operation is:

```
activity setName_row2Added(in _newName:  String, in _newName2:
String) {
  if (Menu.allInstances()→exists(x|x.name=_newName)){
   Menu m = getMenu(_newName);
   m.name = _newName2;
   self.name = _newName;
  }
}
```

Where the operation `getMenu(name)` returns the menu with name `name`.

3. As row 3 of Table 6.10 states, the path includes an action to destroy the existing menu with name `_name`. The result of adding this action to the operation is:

```
activity setName_row3Added(in _newName:  String) {
  if (Menu.allInstances()→exists(x|x.name=_newName)){
   Menu m = getMenu(_newName);
   m.destroy();
   self.name = _newName;
  }
}
```

Note that, in this example, row 4 of Table 6.10 does not apply since the class `Menu` is not a child of a generalization set.

PVA: ClearStructuralFeatureAction: o.attr = null

This subsection shows the conditions to discard a PVA of type **ClearStructuralFeature-Action** (term <t_i, 1, `o.attr = null`>) when it may violate a constraint (SVC) of type mandatory attribute (see Table 6.11).

Table 6.11. Alternative conditions to discard a PVA of type **ClearStructuralFeatureAction** (term <t_i, 1, `o.attr = null`>, where `o.oclIsTypeOf(cl)`) when it may violate a constraint (SVC) of type mandatory attribute: **Mand**(`attr`,`cl`).

	Property	*Conditions to discard the PVA*	
1	WE/SE	*Formal pattern*	∃ <t_j, Cmin(`attr`), `o.attr = #`>, i<j

		Alf pattern	`o.attr = null; //PVA`
			
			`...`
			`for ( i in 1..≥ Cmin(attr) ) {`
			` ...`
			` o.attr = #;`
			` ...`
			`}`

		Description	The path includes, after the PVA, at least Cmin(`attr`) actions to add new values to the attribute, where Cmin(`attr`) is the minimum cardinality of the attribute `attr` according to the class diagram.

Example 29 In order to illustrate how Table 6.11 operates, consider a variant of our initial class diagram (see Figure 6.23) and the operation `clearAddress` (defined in the context of class `RestaurantBranch`) to clear the address of the `self` restaurant.

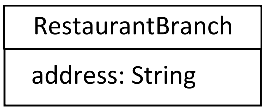


Figure 6.23. Simplified class diagram to illustrate the use of Table 6.11.

```
activity clearAddress() {
  self.address = null;
}
```

The operation `clearAddress` is not WE nor SE since it does not satisfy the necessary condition for being WE/SE (row 1 of Table 6.11). Then, after the execution of this operation we always reach a inconsistent state of the system where the restaurant `self` does not have any address assigned.

In order to become WE/SE, as row 1 of Table 6.11 states, the operation `clearAddress` must include one action (since the minimum cardinality of the `address` attribute is equal to 1) to add a new value to the attribute `address`. The result of adding this action to the operation is:

```
activity clearAddress(in _newAddress: String) {
  self.address = null;
  self.address = _newAddress;
}
```

PVA: CreateLink: as.createLink(r_a=>o_1,r_b=>o_2)

This subsection shows the conditions to discard a PVA of type **CreateLink** (term <t_i, 1, `as.createLink(`r_a=>o_1, r_b=>o_2>) when it may violate a constraint (SVC) of type maximum cardinality of an association (see Table 6.12), symmetric association (see Table 6.13), asymmetric association (see Table 6.14) or irreflexive association (see Table 6.15).

Table 6.12. Alternative conditions to discard a PVA of type **CreateLink** (term <t_i, 1, `as.createLink(`r_a=>o_1, r_b=>o_2`)`>, where o_1`.oclIsTypeOf(cl)`) when it may violate a constraint (SVC) of type maximum cardinalily of an association: **Cmax**(`as`,r_b)≠*. Note: r_b may be replaced for the opposite role of `as` (i.e. r_a).

	Property	*Conditions to discard the PVA*	
1	WE/SE	*Formal pattern*	∃ <$g_{\#}$, $o_1.r_b$→`size()`<Cmax(as,r_b)-o[12]+p[13], t_j, t_k>, 1≤j≤i≤k≤nTerms[9]

		Alf pattern	`if (` $o_1.r_b$→`size()`<`Cmax(as,`r_b`)`-o[12]+p[13] `){`
			 `as.createLink(`r_a=>o_1, r_b=>o_2`); //PVA`

Continued on next page

Table 6.12 – continued from previous page

	Property	*Conditions to discard the PVA*	

			`...`
			`}`

		Description	The path contains a guard that prevents the execution of the PVA when `as` has already the maximum number of links.
2	WE/SE	*Formal pattern*	∃ <t_j, ≥1, `as.destroyLink(`r_a`=>`o_1`,`r_b`=>`x`)`[14]>, ∀ i,j
		Alf pattern	`as.destroyLink(`r_a`=>`o_1`,`r_b`=>`x`)`[14]`;`
		Description	The path includes an action to destroy a link of `as` that compensates the creation of the new link.
3	WE/SE	*Formal pattern*	∃ <t_j, 1, o_1 `= new cl()`>, j<i **AND** ∄ <t_k, ≥Cmax(as,r_b), `as.createLink(`r_a`=>`o_1`,`r_b`=>`x`)`>, k<i

		Alf pattern	o_1 `= new cl();`
			
			`/* Less than Cmax(as,`r_b`) links of as where` o_1 `participates are created */`
			`...`
			`as.createLink(`r_a`=>`o_1`,`r_b`=>`o_2`); //PVA`

		Description	The path includes the creation of the object o_1 and does not assign Cmax(`as`,r_b) links or more before the PVA.
4	WE/SE	*Formal pattern*	∃ <t_j, 1, `classify` o_1 `to [cl]`>, j<i **AND** ∄ <t_k, Cmax(`as`,r_b), `as.createLink(`r_a`=>`o_1`,`r_b`=>`x`)`>, k<i

		Alf pattern	`classify` o_1 `to [cl];`
			
			`/* Less than Cmax(as,`r_b`) links of as where` o_1 `participates are created */`
			`...`
			`as.createLink(`r_a`=>`o_1`,`r_b`=>`o_2`); //PVA`

		Description	The path includes an action to classify the object o_1 to the class `cl` without assigning Cmax(`as`,r_b) links or more before the PVA. Note that this row only applies when `cl` is part (as child) of a generalization set.
5	WE	*Formal pattern*	o[12]-p[13]<Cmax(as,r_b)-Cmin(as,r_b), 1≤j≤i

		Alf pattern	`as.destroyLink(`r_a`=>`o_1`,`r_b`=>`x_1`);`
			
			`...`
			`as.destroy(`r_a`=>`o_1`,`r_b`=>`x_p`);`
			`as.createLink(`r_a`=>`o_1`,`r_b`=>`y_1`);`
			`...`
			`as.createLink(`r_a`=>`o_1`,`r_b`=>`y_o`);`
			`as.createLink(`r_a`=>`o_1`,`r_b`=>`o_2`); //PVA`

Continued on next page

Table 6.12 – continued from previous page

	Property	*Conditions to discard the PVA*	
			`//o`^{12}`-p`^{13}`<Cmax(`as, r_b`)-Cmin(`as, r_b`)`
		Description	The total number of link creations minus link deletions of association `as` before the PVA does not exceed the difference between the maximum and the minimum cardinalities of association `as`.

Example 30 In order to illustrate how Table 6.12 operates, consider a variant of our initial class diagram (see Figure 6.24) and the operation `addLocation` (defined in the context of class `RestaurantBranch`) to link the `self` restaurant branch with its location.

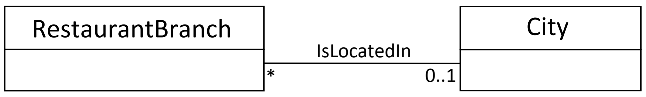


Figure 6.24. Simplified class diagram to illustrate the use of Table 6.12.

```
activity addLocation(in c:  City) {
  IsLocatedIn.createLink(restaurantBranch=>self,city=>c);
}
```

The operation `addLocation` is WE since it satisfies the 5th row of Table 6.12: the total number of link creations (without considering the PVA) (o=0) minus link deletions (p=0) does not exceed the difference between the maximum (Cmax(`IsLocatedIn,city`)=1) and the minimum (Cmin(`IsLocatedIn,city`)=0) cardinalities of the association `IsLocatedIn` in the role `city`, i.e. 0-0<1-0. It means that this operation will be successfully executed when we are not close to the maximum number of links (i.e. when `rb` is not linked to any city).

However, this operation is not SE since it does not satisfy any of the necessary conditions for being SE (rows 1 to 4 of Table 6.12).

In order to become SE, the operation `addLocation` must satisfy one of the following conditions:

1. As row 1 of Table 6.12 states, the path contains a guard to ensure the PVA is only executed when the restaurant branch is not linked to any city. The result of adding this guard to the operation is:

```
activity addLocation_row1Added(in c:  City) {
  if ( self.city->size() < 1 ){
    IsLocatedIn.createLink(restaurantBranch=>self,city=>c);
  }
}
```

2. As row 2 of Table 6.12 states, the path includes an action to destroy an existing link of `IsLocatedIn` between `self` and its current city. The result of adding this action to the operation is:

```
activity addLocation_row2Added(in c:  City) {
 if ( self.city->size() == 1 ){
  IsLocatedIn.destroyLink(restaurantBranch=>self,city=>self.
   city);
  IsLocatedIn.createLink(restaurantBranch=>self,city=>=>c);
 }
}
```

3. As row 3 of Table 6.12 states, the path includes the creation of the restaurant branch and does not assign any city before the PVA. The result of adding this action to the operation is:

```
activity addLocation_row3Added(in c:  City) {
  RestaurantBranch rb = new RestaurantBranch();
  IsLocatedIn.createLink(restaurantBranch=>rb,city=>c);
}
```

Note that, in this example, row 4 of Table 6.12 does not apply since the class `RestaurantBranch` is not a child of a generalization set.

Table 6.13. Conditions to discard a PVA of type **CreateLink** (term $<t_i$, 1, `as.createLink(`r_a`=>`o_1`,`r_b`=>`o_2`)`$>$, where o_1`.oclIsTypeOf(cl))` when it may violate a constraint (SVC) of type symmetric association: **Sym**(`as`).

	Property	*Conditions to discard the PVA*	
1	WE/SE	*Formal pattern*	$\exists <t_j$, 1, `as.createLink(`r_b`=>`o_1`,`r_a`=>`o_2`)`$>$, $\forall$ i,j
		Alf pattern	`as.createLink(`r_b`=>`o_1`,`r_a`=>`o_2`)`
		Description	The path includes the creation of the symmetric link.

Example 31 In order to illustrate how Table 6.13 operates, consider a variant of our initial class diagram (see Figure 6.25) and the operation `addSubstituteCourse` (defined in the context of class `Course`) to add a substituting course to the `self` course.

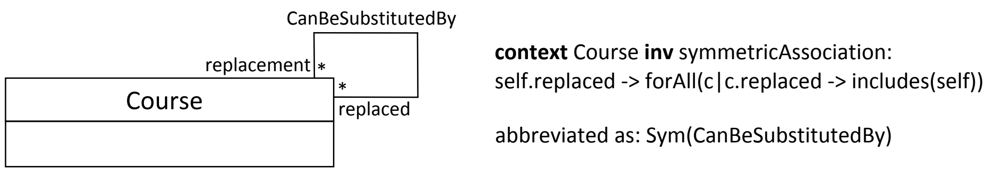

CanBeSubstitutedBy
replacement
*
*
replaced
Course
context Course inv symmetricAssociation:
self.replaced -> forAll(c|c.replaced -> includes(self))
abbreviated as: Sym(CanBeSubstitutedBy)

Figure 6.25. Simplified class diagram to illustrate the use of Table 6.13.

```
activity addSubstituteCourse(in c:  Course) {
  CanBeSubstitutedBy.createLink(replaced=>self,replacement=>c);
}
```

The operation `addSubstituteCourse` is not WE nor SE since it does not satisfy the necessary condition for being WE/SE (row 1 of Table 6.13). Then, after the execution of this operation we always reach a inconsistent state of the system where the course `self` can be substituted by `c` but `c` cannot be substituted by `self`.

In order to become WE/SE, as row 1 of Table 6.13 states, the operation `addSubstituteCourse` must include one action to create the symmetric link. The result of adding this action to the operation is:

```
activity addSubstituteCourse_row1Added(in c:  Course) {
  CanBeSubstitutedBy.createLink(replaced=>self,replacement=>c);
  CanBeSubstitutedBy.createLink(replaced=>c,replacement=>self);
}
```

Table 6.14. Conditions to discard a PVA of type **CreateLink** (term $<t_i$, 1, `as.createLink(`r_a`=>`o_1`,`r_b`=>`o_2`)`$>$, where o_1`.oclIsTypeOf(cl)`) when it may violate a constraint (SVC) of type asymmetric association: **Asym**(`as`).

	Property	*Conditions to discard the PVA*	
1	SE	*Formal pattern*	$\exists <g_{\#}$, `!`o_2`.`r_b`→includes(`o_1`)`, t_j, $t_k>$, 1≤j≤i≤k≤nTerms[9]

		Alf pattern	`if ( !`o_2`.`r_b`→includes(`o_1`) ){`
			
			`as.createLink(`r_a`=>`o_1`,`r_b`=>`o_2`); //PVA`
			`...`
			`}`

		Description	The path contains a guard that prevents the execution of the PVA when the symmetric link does not exist.
2	SE	*Formal pattern*	$\exists <t_j$, 1, `as.destroyLink(`r_b`=>`o_1`,`r_a`=>`o_2`)`$>$, ∀ i,j
		Alf pattern	`as.destroyLink(`r_b`=>`o_1`,`r_a`=>`o_2`);`
		Description	The path includes an action to destroy the symmetric link.

Example 32 In order to illustrate how Table 6.14 operates, consider a variant of our initial class diagram (see Figure 6.26) and the operation `addSubstituteCourse2` (defined in the context of class `Course`) to add a substituting course to the `self` course.

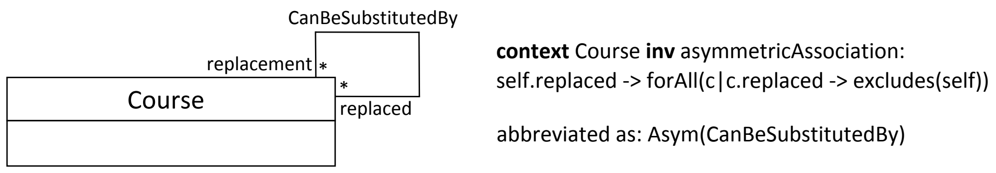

CanBeSubstitutedBy
replacement *
*
replaced
Course
context Course **inv** asymmetricAssociation:
self.replaced -> forAll(c|c.replaced -> excludes(self))
abbreviated as: Asym(CanBeSubstitutedBy)

Figure 6.26. Simplified class diagram to illustrate the use of Table 6.14.

```
activity addSubstituteCourse2(in c:  Course) {
  CanBeSubstitutedBy.createLink(replaced=>self,replacement=>c);
}
```

The operation `addSubstituteCourse2` is WE (since weak executability does not impose any condition in this case). It means that this operation will be successfully executed when the symmetric link does not exist in the system (i.e. `c` cannot be replaced by `self`).

However, this operation is not SE since it does not satisfy any of the necessary conditions for being SE (rows 1 and 2 of Table 6.14).

In order to become SE, the operation `addSubstituteCourse2` must satisfy one of the following conditions:

1. As row 1 of Table 6.14 states, the path contains a guard that prevents the execution of the PVA when `c` can be replaced by `self`. The result of adding this guard to the operation is:

```
 activity addSubstituteCourse2_row1Added(in c:  Course) {
if ( !c.replacement->includes(self) ) {
CanBeSubstitutedBy.createLink(replaced=>self,replacement=>c);
}
}
```

2. As row 2 of Table 6.14 states, the path contains an action to destroy the symmetric link. The result of adding this action to the operation is:

```
 activity addSubstituteCourse2_row2Added(in c:  Course) {
CanBeSubstitutedBy.createLink(replacement=>self,replaced=>c);
CanBeSubstitutedBy.destroyLink(replaced=>self,replacement=>c);
}
```

Table 6.15. Conditions to discard a PVA of type **CreateLink** (term <t_i, 1, `as.createLink(`r_a`=>`o_1`,`r_b`=>`o_2`)`>, where o_1`.oclIsTypeOf(cl))` when it may violate a constraint (SVC) of type irreflexive association: **Irrefl**(`as`).

	Property	*Conditions to discard the PVA*	
1	SE	*Formal pattern*	∃ <$g_{\#}$, $o_1 \neq o_2$, t_j, t_k>, 1≤j≤i≤k≤nTerms[9]

		Alf pattern	`if ( `o_1`!=`o_2` ){`
			
			`  as.createLink(`r_a`=>`o_1`,`r_b`=>`o_2`); //PVA`
			`  ...`
			`}`

		Description	The path contains a guard that prevents the execution of the PVA when the two member ends are the same object.

Example 33 In order to illustrate how Table 6.15 operates, consider a variant of our initial class diagram shown in Figure 6.27 and the operation `addSubstituteCourse3` (defined in the context of class `Course`) to add a substituting course to a course.

```
activity addSubstituteCourse3(in c:  Course) {
  CanBeSubstitutedBy.createLink(replaced=>self,replacement=>c);
}
```

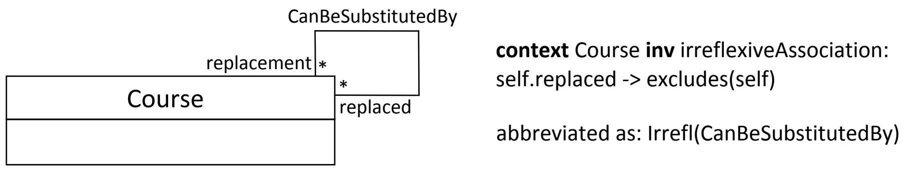

CanBeSubstitutedBy
replacement *
* replaced
Course
context Course inv irreflexiveAssociation:
self.replaced -> excludes(self)
abbreviated as: Irrefl(CanBeSubstitutedBy)

Figure 6.27. Simplified class diagram to illustrate the use of Table 6.15.

The operation `addSubstituteCourse3` is WE (since weak executability does not impose any condition in this case). It means that this operation will be successfully executed when `c` and `self` are not the same object.

However, this operation is not SE since it does not satisfy the condition for being SE (row 1 of Table 6.15).

In order to become SE, the operation `addSubstituteCourse3` should contain a guard to ensure the PVA is only executed when `c` and `self` does not refer to the same object. The result of adding this guard to the operation is:

```
activity addSubstituteCourse3_row1Added(in c:  Course) {
  if ( c != self ) {
   CanBeSubstitutedBy.createLink(replaced=>self,replacement=>c);
  }
}
```

PVA: DestroyLink: as.destroyLink(r_a=>o_1,r_b=>o_2)

This subsection shows the conditions to discard a PVA of type **DestroyLink** (term <t_i, 1, `as.destroyLink`(r_a=>o_1, r_b=>o_2)>) when it may violate a constraint (SVC) of type minimum cardinality of an association (see Table 6.16) or a symmetric association (see Table 6.17).

Example 34 In order to illustrate how Table 6.16 operates, consider a variant of our initial class diagram (see Figure 6.28) and the operation `eliminateRelatedCourse` (defined in the context of class `Menu`) to eliminate course from a menu.

```
activity eliminateRelatedCourse(in c:  Course) {
  IsComposedOf.destroyLink(menu=>self,course=>c);
}
```

The operation `eliminateRelatedCourse` is WE since it satisfies the 5th row of Table 6.16: the total number of link deletions (without considering the PVA) (p=0)

[11]The method requires a different *CreateObjectAction* (`new`) / *ReclassifyObjectAction* (`classify to`) action for each PVA.
[12]o is the multiplicity of `as.createLink`(r_a=>o_1, r_b=>x) in $t_j \ldots t_i$.
[13]p is the multiplicity of `as.destroyLink`(r_a=>o_1, r_b=>x) in $t_j \ldots t_i$.
[14]The method requires a different `destroyLink` action for each PVA.
[15]The method requires a different `createLink` action for each PVA.

Table 6.16. Conditions to discard a PVA of type **DestroyLink** (term <t_i, 1, `as.destroyLink(`r_a`=>`o_1`,`r_b`=>`o_2`)`>, where o_1`.oclIsTypeOf(cl)`) when it may violate a constraint (SVC) of type minimum cardinality of an association: **Cmin**(`as`,r_b)≠0. Note: r_b may be replaced for the opposite role of `as` (i.e. r_a).

	Property	*Conditions to discard the PVA*	
1	WE/SE	*Formal pattern*	∃ <$g_{\#}$, $o_1.r_b$→`size()`>Cmin(as,r_b)+o^{12}-p^{13}, t_j, t_k>, 1≤j≤i≤k≤nTerms[9]

		Alf pattern	`if (` $o_1.r_b$→`size()>Cmin(as,`r_b`)+o`^{12}`-p`^{13} `){`
			
			`as.destroyLink(`r_a`=>`o_1`,`r_b`=>`o_2`); //PVA`
			`...`
			`}`

		Description	The path contains a guard that prevents the execution of the PVA when `as` has already the minimum number of links.
3	WE/SE	*Formal pattern*	∃ <t_j, 1, o_1`.destroy()`>, i<j
		Alf pattern	o_1`.destroy();`
		Description	The path includes the destruction of the object o_1.
3	WE/SE	*Formal pattern*	∃ <t_j, ≥1, `classify` o_1 `from cl`>, i<j
		Alf pattern	`classify` o_1 `from cl`
		Description	The object o_1 is reclassified from `cl` to another class where `as` does not belongs to. Note that this row only applies when `cl` is part of a generalization set.
4	WE	*Formal pattern*	p^{13}-o^{12}<Cmax(`as`,r_b)-Cmin(`as`,r_b), 1≤j≤i

		Alf pattern	`as.createLink(`r_a`=>`o_1`,`r_b`=>`x_1`);`
			
			`...`
			`as.createLink(`r_a`=>`o_1`,`r_b`=>`x_o`);`
			`as.destroyLink(`r_a`=>`o_1`,`r_b`=>`y_1`);`
			`...`
			`as.destroy(`r_a`=>`o_1`,`r_b`=>`y_p`);`
			`as.destroy(`r_a`=>`o_1`,`r_b`=>`o_2`); //PVA`
			`//p`^{13}`-o`^{12}`<Cmax(as,`r_b`)-Cmin(as,`r_b`)`

		Description	The total number of link deletions minus link creations of association `as` before the PVA does not exceed the difference between the maximum and the minimum cardinalities of association `as`.

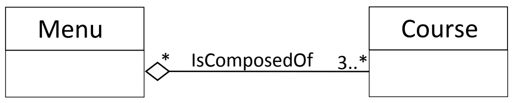


Figure 6.28. Simplified class diagram to illustrate the use of Table 6.16.

minus link creations (o=0) does not exceed the difference between the maximum (Cmax(`IsComposedOf`,`course`)=*) and the minimum (Cmin(`IsComposedOf`,`course`)=3) cardinalities of association `IsComposedOf` in the role `course`, i.e. 0-0<*-3. It means that this operation will be successfully executed when we are not close to the minimum number of links (i.e. at least three related courses). However, this operation is not SE since it does not satisfy any of the necessary conditions for being SE (rows 1 to 4 of Table 6.16).

In order to become SE, the operation `eliminateRelatedCourse` must satisfy one of the following conditions:

1. As row 1 of Table 6.16 states, the path contains a guard to ensure the PVA is only executed when the menu `self` is related with more than three courses. The result of adding this guard to the operation is:

```
activity eliminateRelatedCourse_row1Added(in c: Course) {
  if ( self.course()->size > 3 ) {
   IsComposedOf.destroyLink(menu=>self,course=>c);
  }
}
```

2. As row 2 of Table 6.16 states, the path includes an action to create a new link between the menu `self` and another course to compensate the deletion of the link. The result of adding this action to the operation is:

```
activity eliminateRelatedCourse_row2Added(in c: Course, in
c2: Course) {
  IsComposedOf.createLink(menu=>self,course=>c2);
  IsComposedOf.destroyLink(menu=>self,course=>c);
}
```

   Note also that, in this example, row 3 of Table 6.16 does not apply since the class `Menu` is not a child of a generalization set.

Example 35 In order to illustrate how Table 6.17 operates, consider a variant of our initial class diagram (see Figure 6.29) and the operation `eliminateSubstituteCourse` (defined in the context of class `Course`) to eliminate a substituting course from a course.

```
activity eliminateSubstituteCourse(in c: Course) {
  CanBeSubstitutedBy.destroyLink(replaced=>self,
   replacement=>c);
}
```

The operation `eliminateSubstituteCourse` is not WE nor SE since it does not satisfy any of the necessary conditions for being WE/SE (rows 1 to 3 of Table

Table 6.17. Conditions to discard a PVA of type **DestroyLink** (term $<t_i$, 1, as.destroyLink(r_a=>o_1,r_b=>o_2)$>$, where o_1.oclIsTypeOf(cl)) when it may violate a constraint (SVC) of type symmetric association: **Sym**(as).

	Property	*Conditions to discard the PVA*	
1	WE/SE	*Formal pattern*	$\exists$ $<t_j$, 1, as.destroyLink(r_b=>o_1,r_a=>o_2)$>$, $\forall$ i,j
		Alf pattern	as.destroyLink(r_b=>o_1,r_a=>o_2);
		Description	The path includes an action to destroy the symmetric link.
2	WE/SE	*Formal pattern*	$\exists$ $<t_j$, 1, classify o_k from (cl)$>$, $k = \{1,2\}$, i<j
		Alf pattern	classify o_k from (cl)>; //k = {1,2}
		Description	The path includes an action to reclassify from cl one of the objects which participates in the PVA. Note that this row only applies when cl is part (as child) of a generalization set.

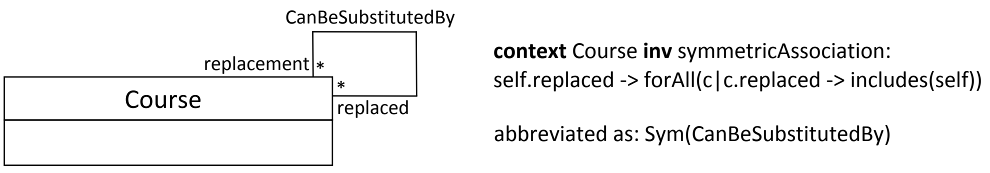

CanBeSubstitutedBy
replacement
*
*
replaced
Course
context Course inv symmetricAssociation:
self.replaced -> forAll(c|c.replaced -> includes(self))
abbreviated as: Sym(CanBeSubstitutedBy)

Figure 6.29. Simplified class diagram to illustrate the use of Table 6.17.

6.17). Then, after the execution of this operation we always reach a inconsistent state of the system where the course c can be substituted by self but self cannot be substituted by c.

In order to become SE, the operation eliminateRelatedCourse must satisfy one of the following conditions:

1. As row 1 of Table 6.17 states, the path includes an action to also destroy the symmetric link. The result of adding this action to the operation is:

```
activity eliminateSubstituteCourse_row1Added(in c:  Course) {
 CanBeSubstitutedBy.destroyLink(replaced=>self,
  replacement=>c);
 CanBeSubstitutedBy.destroyLink(replacement=>self,
  replaced=>c);
}
```

   Note also that, in this example, row 2 of Table 6.17 does not apply since the class Menu is not a child of a generalization set.

PVA: ClearAssociation: as.clearAssoc(o_1)

This subsection shows the conditions to discard a PVA of type **ClearAssociation** (term <t_i, 1, `as.clearAssoc(as)`>) when it may violate a constraint (SVC) of type minimum cardinality of an association (see Table 6.18).

Table 6.18. Conditions to discard a PVA of type **ClearAssociation** (term <t_i, 1, `as.clearAssoc(`o_1`)`>, where o_1`.oclIsTypeOf(cl)`) when it may violate a constraint (SVC) of type minimum cardinality of an association: **Cmin**(`as`,r_b)≠0. Note: r_b may be replaced for the opposite role of `as` (i.e. r_a).

	Property	*Conditions to discard the PVA*	
1	WE/SE	*Formal pattern*	∃ <t_j, ≥Cmin(`as`,r_b), `as.createLink(`r_a=>o_1`,`r_b=>x`)`[15]>, i<j

		Alf pattern	`o = new cl(); //PVA`
			
			`...`
			`for (i in 1..Cmin(as,`r_b`)){`
			 `...`
			 `as.createLink(`r_a=>o_1`,`r_b=>x`)`[15]`;`
			 `...`
			`}`

		Description	The path includes at least Cmin(`as`,r_b) actions which create a link of `as` between the object o_1 (with role r_a) and another obect (with role r_b).
2	WE/SE	*Formal pattern*	∃ <t_j, ≥1, `classify` o_1 `from cl`>, i<j
		Alf pattern	`classify` o_1 `from cl;`
		Description	The path includes an action to reclassify o_1 from `cl` to another class where `as` does not belongs to. Note that this row only applies when `cl` is part (as child) of a generalization set.

Example 36 In order to illustrate how Table 6.18 operates, consider a variant of our initial class diagram (see Figure 6.30) and the operation `clearCourses` (defined in the context of class `Menu`) to clear all the courses related with a menu.

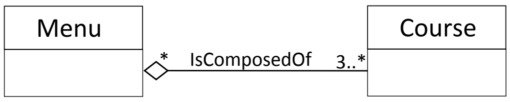


Figure 6.30. Simplified class diagram to illustrate the use of Table 6.18.

```
activity clearCourses() {
  IsComposedOf.clearAssoc(self);
}
```

The operation `clearCourses` is not WE nor SE since it does not satisfy any of the necessary conditions for being WE/SE (rows 1 to 3 of Table 6.18). Then, after the execution of this operation we always reach a inconsistent state of the system where the menu `self` is not composed of any course.

In order to become WE/SE, the operation `clearCourses` must satisfy one of the following conditions:

1. As row 1 of Table 6.18 states, the path includes the creation of at least Cmin(`IsComposedOf`,`course`) links of `IsComposedOf` between the menu `self` and any courses. The result of adding this action to the operation is:

   ```
   activity clearCourses_row1Added(in _courses:  Courses[3..*])
   {
     IsComposedOf.clearAssoc(self);
     for ( i in 1.._courses->size() ) {
      IsComposedOf.createLink(menu->self,course->_courses[i]);
     }
   }
   ```

   Note also that, in this example, row 2 of Table 6.18 does not apply since the class `Menu` is not a child of a generalization set.

PVA: ReclassifyObject: classify o from cl_r

This subsection shows the conditions to discard a PVA of type **ReclassifyObject** (term $<t_i$, 1, `classify o from` $cl_r>$) when it may violate a constraint (SVC) of type covering (see Table 6.19) or of type referential (see Table 6.20). Note that, when an object is reclassified to a new subclass, the constraint minimum cardinality of a class may also be violated. Then, the conditions shown in Table 6.8 must also be satisfied.

Example 37 In order to illustrate how Table 6.19 operates, consider a variant of our initial class diagram (see Figure 6.31) and the operation `eliminateFromSpecialMenu` (defined in the context of class `Menu`) to take off a menu from the subclass `SpecialMenu`.

Table 6.19. Conditions to discard a PVA of type **ReclassifyObject** (term $<t_i$, 1, `classify o from` $cl_r>$, where `cl` generalizes $cl_1,\ldots,cl_n$) when it may violate a constraint (SVC) of type covering: **Cov**(`cl`,{cl_1,..., cl_n}).

	Property	*Conditions to discard the PVA*	
1	SE	*Formal pattern*	$\exists <g_\#$, `o.oclIsTypeOf(`cl_s`)`, t_j, $t_k>$, $1\leq j\leq i\leq k\leq$nTerms[9], s=1,... n, s≠r

		Alf pattern	`if ( o.oclIsTypeOf(`cl_s`) ){ //s=1,...n, s≠r`
			 `classify o from` cl_r`; //PVA`
			 `...`
			 `}`

		Description	The path contains a guard that prevents the execution of the PVA when the object `o` belongs to another subclass.
2	SE	*Formal pattern*	$\exists <t_j$, ≥1, `classify o to` cl_s)>, cl_s=cl_1,... cl_n (or one of its subclasses), s≠r
		Alf pattern	`classify o to` cl_s`;`
		Description	The path includes an action to reclassify the object `o` to another subclass.

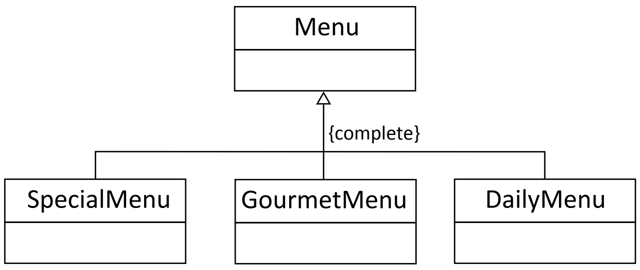


Figure 6.31. Simplified class diagram to illustrate the use of Table 6.19.

```
activity eliminateFromSpecialMenu() {
  classify self from SpecialMenu;
}
```

The operation `eliminateFromSpecialMenu` is WE (since weak executability does not impose any condition in this case). It means that this operation will be successfully executed when `self` belongs to more subclasses.

However, this operation is not SE since it does not satisfy any of the necessary conditions for being SE (rows 1 and 2 of Table 6.19).

In order to become SE, the operation `eliminateFromSpecialMenu` must satisfy one of the following conditions:

1. As row 1 of Table 6.19 states, the path includes a guard to ensure the PVA is only executed when the menu `self` belongs to another subclass. The result of adding this guard to the operation is:

```
activity eliminateFromSpecialMenu_row1Added() {
  if ( self.oclIsTypeOf(GourmetMenu) OR
   self.oclIsTypeOf(DailyMenu) ) {
    classify self from SpecialMenu;
  }
}
```

2. As row 2 of Table 6.19 states, the path includes an action to reclassify the menu `self` to another subclass. The result of adding this action to the operation is, for instance:

```
activity eliminateFromSpecialMenu_row2Added() {
  classify self from SpecialMenu to DailyMenu;
}
```

Note that the designer could classify `self` to `GourmetMenu` instead of `DailyMenu`.

Table 6.20. Conditions to discard a PVA of type **ReclassifyObject** (term $<t_i$, 1, `classify o from` [cl_r]$>$, where `cl` generalizes $cl_1,\ldots,cl_n$) when it may violate a constraint (SVC) of type referential: **Referential**(cl_r,`as`).

	Property	*Conditions to discard the PVA*	
1	WE/SE	*Formal pattern*	$\exists <g_{\#}$, `o.`r_a`→isEmpty()`, t_j, $t_k>$, where r_a is the member end of `as` opposed to cl_r

		Alf pattern	`if ( o.`r_a`→isEmpty() ){`
			
			`classify o from` cl_r`; //PVA`
			`...`
			`}`

		Description	The path contains a guard that prevents the execution of the PVA when the object `o` does not participate in the association `as`.
2	WE/SE	*Formal pattern*	$\exists <g_{\#}$, `o.`r_a`→notEmpty()`, t_j, $t_k>$, where r_a is the member end of *as* opposed to *cl* **AND** $\exists <t_l$, 1, `as.clearAssoc(o)`$>$, 1≤j≤l≤k≤i≤nTerms[9]

		Alf pattern	`if ( o.`r_a`→notEmpty() ){`
			
			`as.clearAssoc(o);`
			`classify o from` cl_r`; //PVA`
			`...`
			`}`

		Description	If the object `o` participates in the association `as`, all the existing links are destroyed before the execution of the PVA.
3	WE	*Formal pattern*	-
		Alf pattern	-
		Description	The minimum cardinality of the association `as` in the role `r` (where `r` is the opposite role to the class cl_r), is zero (i.e. (Cmin(`as`,r)=0)).

Example 38 In order to illustrate how Table 6.20 operates, consider a variant of our initial class diagram (see Figure 6.32) and the operation `eliminateFromGourmetMenu` (defined in the context of class `Menu`) to take off a menu from the subclass special menu.

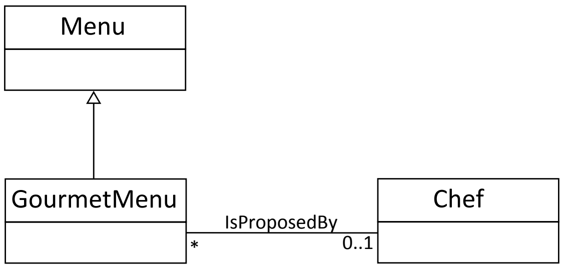


Figure 6.32. Simplified class diagram to illustrate the use of Table 6.20.

```
activity eliminateFromGourmetMenu() {
  classify self from GourometMenu;
}
```

The operation `eliminateFromGourmetMenu` is WE since it satisfies the 3th row of Table 6.20: the minimum cardinality of the association `IsProposedBy` in the role `chef` is zero (Cmin(`IsProposedBy`,`chef`)=0). It means that this operation will be successfully executed when the gourmet menu `self` does not have any chef assigned.

However, this operation is not SE since it does not satisfy any of the necessary conditions for being SE (rows 1 and 2 of Table 6.20).

In order to become SE, the operation `eliminateFromGourmetMenu` must satisfy one of the following conditions:

1. As row 1 of Table 6.20 states, the path includes a guard to ensure the PVA is only executed when the menu `self` is not related to any chef. The result of adding this guard to the operation is:

```
activity eliminateFromGourmetMenu_row1Added() {
  if ( self.chef->isEmpty() ) {
   classify self from GourometMenu;
  }
}
```

2. As row 2 of Table 6.20 states, if the menu `self` is related to one chef, the path includes an action to destroy the existing link. The result of adding this action to the operation is, for instance:

```
activity eliminateFromGourmetMenu_row2Added() {
  IsProposedBy.clearAssoc(self);
  classify self from GourometMenu;
}
```

PVA: ReclassifyObject: classify o to cl_r

This subsection shows the conditions to discard a PVA of type **ReclassifyObject** (term <t_i, 1, `classify o to` cl_r>) when it may violate a constraint (SVC) of type disjointness (see Table 6.21) or covering (see Table 6.22). Note that, when an object is reclassified to a new subclass, the constraints maximum cardinality of a class, the mandatory of an attribute and the minimum cardinality of an association may also be violated. Then, the conditions shown in Tables 6.5, 6.6 and 6.7 must also be satisfied.

Table 6.21. Conditions to discard a PVA of type *ReclassifyObject* (term <t_i, 1, `classify o to [`cl_r`]`>, where `cl` generalizes cl_1,...,cl_n) when it may violate a constraint (SVC) of type disjointness: **Disj**(cl,{cl_1,..., cl_n}).

	Property	*Conditions to discard the PVA*	
1	SE	*Formal pattern*	∃ <$g_{\#}$, `!o.oclIsTypeOf(`cl_s`)`, t_j, t_k>, 1≤j≤i≤k≤nTerms[9], ∀ s=1...n, s≠r

		Alf pattern	`if ( !o.oclIsTypeOf(`cl_s`) ){ //∀ s=1...n, s≠r`
			` ...`
			` classify o to `cl_r`; //PVA`
			` ...`
			`}`

		Description	The path contains a guard that prevents the execution of the PVA when the object o belongs to another subclass.
2	SE	*Formal pattern*	∃ <t_j, 1, `classify o from` cl_s)>, cl_s=cl_1,...cl_s (or one of its subclasses), s≠r
		Alf pattern	`classify o from `cl_s`; //`cl_s=cl_1`,...`cl_s` (or one of its subclasses), s≠r`
		Description	The path includes an action to reclassify the object o from any other subclass.

Example 39 In order to illustrate how Table 6.21 operates, consider a variant of our initial class diagram (see Figure 6.33) and the operation `classifyAsGourmetMenu` (defined in the context of class `Menu`) to classify an existing menu as gourmet menu.

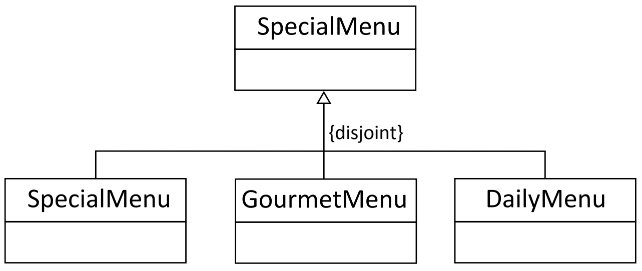


Figure 6.33. Simplified class diagram to illustrate the use of Table 6.21.

```
activity classifyAsGourmetMenu() {
  classify self to GourmetMenu;
}
```

The operation `classifyAsGourmetMenu` is WE (since weak executability does not impose any condition in this case). It means that this operation will be successfully executed when `self` does not belongs to another subclass.

However, this operation is not SE since it does not satisfy any of the necessary conditions for being SE (rows 1 and 2 of Table 6.21).

In order to become SE, the operation `classifyAsGourmetMenu` must satisfy one of the following conditions:

1. As row 1 of Table 6.21 states, the path includes a guard to ensure the PVA is only executed when the menu `self` does not belongs to another subclass. The result of adding this guard to the operation is:

```
activity classifyAsGourmetMenu_row1Added() {
  if ( !self.oclIsTypeOf(SpecialMenu) and
  !self.oclIsTypeOf(DailyMenu) ) {
   classify self to GourmetMenu;
  }
}
```

2. As row 2 of Table 6.21 states, the path includes an action to reclassify the menu `self` from any other subclass. The result of adding this action to the operation is:

```
activity classifyAsGourmetMenu_row2Added() {
  classify self from self.oclIsTypeOf(self.oclType()[16]);
  classify self to GourmetMenu;
}
```

Table 6.22. Conditions to discard a PVA of type **ReclassifyObject** (term $<t_i$, 1, `classify o to` $cl_r>$, where `cl` generalizes $cl_1,\ldots,cl_n$) when it may violate a constraint (SVC) of type covering: **Cov**(cl_r,{cl_{r_1},..., cl_{r_n}}).

	Property	*Conditions to discard the PVA*	
1	WE/SE	*Formal pattern*	$\exists <t_j$, 1, `classify o to` $cl_{r_w})>$, $cl_{r_w}=cl_{r_1},\ldots,cl_{r_w}$ (or one of its subclasses)
		Alf pattern	`classify o to` cl_{r_w}`; //`$cl_{r_w}=cl_{r_1}$`, ...,`cl_{r_w} `(or one of its subclasses)`
		Description	The path includes an action to reclassify the object `o` to a subclass of cl_r.

Example 40 In order to illustrate how Table 6.22 operates, consider a variant of our initial class diagram (see Figure 6.34) and the operation

[16]Despite the function `oclType():Classifier` is included in the latest versions of the OCL standard, this function is not well defined, since in UML an instance may have several types (and classifiers) but the function return just one classifier. In this thesis we assume this function returns the most specialized classifier.

classifyAsDailyMenu (defined in the context of class Menu) to classify an existing menu as daily menu.

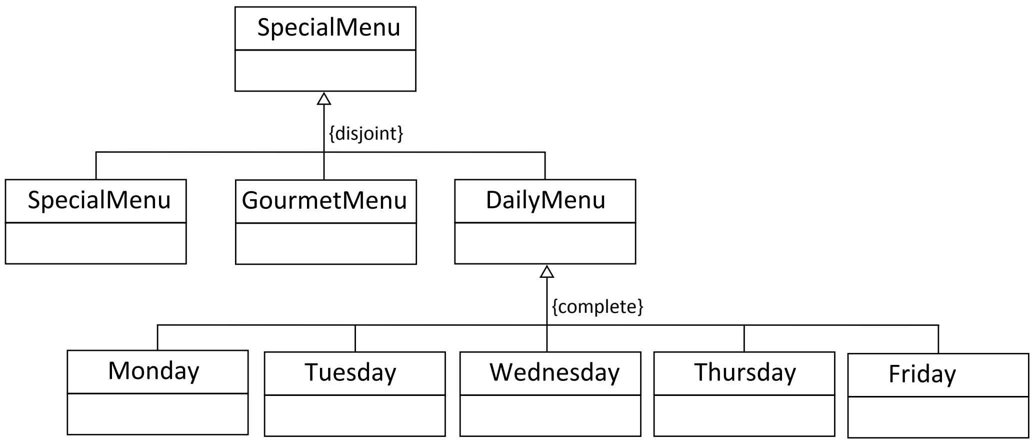


Figure 6.34. Simplified class diagram to illustrate the use of Table 6.22.

```
activity classifyAsDailyMenu() {
  classify self to DailyMenu;
}
```

The operation `classifyAsDailyMenu` is not WE nor SE since it does not satisfy the necessary condition for being WE/SE (row 1 of Table 6.22). Then, after the execution of this operation we always reach a inconsistent state of the system where the menu `self` is not classified to any subclass of `DailyMenu`.

In order to become SE, the operation `classifyAsDailyMenu` must includee an action to reclassify the menu `self` to a subclass of `DailyMenu`. The result of adding this action to the operation is, for instance:

```
activity classifyAsDailyMenu_row1Added() {
  classify self to DailyMenu;
  classify self to Firday;
}
```

Note that the designer could classify `self` to another subclass (`Monday` to `Thursday`) instead of `Friday`.

6.3.3 Step 3: Classifying the operation

Last step of our method classifies the operation depending on the results obtained in the previous step regarding each execution path of the operation.

If at least one of the execution paths of the operation is WE, the operation is classified as WE. If all its execution paths are SE, the operation is classified as SE. Otherwise, the operation is classified as non-executable. See an example of this step in Section 6.3.4.

6.3.4 Feedback

Besides determining the executability of an operation, a distinguishing feature of our method is that for non-WE/SE operations it returns valuable information to help designers identifying and correcting the detected errors. This feedback information is expressed in terms of the operation itself so it can be easily understood and processed by the designer. For non-WE/SE operations, our method provides two kinds of informations.

First, the returned feedback identifies *why* the operation is not WE/SE. For each non-WE/SE path our method provides the list of PVAs that could eventually induce a violation of the integrity constraints together with the specific list of SVCs that those PVAs could violate.

Second, the returned feedback explains *how* the designer may fix these (potentially) violating scenarios by providing a set of possible repair alternatives that should be included in the non-WE/SE execution paths of the operation. These alternatives are expressed as a finite set of *terms* and *guards* to be added to the path and correspond to the rows (of the tables of the Section 6.3.2) of those PVAs that cannot be discarded. The designer should choose the most appropriate alternative in her context.

These repair alternatives can be automatically integrated in the operation body by following two rules:

- Each suggested guard g=<g_i,$condition_i$,t_{ini_i},t_{end_i}> is translated as a new conditional structure with condition $condition_i$, starting before t_{ini_i} and ending after t_{end_i}.
- Each suggested term t=<t_i,$multiplicity_i$,$action_i$> is translated as a new action in the operation. If the term has multiplicity "1", the action is added once. Otherwise, the action is included inside a loop that is executed $multiplicity_i$ times.

6.3.5 Example of use

This section illustrates the complete process our lightweight and static method carries out in order to verify the weak and strong executability of an input Alf-based operation.

In order to show how our method internally works, we use three operations (`newCourse`, `addMenu` and `classifyAsSpecialMenu`) as example:

Operation `newCourse` creates a new course in the system.

```
activity newCourse(in _description:  String,
in _substitutingCourses:Course[*]) {
  Course c = new Course();
  c.description = _description;
  for ( i in 1.._substitutingCourses→size() ) {
   CanBeSubstitutedBy.createLink(replaced=>c,replacement=>_substituting-
    Courses[i]);
  }
 }
```

Operation addMenu adds a new menu to the system.

```
activity addMenu(in _name:  String, in _price:  Real, in _courses:
Course[3..*]) {
  if ( !Menu.allInstances()→exists(m|m.name=_name) ) {
   Menu m = new Menu();
   m.name = _name;
   m.price = _price;
   for ( i in 1.._courses→size() ) {
    IsComposedOf.createLink(menu=>m,course=>_courses[i]);
   }
  }
}
```

Operation classifyAsSpecialMenu classifies a menu as a special menu.

```
activity classifyAsSpecialMenu(in _discount:  Real) {
  if ( _discount ≥ 10 ) {
   classify self to SpecialMenu;
   self.discount = _discount;
  }
}
```

Step 0: Computing the execution paths

Prior to check the weak/strong executability of an operation, our method performs a preprocessing step to compute its execution paths (see section 6.1).

Figures 6.35, 6.36 and 6.37 show the MBCFGs (Model-Based Control Flow Graph) for our running operations.

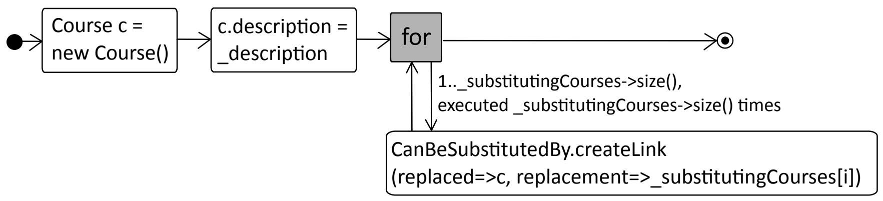

Course c =
new Course()
c.description =
_description
for
1.._substitutingCourses->size(),
executed _substitutingCourses->size() times
CanBeSubstitutedBy.createLink
(replaced=>c, replacement=>_substitutingCourses[i])

Figure 6.35. MBCFG of newCourse operation.

Operation newCourse has two execution paths: $p1_{newCourse}$ is the sequence of actions executed when the new course does not have substituting courses, and $p2_{newCourse}$ (as g_1 states) executed otherwise. Actions included in the loop have *multiplicity*= _substitutingCourses→size(), given that they will be executed exactly _substitutingCourses→size() times (as stated by the loop condition).

Execution paths for operation `newCourse`:
$p1_{newCourse}$ = {
{ <t_1, 1, `c = new Course()`>,
<t_2, 1, `c.description = _description`>}, {} }
$p2_{newCourse}$ = {
{ <t_1, 1, `c = new Course()`>,
<t_2, 1, `c.description = _description`>,
<t_3, `_substitutingCourses→size()`, `CanBeSubstitutedBy.createLink(replaced=>c, replacement=>_substitutingCourses[i])`>},
{<g_1, `_substitutingCourses→size()≥1`, t_3, t_3>} }

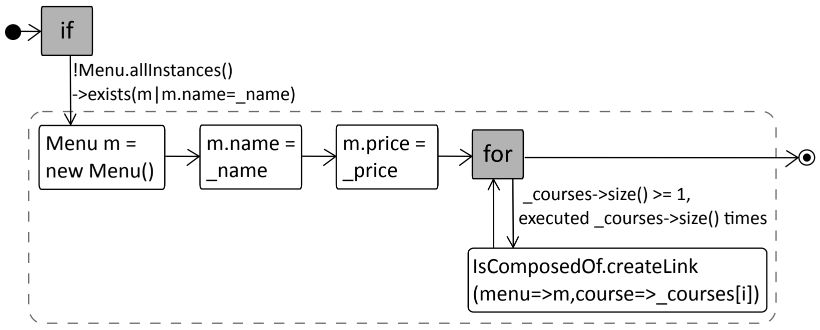


Figure 6.36. MBCFG of `addMenu` operation.

Operation `addMenu` has also two execution paths: $p1_{addMenu}$ is the sequence of actions executed when the new menu does not contain any course, and $p2_{addMenu}$ (as g_1 states) executed otherwise. Note that we automatically discard the empty path.

Execution path for operation `addMenu`:
$p1_{addMenu}$ = {
{ <t_1, 1, `m = new Menu()`>,
<t_2, 1, `m.name = _name`>,
<t_3, 1, `m.price = _price`> },
{<g_1, `!Menu.allInstances()→exists(m|m.name=_name)`, t_1, t_3>} }
$p2_{addMenu}$ = {
{ <t_1, 1, `m = new Menu()`>,
<t_2, 1, `m.name = _name`>,
<t_3, 1, `m.price = _price`>,
<t_4, 1, `IsComposedOf.createLink(menu=>m, course=>_courses[i])`> },
{<g_1, `!Menu.allInstances()→exists(m|m.name=_name)`, t_1, t_4>,
<g_2, `_courses→size()≥1`, t_4, t_4>} }

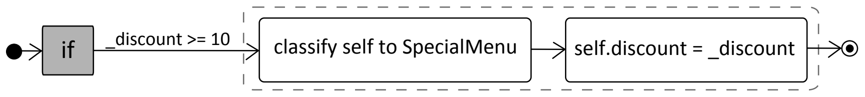


Figure 6.37. MBCFG of `classifyAsSpecialMenu` operation.

Finally, operation `classifyAsSpecialMenu` has a single execution path.

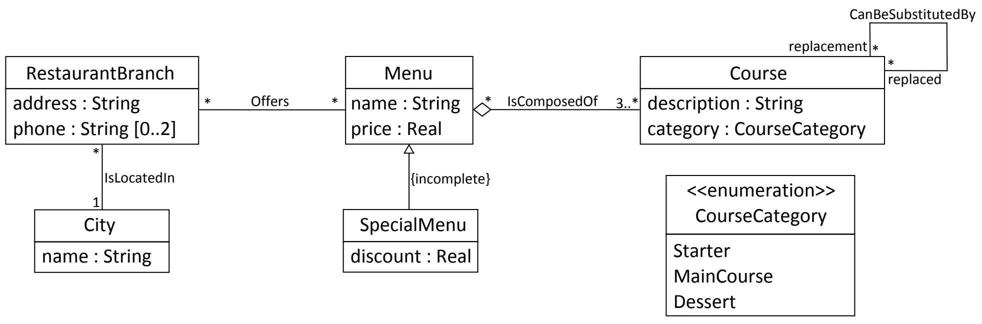
Execution paths for operation `classifyAsSpecialMenu`:
$p_{classifyAsSpecialMenu} = \{$
{ <t_1, 1, `classify self to SpecialMenu`>,
<t_2, 1, `self.discount = _discount`>},
{<g_1, `_discount`≥`10`, t_1, t_2>} }

Once the execution paths have been computed, Steps 1 and 2 of our method are applied on each path until we recognize a WE path (in case of verifying weak executability) or until we check all paths are SE (in case of verifying strong executability).

Step 1: Analyzing the existence of Potentially Violating Actions

Step 1 of our method (see Section 6.3.1) analyzes individually each action in the path p to see whether it may violate some integrity constraints of the structural model.

Applying the rules of Table 6.1 to the whole class diagram of Figure 6.38 we obtain the set of possible PVAs derived from the structural model. Table 6.23 shows these PVAs. For each SVC, we show the set of PVAs that may violate it (sharp sign (#) represent irrelevant variables and consecutive letters (x, y,...) represent free variables that may be bound to any value in the term).



context Menu **inv** menuPrimaryKey: Menu.allInstances()->isUnique(name)	ID(name,Menu)
context SpecialMenu **inv** atMost3SpecialMenus: SpecialMenu.allInstances()->size()<=3	Cmax(SpecialMenu)=3
context SpecialMenu **inv** validDiscount: self.discount >= 10	ValueComp(self.discount,>=,10)
context Course **inv** symmetricAssociation: self.replaced -> forAll(c|c.replaced -> includes(self))	Sym(CanBeSubstitutedBy)

Figure 6.38. Excerpt of a restaurant chain class diagram.

Table 6.23. PVAs of class diagram of Figure 6.38 according to Table 6.1.

SVCs	*PVAs*

Cmax(`SpecialMenu`)=3	`x = new SpecialMenu()`
	`classify x to SpecialMenu`

Mand(`name`, `City`)	`x = new City()`
	`x.name = null`, where `x.oclIsTypeOf(City)`

Mand(`address`, `RestaurantBranch`)	`x = new RestaurantBranch()` `x.address = null`, where `x.oclIsTypeOf(RestaurantBranch)`

Mand(`name`, `Menu`)	`x = new Menu()`
	`x = new SpecialMenu()`
	`x.name = null`, where `x.oclIsTypeOf(Menu)`

Mand(`price`, `Menu`)	`x = new Menu()`
	`x = new SpecialMenu()`
	`x.price = null`, where `x.oclIsTypeOf(Menu)`

Mand(`discount`, `SpecialMenu`)	`x = new SpecialMenu()`
	
	`classify x to SpecialMenu`
	`x.discount = null`, where `x.oclIsTypeOf(SpecialMenu)`

Mand(`description`, `Course`)	`x = new Course()`
Mand(`category`, `Course`)	`x = new Course()`

Cmin(`IsLocatedIn`, `city`)=1	`x = new RestaurantBranch()`
	
	`IsLocatedIn.destroyLink(restaurantBranch=>x, city=>y)`
	`IsLocatedIn.clearAssoc(x)`,
	where `x.oclIsTypeOf(RestaurantBranch)`

Cmin(`IsComposedOf`, `course`)=3	`x = new Menu()`
	
	`x = new SpecialMenu()`
	`IsComposedOf.destroyLink(menu=>x, course=>y)`
	`IsComposedOf.clearAssoc(x)`,
	where `x.oclIsTypeOf(Menu)`

Cmax(`IsLocatedIn`, `city`)=1	`IsLocatedIn.createLink(restaurantBranch=>x, city=>y)`
ID(`name`, `Menu`)	`x.name = #`, where `x.oclIsTypeOf(Menu)`

Sym(`CanBeSubstitutedBy`)	`CanBeSubstitutedBy.createLink(replaced=>x, replacement=>y)`
	`CanBeSubstitutedBy.destroyLink(replaced=>x, replacement=>y)`

ValueComp (`self.discount`,>,10)	`x.discount = #`, where `x.oclIsTypeOf(SpecialMenu)`

Intersecting the set of PVAs shown in Table 6.23 with the actions that appear in the terms of the execution paths of our running operations, we obtain the PVAs for each path.

In the following we show the PVAs for the two execution paths of operation `newCourse`. First path ($p1_{newCourse}$) contains one PVA: `c = new Course()` (PVA_1), which may violate two mandatory constraints (when the attributes `description` and `category` are not initialized). Second path ($p2_{newCourse}$), in addition to the above, contains another PVA: `CanBeSubstitutedBy.createLink(replaced=>c,replacement=>_substituting-Courses[i])` (PVA_2), which may violate the `symmetricAssociation` constraint (when the opposite link is not created).

Potentially Violating Actions (PVAs) of path $p1_{newCourse}$ and Susceptible Violating Constraints (SVC) they may violate:
- PVA_1: `c = new Course()`
  $SVC_{1.1}$: Mand(`description`, `Course`)
  $SVC_{1.2}$: Mand(`category`, `Course`)

Potentially Violating Actions (PVAs) of path $p2_{newCourse}$ and Susceptible Violating Constraints (SVC) they may violate:
- PVA_1: `c = new Course()`
  $SVC_{1.1}$: Mand(`description`, `Course`)
  $SVC_{1.2}$: Mand(`category`, `Course`)
- PVA_2: `CanBeSubstitutedBy.createLink(replaced=>c,replacement=>_substitu-tingCourses[i])`
  $SVC_{2.1}$: Sym(`CanBeSubstitutedBy`)

Similarly, first path of `addMenu` ($p1_{addMenu}$) contains two PVAs. The first PVA, `m = new Menu()` (PVA_1), may violate two mandatory constraints (when the attributes `name` and `price` are not initialized) and one minimum cardinality constraint of the association `IsComposedOf` (when the new menu contains less than three courses). The second PVA, `m.name = _name` (PVA_2), may violate the identifier constraint ID(`name`,`Menu`) (when the system state contains another menu with the same name). Second path of `addMenu` ($p1_{addMenu}$) contains exactly the same PVAs.

Potentially Violating Actions (PVAs) of path $p1_{addMenu}$ and Susceptible Violating Constraints (SVC) they may violate:
- PVA_1: `m = new Menu()`
  $SVC_{1.1}$: Mand(`name`, `Menu`)
  $SVC_{1.2}$: Mand(`price`, `Menu`)
  $SVC_{1.3}$: Cmin(`IsComposedOf`,`course`)=3
- PVA_2: `m.name = _name`
  $SVC_{2.1}$: ID(`name`, `Menu`)

Potentially Violating Actions (PVAs) of path $p2_{addMenu}$ and Susceptible Violating Constraints (SVC) they may violate:

- PVA_1: `m = new Menu()`
  - $SVC_{1.1}$: Mand(`name`, `Menu`)
  - $SVC_{1.2}$: Mand(`price`, `Menu`)
  - $SVC_{1.3}$: Cmin(`IsComposedOf`,`course`)=3
- PVA_2: `m.name = _name`
  - $SVC_{2.1}$: ID(`name`, `Menu`)

Finally, the single path of `classifyAsSpecialMenu` contains two PVAs. First PVA, `classify self to SpecialMenu` (PVA_1), may violate a mandatory constraint (when the attribute `discount` is not initialized) and a maximum cardinality constraint of class `SpecialMenu` (when the system state contains already three special menus). Second PVA, `self.discount = _discount` (PVA_2), may violate the `validDiscount` constraint (when `self.discount<10`).

Potentially Violating Actions of path $p_{classifyAsSpecialMenu}$ and Susceptible Violating Constraints (SVC) they may violate:

- PVA_1: `classify self to SpecialMenu`
  - $SVC_{1.1}$: Mand(`discount`,`SpecialMenu`)
  - $SVC_{1.2}$: Cmax(`SpecialMenu`)=3
- PVA_2: `self.discount = _discount`
  - $SVC_{2.1}$: ValueComp(`self.discount`, >=, `10`)

Since all paths of our running operations are susceptible to be non-WE and non-SE (given that all of them contain some PVAs that may affect its executability) we must proceed with the second step of our method.

Step 2: Discarding the Potentially Violating Actions

Step 2 of our method (see Section 6.3.2) performs a contextual analysis of each potentially violating action to see whether other actions or conditions in p compensate or complement its effect to ensure that we sometimes/always reach a consistent state at the end of the operation execution. If all potential violation actions can be discarded we can conclude that p is WE/SE.

In order to illustrate the full usage of the tables shown in Section 6.3.2, in the following we try to discard the PVAs identified during the previous step. For each PVA and SVC of each path, we identify the conditions (table) that the path must satisfy to discard that PVA (for the sake of simplicity, we only show the formal pattern; the equivalent Alf and textual patterns can be found on the tables of Section 6.3.2). Before each condition we indicate whether it must hold when verifying weak executability (WE) or both weak and strong executability (WE/SE). Then, $\{sat\ by\ t_i\}$ or $\{sat\ by\ g_i\}$ states that the condition is satisfied by the term/guard i in the path, while $\{not\ sat\}$ states the opposite.

As we justified in the first step, first path of `newCourse` operation ($p1_{newCourse}$) has one

PVA. According to Table 6.6, in order to discard the PVA_1 when it may violate the $SVC_{1.1}$, the path must include, after the PVA, at least one action to initialize the attribute `description`. The second term of $p1_{newCourse}$ (<t_2, 1, `c.description = _description`>) contains this initialization, then, we can ensure the PVA_1 will never violate the $SVC_{1.1}$. Similarly, in order to discard the PVA_1 when it may violate the $SVC_{1.2}$, the path must include, after the PVA, at least one action to initialize the attribute `category`. There is no term in $p1_{newCourse}$ which contains this initialization. Then, the PVA_1 will always violate the $SVC_{1.2}$ and, consequently, PVA_1 cannot be discarded. Hence, our method concludes $p1_{newCourse}$ is not WE/SE.

Conditions to discard the PVAs of path $p1_{newCourse}$:

- PVA_1 (`c = new Course()`), $SVC_{1.1}$ (Mand(`description`,`Course`)):
  Table 6.6: (WE/SE) ∃ <t_j, ≥1, `c.description = #`>
  {*sat by* t_2}

- PVA_1 (`c = new Course()`), $SVC_{1.2}$ (Mand(`category`,`Course`)):
  Table 6.6: (WE/SE) ∃ <t_j, ≥1, `c.category = #`>
  {*not sat*}

Besides the above PVA, second path of `newCourse` operation ($p2_{newCourse}$) has another PVA. According to Table 6.13, in order to discard the PVA_2 when it may violate the $SVC_{2.1}$, the path must include the creation of the symmetric link. There is no term in $p2_{newCourse}$ which contains this link creation. Then, the PVA_2 will always violate the $SVC_{2.1}$ and, consequently, PVA_2 cannot be discarded. Since $p2_{newCourse}$ contains two PVAs that cannot be discarded, our method concludes $p2_{newCourse}$ is neither WE/SE.

Paths $p2_{newCourse}$ and $p1_{newCourse}$ (which is a subset of the former) do not satisfy all the required conditions to be WE/SE, hence, our method concludes these paths are not WE/SE.

Conditions to discard the PVAs of path $p2_{newCourse}$:

- PVA_1 (`c = new Course()`), $SVC_{1.1}$ (Mand(`description`,`Course`)):
  Table 6.6: (WE/SE) ∃ <t_j, ≥1, `c.description = #`>
  {*sat by* t_2}

- PVA_1 (`c = new Course()`), $SVC_{1.2}$ (Mand(`category`,`Course`)):
  Table 6.6: (WE/SE) ∃ <t_j, ≥1, `c.category = #`>
  {*not sat*}

- PVA_2 (`CanBeSubstitutedBy.createLink(replaced=>c,replacement=> _substitutingCourses[i])`), $SVC_{2.1}$ (Sym(`CanBeSubstitutedBy`)):
  Table 6.13: (WE/SE) ∃ <t_j, 1, `CanBeSubstitutedBy.createLink (replacement=>c, replaced=>_substitutingCourses[i])`>
  {*not sat*}

As we justified in the first step, second path of `addMenu` operation ($p2_{addMenu}$) ($p1_{addMenu}$ is not explicitly shown since it is a subset of this one) has two PVAs. According to Table 6.6, in order to discard the PVA_1 when it may violate the constraints $SVC_{1.1}$ and $SVC_{1.2}$, the path must include, after the PVA, at least one action to initialize the attributes `description` and `price`. The second (<t_2, 1, `m.name = _name`>) and third (<t_3, 1, `m.price = _price`>) terms of this path contain these initializations, then, we can ensure the PVA_1 will never violate the constraints $SVC_{1.1}$ nor $SVC_{1.2}$.

Besides, according to Table 6.7, in order to discard the PVA_1 when it may violate the $SVC_{1.3}$, the path must include, after the PVA, at least three actions to create a link of `IsComposedOf` between `m` and any course. This condition is satisfied by the fourth term of the path (<t_4, 1, `IsComposedOf.createLink(menu=>m,course=>_courses[i])`>) iff the condition `_courses→size()≥3` is true. Our lightweight method cannot solve this inequality, hence, it suppose it is not true (returning a false positive). However, if the user intervenes during this step, she may easily conclude the above inequality is always true, since the multiplicity of the input parameter `_courses` is at least 3. Then, PVA_1 may be actually discarded when verifying both weak and strong executability.

Finally, the PVA_2 may violate the $SVC_{2.1}$ only when verifying whether the path is SE. According to Table 6.10 there are three alternatives to discard this PVA. The first alternative is satisfied by the first guard of the path (<g_1, `!Menu.allInstances()→exists(m|m.name=_name)`, t_1, t_4>), then, PVA_2 may be discarded.

Since both PVAs may be discarded, this path is WE/SE.

Conditions to discard the PVAs of path $p2_{addMenu}$:

- PVA_1 (`m = new Menu()`), $SVC_{1.1}$ (Mand(`name`,`Menu`)):
  Table 6.6: (WE/SE) ∃ <t_j, ≥1, `m.name = #`>
  {*sat by* t_2}

- PVA_1 (`m = new Menu()`), $SVC_{1.2}$ (Mand(`price`,`Menu`)):
  Table 6.6: (WE/SE) ∃ <t_j, ≥1, `m.price = #`>
  {*sat by* t_3}

- PVA_1 (`m = new Menu()`), $SVC_{1.3}$ (Cmin(`IsComposedOf`,`course`)=3):
  Table 6.7: (WE/SE) ∃ <t_j, ≥3, `IsComposedOf.createLink(menu=>m,course=>x)`>
  {*sat by* t_4 *iff* `_courses→size()≥3` }

- PVA_2 (`m.name = _name`) ,$SVC_{2.1}$ (ID(`name`,`Menu`)):
  Table 6.10, Row 1: (SE) (∃ <$g_{\#}$, `!Menu.allInstances()→exists(x|x.name=_name)`, $t_{\geq 1}$, $t_{\leq 4}$> AND ∄ <t_l, ≥1, `y.name = _name`>)
  {*sat*}

  Table 6.10, Row 2: (SE) (<$g_{\#}$, `x.attr = _name`, t_j, t_k> AND ∃ <t_j, 1, `x.name = ` *value*$_2$>), *value*$_2$ ≠ `_name`
  {*not sat*}

  Table 6.10, Row 3: (SE) (<$g_{\#}$, `x.attr = _name`, t_j, t_k> AND ∃ <t_j, 1, `x.destroy()`>)
  {*not sat*}

As we justified in the first step, the single path of `classifyAsSpecialMenu` operation ($p_{classifyAsSpecialMenu}$) has two PVAs. According to Table 6.6, in order to discard the PVA_1 when it may violate the $SVC_{1.1}$, the path must include, after the PVA, at least one action to initialize the attribute `discount`. The second term of the path (<t_2, 1, `self.discount = _discount`>) contains this initialization, then, we can ensure the PVA_1 will never violate the $SVC_{1.1}$.

Besides, according to Table 6.5, in order to discard the PVA_1 when it may violate the $SVC_{1.2}$, the path should satisfy one of the four alternative conditions (three if we are verifying if the path is SE). In this case, only the last alternative (row 4) is satisfied, then, the method concludes PVA_1 regarding $SVC_{1.2}$ may be discarded when verifying the weak executability but not when verifying the strong executability.

Finally, according to Table 6.9 the PVA_2 regarding $SVC_{2.1}$ may be discarded if the path contains a guard that prevents the execution of the PVA when the comparison stated in the constraint is not satisfied. This condition is satisfied by the first guard of the path (<g_1, `_discount`≥`10`, t_1, t_2>), then, PVA_2 may be discarded.

All conditions to make the $p_{classifyAsSpecialMenu}$ path WE are satisfied, thus, our method concludes this path is WE. Otherwise, not all conditions to make this path SE are satisfied, thus, this path is not SE.

Conditions to discard the PVAs of path $p_{classifyAsSpecialMenu}$:

- PVA_1 (`classify self to SpecialMenu`), $SVC_{1.1}$ (Mand(`discount`,`SpecialMenu`)):
  Table 6.6: (WE/SE) ∃ <t_j, ≥1, `self.discount = #`>
  {*sat by* t_2}

- PVA_1 (`classify self to SpecialMenu`), $SVC_{1.2}$ (Cmax(`SpecialMenu`)=3):
  Table 6.5, Row 1: (WE/SE) ∃ <$g_{\#}$, `SpecialMenu.allInstances()→size()<3`, $t_{\leq 1}$, $t_{\geq 1}$>
  {*not sat*}

  Table 6.5, Row 2: (WE/SE) ∃ <t_j, ≥1, `x.destroy()`>, `x.oclIsTypeOf(SpecialMenu)`
  {*not sat*}

  Table 6.5, Row 3: (WE/SE) ∃ <t_j, ≥1, `classify x from SpecialMenu`>, `x.oclIsTypeOf(SpecialMenu)`
  {*not sat*}

  Table 6.5, Row 4: (WE) n-m=0<Cmax(`SpecialMenu`)-Cmin(`SpecialMenu`)=3-0=3
  {*sat*}

- PVA_2 (`self.discount = _discount`), $SVC_{2.1}$ (ValueComp(`self.discount`,>=,10)):
  Table 6.9: (WE/SE) ∃ <$g_{\#}$, `self.discount>=10`, $t_{\leq 2}$, $t_{\geq 2}$>
  {*sat by* g_1}

Step 3: Classifying the operation

Step 3 of our method (see Section 6.3.3) classifies the operation (see Table 6.24) depending on the results obtained in the previous step. If at least one of the execution paths of the operation is WE, the operation is classified as WE. If all its execution paths are SE, the operation is classified as SE.

Table 6.24. Classification of our running operations.

Operation	*is weakly executable?*	*is strongly executable?*
$newCourse$	No	No
$addMenu$	Yes	Yes
$classifyAsSpecialMenu$	Yes	No

Since both path of operation `newCourse` are not WE neither SE, our method concludes the operation `newCourse` is not WE neither SE. Section 6.3.4 shows how to correct this.

Since both paths of `addMenu` are WE/SE, the method concludes this operation is WE and SE.

Since the single path of `classifyAsSpecialMenu` is WE but not SE, the method concludes this operation is WE but not SE.

The non-WE/SE operations should be repaired as the feedback points out.

Feedback

The non-satisfied conditions computed in the previous step are returned as feedback. In the following we examine the feedback returned by our method when verifying our running operations.

For `newCourse` operation, our method returns the following feedback:

Conditions that should be satisfied by the path $p2_{newCourse}$:

- In order to avoid the PVA_1 (`c = new Course()`) violates the $SVC_{1.2}$ (Mand(`category`,`Course`)), the following term should be added:
  (WE/SE) ∃ <t_j, ≥1, `c.category = #`>

- In order to avoid the PVA_2 (`CanBeSubstitutedBy.createLink (replaced=>c,replacement=> _substitutingCourses[i])`) violates the $SVC_{2.1}$ (Sym(`CanBeSubstitutedBy`)), the following term should be added:
  (WE/SE) ∃ <t_j, 1, `CanBeSubstitutedBy.createLink (replacement=>c,replaced=>_substitutingCourses[i])`>

Next, we show the repaired operation once the feedback provided by our method has been integrated. The added sentences are emphasized in bold type. Each added sentence fixes one of the problems detected in the previous section.

```
activity **newCourse**(in _description: String, in _substitutingCourses:
Course[*], **in _category: CourseCategory**) {
  Course c = new Course();
  c.description = _description;
  **c.category = _category;**
  for (i in 1.._substitutingCourses→size()) {
   CanBeSubstitutedBy.createLink(replaced=>c,
    replacement=> _substitutingCourses[i]);
   **CanBeSubstitutedBy.createLink(replacement=>c,**
    **replaced=> _substitutingCourses[i]);**
  }
}
```

The initialization of the attribute `category` ensures the constraint Mand(`category`,`Course`) will never be violated and the new created link ensures the symmetric constraint Sym(`CanBeSubstitutedBy`) will never be violated. After applying this changes, the operation `newCourse` becomes WE and SE.

For `addMenu` operation, our method returns the following feedback:

Conditions that should be satisfied by the path $p2_{addMenu}$:

If `_courses→size()`≥3, then, $p2_{addMenu}$ is WE and SE.
Otherwise, in order to avoid the PVA_1 (`m = new Menu()`) violates the $SVC_{1.3}$ (Cmin(`IsComposedOf`,`course`)=3), the following term should be added:
(WE/SE) ∃ <t_j, ≥3, `IsComposedOf.createLink(menu=>m,course=>x)`>

Note that, since the multiplicity of the `_courses` parameter is at least 3, the designer may easily conclude the above inequality is always true. Then, `addMenu` operation is WE and SE.

When verifying if the operation `classifyAsSpecialMenu` is SE, our method returns the following feedback:

Conditions that should be satisfied by the path $p_{classifyAsSpecialMenu}$:

- In order to avoid the PVA_1 (`classify self to SpecialMenu`) violates the $SVC_{1.2}$ (Cmax(`SpecialMenu`)=3), one of the following conditions should be added:

∃ <$g_{\#}$, `SpecialMenu.allInstances()→size()<3`, $t_{\leq 1}$, $t_{\geq 1}$>

∃ <t_j, ≥1, `x.destroy()`>, where `x.oclIsTypeOf(SpecialMenu)`

∃ <t_j, ≥1, `classify x from SpecialMenu`>,
where `x.oclIsTypeOf(SpecialMenu)`

Next, we show the repaired operation once the first repair alternative provided by our method has been integrated.

```
activity classifyAsSpecialMenu(in _discount:Real) {
  if ( _discount ≥ 10 and SpecialMenu.allInstances()→size()<3  ) {
   classify self to SpecialMenu;
   self.discount = _discount;
  }
}
```

6.4 Discussion

In this section we expose the assumptions our method for verifying the executability of operations relies on and discuss their limitations in order to evaluate its pros and cons.

6.4.1 Assumptions of our method

Our method to verify whether an operation is WE/SE assumes all Alf operations are syntactically correct (see Chapter 5). This is a reasonable assumption necessary to begin our analysis.

According to the widely accepted criteria about the elimination of the unreachable code [16, 42], our method also assumes the body of all conditional and loop structures are reachable (given the proper input values). This means that the condition of all conditional and loop structures may be satisfied (i.e. they can evaluate to true) and then the body of these structures may be executed. Otherwise, the actions in those paths that may be needed to compensate the effect of a PVA could not be used and thus falsify the results of the method. Roughly, this SAT-problem could be tackled with UML/OCL verification tools [32] adding the test condition as an additional constraint to the model and checking if the extended model is still satisfiable. However, this analysis, would worsen the efficiency of our method. It is up to the designer to decide whether this is needed or not.

6.4.2 Limitations of our method

Our method for verifying the executability of action-based operations presents several trade-offs that are required to enable our lightweight analysis.

As we introduced, one limitation of our method is the fact that our it performs an over-approximation analysis. This implies that it may classify as a non-WE/SE an operation which is actually WE/SE (but not the other way round, the method never marks as WE/SE an operation which is not actually executable).

This over-approximation can be resolved by the designer by participating in the second step of the method in order to disambiguate some situations that cannot be automatically computed without resorting to a search-based approach (which would then limit the benefits of the static analysis we perform). For instance, the method could ask for user intervention to determine whether two conditions are equivalent or one implies the other.

Our method is able to determine this in a number of cases but assumes the worst case scenario when it cannot be sure. Instead of extending the method with a simulation component to decide these situations (since, as said above, this would hinder the efficiency of the method), the designer could directly decide this by herself since the type of queries for which our method requires the user intervention, are basic inequalities that a designer may easily solve by examining the operation.

> **Example 41** When verifying the operation `addMenu` the user intervention is required to determine whether "`_courses→size()`" is greater or equal to 3. As we have seen, since the multiplicity of the `_courses` parameter in the `addMenu` operation is at least 3, the designer may easily conclude the above inequality is always true.

Besides, our method is not indicated to consider all the possible integrity constraints that may be defined in the structural model. Although some new constraints could be added to our patterns (adding the proper conditions in the tables of Steps 1 and 2 of our method), our method is not suitable to address complex integrity constraints. This is because the constraints our method addresses are constraints that conform to a well-known pattern that has been previously studied. The effort to add any possible pattern that an OCL integrity constraint can conform to is practically impossible.

6.4.3 Performance of our method

The time complexity of our method is exponential (wrt the size of the MBCFG) in the worst case (when all the terms of the operation are PVAs and none of them may be discarded).

The time complexity of the whole method is determined by the complexity of each step:

- Step 0: Computing the set of execution paths. The time complexity of this step of the method is exponential wrt the size of the MBCFG, since it consists of passing through all the statements of the operation in order to construct the MBCFG and, next, passing through the MBCFG in order to determine the existing execution paths.
- Steps 1 and 2: Analyzing the existence of PVAs and discarding PVAs. The time complexity of the steps 1 and 2 of the method is polynomial wrt the number of the possible execution paths, since they imply passing through several nested loops to: (step 1) determining the PVAs of each path of the operation; and (step 2) trying to discard each PVA.
- Step 3: Classifying the operation. The time complexity of the step 3 of the method is linear since it consists in examining the results obtained on the previous step in order to classify the operation.

As a result, the time complexity of the whole method is exponential wrt the size of the MBCFG. Note that, although the time complexity is exponential, it is unaffected by the size of the structural model (i.e. the number of classes, attributes, integrity constraints, etc). In this sense, our method is still faster than other non-lightweight formal methods that suffer from the state explosion problem (see some of them in Chapter 11).

Note also that the time complexity when verifying the weak executability is lower than the time complexity when verifying the strong executability, since in the former we must apply the method until we reach a WE path while in the second we must apply the method over all the paths of the operation. This, together with the second main and distinguishing benefit of our method (the kind of feedback provided to the user), justify the above assumptions and limitations.

6.5 Summary

Our method for verifying the executability of an action-based operation classifies the operations in several levels of correctness regarding the state of the system they reach after executing:

- *Non executable operations*, that is, operations that never reach a consistent system state after executing.
- *Weakly executable (WE) operations*, that is, operations that may reach a consistent system state after executing, but are not guaranteed to do so.
- *Strongly executable (SE) operations*, that is, operations that always reach a consistent system state after executing, regardless of the input values provided to the operation and the initial system state where the operation is applied over.

In order to classify an operation into one of the above types, we propose a lightweight and static method based on several steps:

1. **Step 0: Computing Execution Paths.** Prior to check the executability of an operation, our method computes all its execution paths according to the process explained in section 6.1.
2. **Step 1: Analyzing the existence of Potentially Violating Actions.** For each path, our method analyzes individually each action to see if it may violate some integrity constraints of the structural model. See more details in Section 6.3.1.
3. **Step 2: Discarding Potentially Violating Actions.** Then, our method performs a contextual analysis of each potentially violating action (PVA) to see if other actions or conditions in the path compensate or complement its effect to ensure that we sometimes/always reach a consistent state at the end of the operation execution. If all PVAs may be discarded, the method concludes the path is WE/SE. See more details in Section 6.3.2.

4. **Step 3: Classifying the operation.** Finally, our method classifies the operation depending on the results obtained in the previous step. If at least one of the execution paths of the operation is WE, the operation is classified as WE. If all its execution paths are SE, the operation is classified as SE. In case of non-WE/SE, the method returns a correcting feedback. See more details in Section 6.3.3.

My goal is simple. It is a complete understanding of the universe, why it is as it is and why it exists at all.
Stephen Hawking

7 Completeness

The aim of this chapter is to precisely define the notion of *completeness* and to provide a lightweight and static method to determine whether a behavioural model based on a set of action-based operations satisfies this property.

This chapter is divided into three sections: Section 7.1 precisely defines the *completeness* property; Section 7.2 describes a lightweight and static method we propose to determine whether a behavioural model satisfies this property; and finally, Section 7.3 summarizes and concludes the chapter.

7.1 Completeness Definition

Users may evolve the system state by executing the set of operations that compose the behavioural model.

We consider a behavioural model (i.e. a set of action-based operations) is *complete* if all possible changes (inserts/updates/deletes/ ...) on all parts of the system state can be performed through the execution of those operations. Otherwise, there will be parts of the system that users will not be able to modify since no behavioural elements address their modification.

Example 42 Consider the excerpt of the class diagram shown in Figure 7.1 and a behavioural model composed by the operations `addCourse` and `deleteMenu` shown in the following.

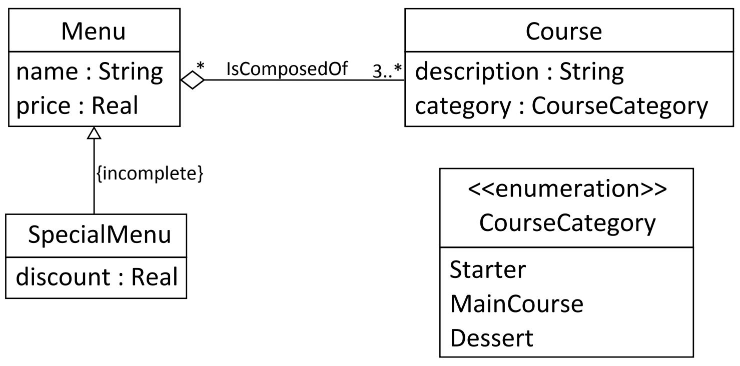


Figure 7.1. Excerpt of the structural model.

```
activity addCourse(in _description: String, in _category:
CourseCategory) {
  Course c = new Course();
  c.description = _description;
  c.category = _category;
}
```

```
activity deleteMenu(in _menu: Menu) {
  _menu.destroy();
}
```

This behavioural model is incomplete since, for instance, actions to destroy courses or to create menus are not specified, forbidding users to perform such kind of changes on the data.

The possible changes on the system state are determined by the modifiable elements in the structural model (i.e. the elements whose value or population can be changed by the user at run-time). For instance, the population of a class may change during the life-span of the system. As an example, in our restaurant chain structural model, menus can be added and deleted along the time.

Given a structural model (SM), *ModifiableElements(SM)* returns the set of all modifiable elements in SM.

Each modifiable element requires specific actions to be modified. For instance, a class `cl` requires an action to create instances of `cl` and an action to destroy instances of `cl`.

Given an element e of the structural model, *RequiredActions(e)* returns the set of necessary actions to modify the element e of the structural model.

Then, more formally:

Let ExM = ⟨SM,BM⟩ be an executable model, BM is **complete** iff ∀ e ∈ ModifiableElements(SM) ∧ ∀ a ∈ RequiredActions(e) ∃ op ∈ BM | IsWeakExecutable(op) and a ∈ Actions(op).

Where *IsWeakExecutable(op)* returns true if *op* is weakly executable, and *Actions(op)* returns a list of all the actions included in the operation *op*.

We feel this property is important to guarantee that no behavioural elements are missing in the model. Clearly, it may happen that a class diagram contains some elements that designers do not want the users to modify but then those elements should be defined, for instance, as derived information.

7.2 Verifying the completeness

The method we have developed for verifying the completeness property (see Figure 7.2) takes as input an executable model composed by a structural model (a UML class diagram) and a behavioural model (a set of Alf operations). We consider all the input operations are syntactically correct and (weak) executable. Then, our method returns either a positive answer, meaning that the behavioural model is complete, or a corrective feedback, consisting in a set of actions that should be included in some operation of the behavioural model in order to make it complete.

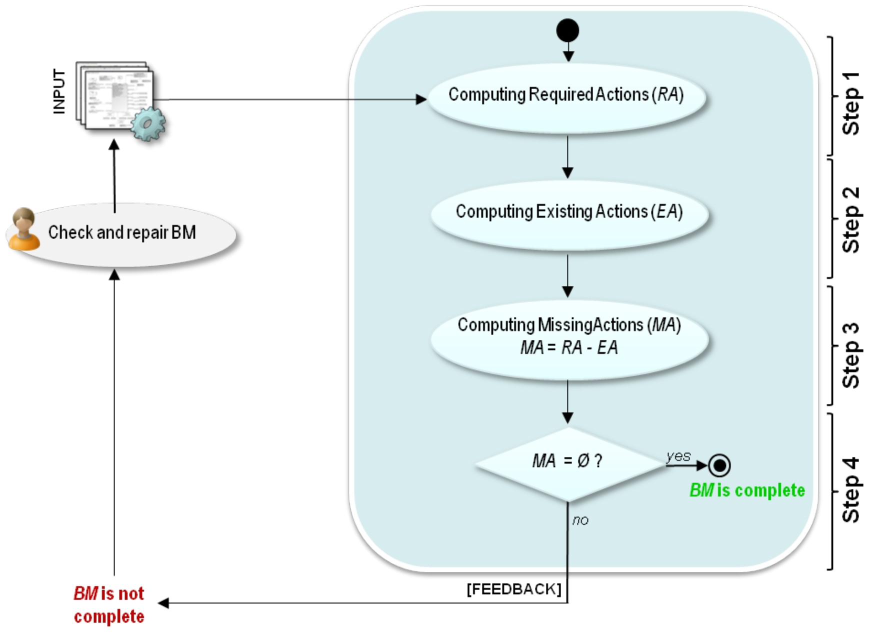


Figure 7.2. Completeness method overview.

Our method is based on four steps. Step 1 (see Section 7.2.1) computes the required actions that the behavioural model should include. Step 2 (see Section 7.2.2) computes the existing actions included in the input behavioural model. Step 3 (see Section 7.2.3) determines the missing actions, i.e. those actions that should be included in the behavioural model but they do not belong to. Finally, Step 4 (see Section 7.2.4) classifies the behavioural model depending on the result obtained in the previous step.

At this point we would like to remark that the method proposed in this thesis verifies the completeness regarding the modifications over the structural model (i.e. it does not consider reading/querying actions). As we stated in Chapter 2, in this thesis we focus on modification actions. However, our method could be extended to consider the completeness regarding other types of actions.

7.2.1 Step 1: Computing Required Actions (RA)

First step of our verification method computes the required actions (RA) that the behavioural model should include, i.e. those actions that allow modifying all the modifiable elements of the structural model.

Given a class diagram, their modifiable elements are determined according to the following rules:

- A **class** is modifiable as long as it is not an abstract class and it is not the supertype of a complete generalization set (instances of such supertypes must be created/deleted through their subclasses).
- An **attribute** is modifiable when it is not derived[17].
- An **association** is modifiable if none of its member ends are derived. Aggregations and compositions are treated as associations.
- A **generalization** is always modifiable since objects may be classified from/to their subclasses.

> **Example 43** Applying the above rules to the class diagram of Figure 7.1, our method determines its modifiable elements (see Table 7.1).

Once the modifiable elements have been identified, our method determines the required actions to modify each modifiable element. Table 7.2 shows the required actions for each element type. Sharp sign (#) represents irrelevant variables and consecutive letters (x, y,...) represent free variables that may be bound to any value in the action.

[17]Read-only attributes are considered modifiable because users must be able to initialize their value (and similar for read-only associations).

Table 7.1. Example: Modifiable elements of the structural model shown in Figure 7.1.

Type	*Element*

Class	`Menu`
	`SpecialMenu`
	`Course`

Attribute	`name` (from `Menu`) `price` (from `Menu`) `discount` (from `SpecialMenu`) `description` (from `Course`) `category` (from `Course`)
Association	`IsComposedOf(Menu,Course)`
Generalization	`Menu` generalizes `SpecialMenu`

Table 7.2. Required actions to modify each element of the structural model.

Element	*Required actions to modify the element*	*Description*
Class `cl`	`x = new class()`, where `class` is equal to `cl` or one of its subclasses	A modifiable class `cl` requires an action to create instances of `cl` or of one of its subclasses
	`x.destroy()`, where `x` is an instance of `cl` or of one of its subclasses	A modifiable class `cl` requires an action to destroy instances from `cl` or from one of its subclasses
Attribute `attr` from `cl`	`x.attr = #`, where `x` is an instance of `cl` or of one of its subclasses	A modifiable attribute `attr` requires an action to modify its value
Association `as` between `cl` (with role r_a) and `cl'` (with role r_b)	`as.createLink(`r_a`=>x,` r_b`=>y)`, `x` is an instance of `cl` and `y` is an instance of `cl'`	A modifiable association `as` requires an action to create links of `as`
	`as.destroyLink(`r_a`=>x,` r_b`=>y)` OR `as.clearAssoc(x)`, where `x` is an instance of `cl` and `y` is an instance of `cl'`	A modifiable association `as` requires an action to destroy some or all links of `as`
Generalization `g`, where `cl` generalizes $\{cl_1,..., cl_n\}$	`classify x from oldCl`, where `oldCl` $\subset \{cl_1,..., cl_n\}$	A modifiable generalization `g` requires an action to reclassify objects from a subclass of `cl`
	`classify x to newCl`, where `newCl` $\subset \{cl_1,..., cl_n\}$	A modifiable generalization `g` requires an action to reclassify objects to a subclass of `cl`

Example 44 Applying the rules of Table 7.2 to the modifiable elements identified in Table 7.1 our method determines the required actions. They are listed in Table 7.3:

Table 7.3. Example: Required actions to modify each modifiable element (see Table 7.1).

Element	*Required actions to modify the element*
Class `Menu`	`x = new Menu()` OR `x = new SpecialMenu()`
	`x.destroy()`, where `x` is an instance of `Menu` or of `SpecialMenu`
Class `SpecialMenu`	`x = new SpecialMenu()`
	`x.destroy()`, where `x` is an instance of `SpecialMenu`
Class `Course`	`x = new Course()`
	`x.destroy()`, where `x` is an instance of `Course`
Attribute `name` (from `Menu`)	`x.name = #`, where `x` is an instance of `Menu` or of `SpecialMenu`
Attribute `price` (from `Menu`)	`x.price = #`, where `x` is an instance of `Menu` or of `SpecialMenu`
Attribute `discount` (from `SpecialMenu`)	`x.discount = #`, where `x` is an instance of `SpecialMenu`
Attribute `description` (from `Course`)	`x.description = #`, where `x` is an instance of `Course`
Attribute `category` (from `Course`)	`x.category = #`, where `x` is an instance of `Course`
Association `IsComposedOf (Menu,Course)`	`IsComposedOf.createLink(menu=>x,course=>y)`, where `x` is an instance of `Menu` and `y` is an instance of `Course`
	`IsComposedOf.destroyLink(menu=>x,course=>y)` OR `IsComposedOf.clearAssoc(x)`, where `x` is an instance of `Menu` and `y` is an instance of `Course`
Generalization `Menu` generalizes `SpecialMenu`	`classify x from SpecialMenu`
	`classify x to SpecialMenu`

The set of required actions do not contain repeated actions. It means that, if two modifiable elements require the same action to be modified, this action is added only once in the required actions set. Note also that the required actions set may contain subsets of disjoint actions, meaning that at least one of the disjoint actions must exist in the behavioural model.

Example 45 The set of required actions for our example (where repeated actions have been removed) is:

Required actions (RA) for the structural model of Figure 7.1:
  `x = new Menu()` OR `x = new SpecialMenu()`
  `x.destroy()`, where `x` is an instance of `Menu` or of `SpecialMenu`
  `x = new Course()`
  `x.destroy()`, where `x` is an instance of `Course`
  `x.name = #`, where `x` is an instance of `Menu` or of `SpecialMenu`
  `x.price = #`, where `x` is an instance of `Menu` or of `SpecialMenu`
  `x.discount = #`, where `x` is an instance of `SpecialMenu`
  `x.description = #`, where `x` is an instance of `Course`
  `x.category = #`, where `x` is an instance of `Course`
  `IsComposedOf.createLink(menu=>x,course=>y)`, where `x` is an instance of `Menu` and `y` is an instance of `Course`
  `IsComposedOf.destroyLink(menu=>x,course=>y)` OR `IsComposedOf.clearAssoc(x)`, where `x` is an instance of `Menu` and `y` is an instance of `Course`
  `classify x from SpecialMenu`, where `x` is an instance of `SpecialMenu`
  `classify x to SpecialMenu`, where `x` is an instance of `Menu`

7.2.2 Step 2: Computing Existing Actions (EA)

Second step of our verification method computes the existing actions (EA), that is, those actions that are included in the operations of the input behavioural model.

This step simply retrieves a set of non repeated actions that appear on the operations that compose the behavioural model we are verifying. Equivalent actions (i.e. actions of the same type that address the same model elements although the instance-level parameters are not required to be equal) are added only once in the set of existing actions. For instance, the actions `Menu m1 = new Menu()` and `Menu m2 = new Menu()` are equivalent since both create a new menu to the system and then only one of them (regardless which one) should be included in the existing actions set.

Example 46 Given the behavioural model composed by the operations `addCourse` and `deleteMenu`, the set of existing actions is:

Set of existing actions (EA) from `addCourse` and `deleteMenu` operations:
```
Course c = new Course()
c.description = _description
c.category = _category
_menu.destroy()
```

7.2.3 Step 3: Computing Missing Actions (MA)

Third step of our verification method computes the missing actions (MA), that is, those actions that should be included in some operation of the behavioural model but they do not belong to any operation. In order to obtain the missing actions, our method computes the difference between the set of required actions (RA) (those computed in the Step 1) and the set of existing actions (EA) (those computed in the Step 2) and returns a (maybe empty) subset of missing actions (MA).

During the comparison between the required actions and the existing actions, we do not take into account the concrete values of the elements which participate in the action. It means that irrelevant values (#) and free variables (x, y, ...) of the required actions may map to any concrete value/variable of the existing actions. For instance, the existing action _menu.destroy() covers the required action x.destroy() since the object _menu maps to the free variable x.

Example 47 Table 7.4 shows the difference between the required actions and the existing actions.

Table 7.4. Example: Difference between the required actions and the existing actions.

Required Actions	*Existing Actions*	*Missing?*
`x = new Menu()` OR `x = new SpecialMenu()`		Yes
`x.destroy()`, where `x` is an instance of `Menu` or of `SpecialMenu`	`_menu.destroy()`	No
`x = new SpecialMenu()`		Yes
`x.destroy()`, where `x` is an instance of `SpecialMenu`		Yes
`x = new Course()`	`c = new Course()`	No
`x.destroy()`, where `x` is an instance of `Course`		Yes
`x.name = #`, where `x` is an instance of `Menu` or of `SpecialMenu`		Yes
`x.price = #`, where `x` is an instance of `Menu` or of `SpecialMenu`		Yes
`x.discount = #`, where `x` is an instance of `SpecialMenu`		Yes
`x.description = #`, where `x` is an instance of `Course`	`c.description = _description`	No
`x.category = #`, where `x` is an instance of `Course`	`c.category = _category`	No
`IsComposedOf.createLink(menu=>x, course=>y)`		Yes
`IsComposedOf.destroyLink(menu=>x, course=>y)` OR `IsComposedOf.clearAssoc(x)`		Yes
`classify x from SpecialMenu`, where `x` is an instance of `SpecialMenu`		Yes
`classify x to SpecialMenu`, where `x` is an instance of `Menu`		Yes

Then, the set of missing actions that should be included in some action of the behavioural model is:

Set of missing actions (MA):
  `x = new Menu()` OR `x = new SpecialMenu()`
  `x = new SpecialMenu()`
  `x.destroy()`, where `x` is an instance of `SpecialMenu`
  `x.destroy()`, where `x` is an instance of `Course`
  `x.name = #`, where `x` is an instance of `Menu` or of `SpecialMenu`
  `x.price = #`, where `x` is an instance of `Menu` or of `SpecialMenu`
  `x.discount = #`, where `x` is an instance of `SpecialMenu`
  `IsComposedOf.createLink(menu=>x,course=>y)`, where `x` is an instance of `Menu` and `y` is an instance of `Course`
  `IsComposedOf.destroyLink(menu=>x,course=>y)` OR `IsComposedOf.clearAssoc(x)`, where `x` is an instance of `Menu` and `y` is an instance of `Course`
  `classify x from SpecialMenu`, where `x` is an instance of `SpecialMenu`
  `classify x to SpecialMenu`, where `x` is an instance of `Menu`

7.2.4 Step 4: Classifying the Behavioural Model

Last step of our method classifies the behavioural model depending on the result obtained in the previous step.

If the set of missing actions (MA) is empty, it means that the sets RA (required actions) and EA (existing actions) contain exactly the same elements (i.e. all the required actions exist in the operations of the input behavioural model). Then, the behavioural model is complete. Otherwise, the behavioural model is incomplete and our method returns as feedback the subset of actions that are required but they are not included in any operation of the behavioural model.

Example 48 Since the set of missing actions of our example is not empty, our method concludes the input behavioural model is not complete. As a result, the method provides the set of actions shown in the previous example. These actions should be added in some operation of the behavioural model in order to being complete.

7.3 Summary

In this chapter we have reviewed the completeness property of a behavioural model based on a set of action-based operations.

We stated that a behavioural model is **complete** when all possible changes (inserts/updates/deletes/ ...) on all parts of the system state can be performed through the execution of those operations. Otherwise, there will be parts of the system that users will not be able to modify since no behavioural elements address their modification.

Finally, we have described a lightweight and static method for checking the above property. In case of the checked property is not satisfied, our method returns a correcting feedback to help the designers repair the input model.

The greatest happiness is to transform one's feelings into action.

Madame de Stael

8 Application to Model-to-Model Transformations

Models are neither isolated nor static entities. As part of the MDE process, models are *merged* and *aligned* (e.g. to create a global representation of the system from different views to reason about multi-viewpoint consistency), *refactored* (to improve their internal structure without changing their observable semantics), *refined* (to detail high-level models), and *translated* (to other languages/representations, e.g. as part of code-generation or verification/simulation processes) [24].

All these operations on models are implemented as *model transformations*, which automate the translation of models between a *source* and a *target* language using a model transformation language. Model transformations are in many ways similar to traditional software artifacts (for instance, they require maintenance, they have to be changed according to changing requirements, they should be preferably reused, and so on). Therefore, they need to be verified as well.

The aim of this chapter is to adapt part of the verification methods we have presented in the previous chapters to the context of model-to-model (M2M) transformations. In particular, in this chapter we address the *weak executability* (see Chapter 6) and the *completeness* (see Chapter 7) properties. We focus on these two properties according to the Proof of Concept (PoC) principle, i.e. we develop a partial solution (focused on these two properties) to demonstrate the feasibility of our methods in the context of M2M transformations. Both methods must be adapted from the version described in the above chapters since now the input is a M2M transformation instead of an Alf-based operation. Besides, as we will see in this chapter, given the structural differences between action-based operations and M2M transformations, some steps of the original methods may be simplified while others must be extended.

This chapter is divided into four sections: Section 8.1 introduces M2M transformations; Sections 8.2 and 8.3 describe how different types of M2M transformations may be verified using our lightweight static methods; and finally, Section 8.4 summarizes and concludes the chapter.

8.1 Introduction to model-to-model transformations

In a general sense, a model-to-model (M2M) transformation is a program which takes one or more models as input (i.e. source models) to produce one or more models as output (i.e. target models) based on some well-defined rules [24].

More formally, as shown in Figure 8.1, a model transformation has to define the way for generating a target model `Mb`, conforming to a metamodel `MMb`, from a source model `Ma` conforming to a metamodel `MMa`. The model transformation itself (`Mt`) is also defined as a model. This transformation model has to conform to a transformation metamodel (`MMt`) that defines the model transformation semantics. As the other metamodels, the transformation metamodel has, in turn, to conform to the considered metametamodel.

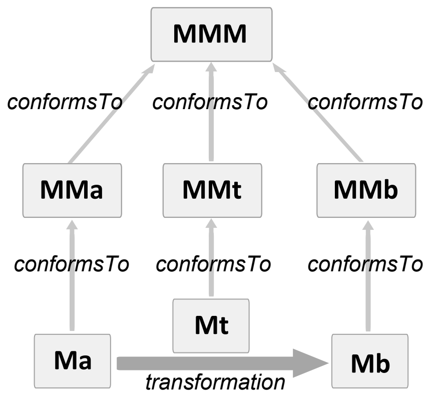


Figure 8.1. Model Transformation schema.

Note that the nomenclature used by the model transformations community differs from that used by the conceptual modelling community (see Figure 8.2). In the model transformations community, the term *model* refers to the source and target models which participate in the transformation. These models, according to the OMG vision, may be described at different levels of abstraction: M0 (i.e. a transformation from/to object models), M1 (i.e. a transformation from/to UML models), M2 (i.e. a transformation from/to a UML metamodel), etc. In the modelling community context, however, the term *model* always refers to the user model (M1 level). In this chapter we use the nomenclature adopted by the model transformations community, i.e. we consider that a *model* may be described at different (but only one) abstraction levels according to the OMG vision. In particular, the *models* considered as a source and target models of the transformations presented in this chapter are at the M0 level of the OMG.

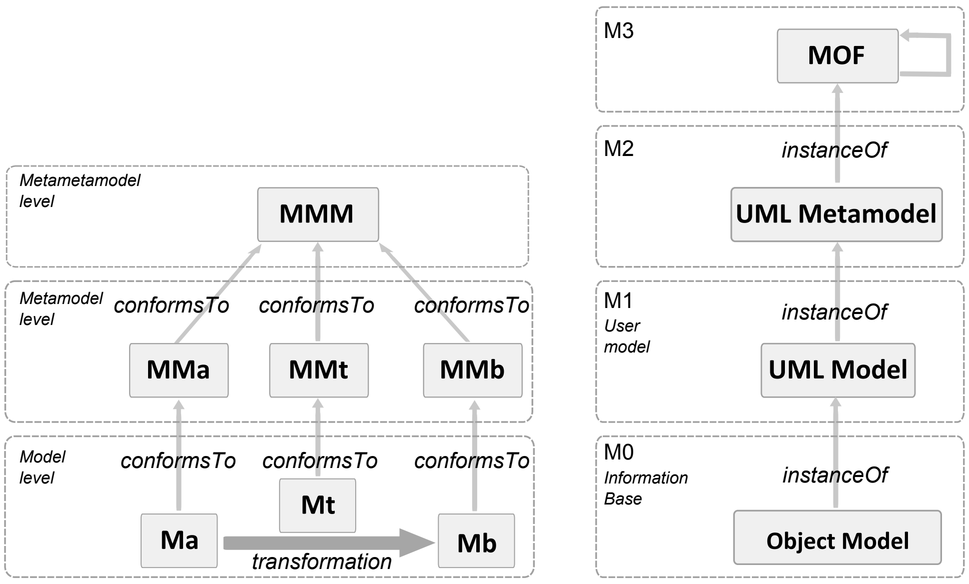


Figure 8.2. Model Transformation schema (left) vs UML/MOF instantiation hierarchy according to the OMG vision (right).

M2M transformations may be classified in several categories according to different perspectives (see Figure 8.3):

- According to the number of input and output models, M2M transformations may be classified in: (1) *one-to-one* transformations, having one input/output model; or (2) *one-to-many*/*many-to-many* transformations, when several models are required.
- According to the languages in which the input/output models are defined, M2M transformations may be classified in: (1) *exogenous* transformations, when they are defined between two different languages (for instance, in the MDA scenario, a transformation from a platform-independent model, e.g. a UML model, to a platform-specific model, e.g. a Java model); or (2) *endogenous* transformations, when they are defined within one language (for instance, a model refactoring).
- According to the implementation paradigm they rely on, M2M transformations may be classified in: (1) *in-place* transformations, for *rewriting* a model by creating, deleting and updating elements in the input model (used, for instance, to refactor models); or (2) *out-place* transformations, for *generating* the output model from scratch (used, for instance, in a code generation scenario). Note that *in-place* transformations are specially suited for *endogenous* transformations, while *out-place* transformations are specially suited for *exogenous* transformations.

In order to understand the remaining of this chapter, in the following we present how exogenous transformations may be specified as out-place transformations using ATL (see Section 8.1.1) and how endogenous transformations may be specified as in-place transformations using Graph Transformation rules (see Section 8.1.2).

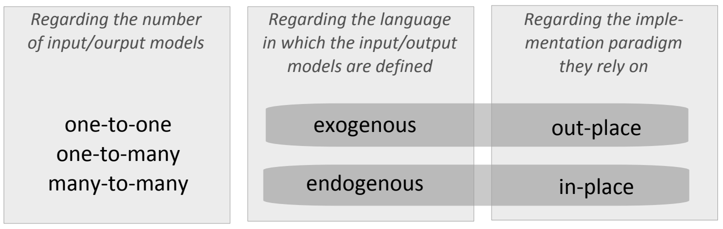

Regarding the number of input/ourput models
one-to-one
one-to-many
many-to-many
Regarding the language in which the input/output models are defined
exogenous
endogenous
Regarding the imple-mentation paradigm they rely on
out-place
in-place

Figure 8.3. Classification of model-to-model (M2M) transformations regarding several perspectives.

8.1.1 Exogenous out-place transformations with ATL

This section briefly introduces some preliminary concepts to define exogenous out-place transformations and presents a running example that will be used in Section 8.2.

In the remaining of this chapter we presume exogenous out-place transformations are specified using the ATL [94] language. We choose that language because it is one of the most widely used transformation languages, both in academia and industry, and there is mature tool support available. However, the ideas presented in this chapter could also be adapted to models specified by means of other languages such as Query-View-Transformation language (QVT) [122], Triple Graph Grammars (TGG) [160], Epsilon Transformation Language (ETL) [98] or RubyTL [49].

ATL is designed as a hybrid model transformation language containing a mixture of declarative and imperative constructs. ATL transformations are $uni-directional$ meaning that if a transformation from language A to language B is required, and vice versa, two transformations have to be developed. ATL transformations are operating on *read-only* source models and producing *write-only* target models. This means that, during the execution of a transformation, source models may be queried but no changes to them are allowed. In contrast target model elements are created, but should not be queried directly during the transformation.

Metamodels

ATL rules (see Figure 8.4) describe the transformation from a source model (which conforms to a source metamodel) to a target model (which conforms to a target metamodel) by relating its metamodels. The source and target metamodels are described in the Ecore language (which allows to formally define its structure).

Example 49 In the rest of this section we make use of a running example that describes a simple transformation between people and students.

The transformation is defined between the `Person` metamodel (see Figure 8.5 left) and the `Student` metamodel (see Figure 8.5 right). The `Person` metamodel (`PersonMM`) consists of people having a name, surname, age and college name. Besides, people may know other people (in a symmetric way). On the other hand, the `Student` metamodel (`StudentMM`) consists of students having a full name.

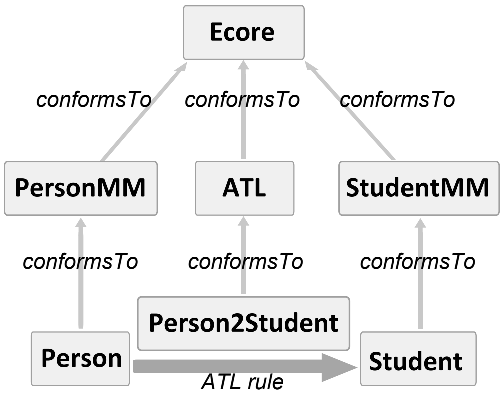


Figure 8.4. Schema of an ATL Model Transformation.

Students study at a college and they must be enrolled in at least one subject.

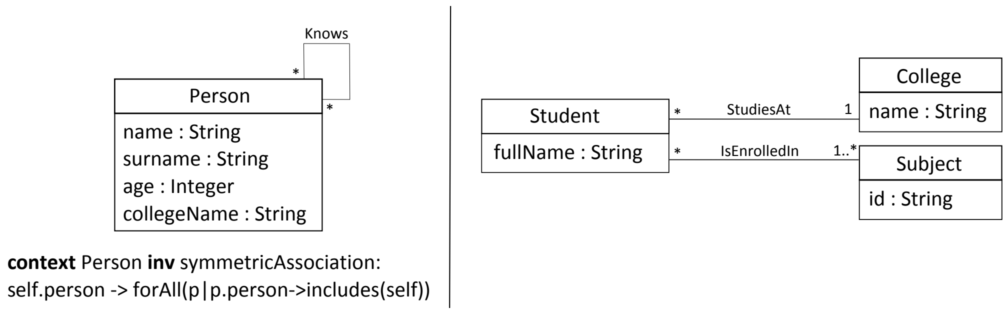


Figure 8.5. `PersonMM` (source metamodel) and `StudentMM` (target metamodel).

ATL Rules specification

A transformation defined in ATL is represented as a module (preceded by the keyword `module`). A module is defined in the header section of an ATL transformation by starting the name of the transformation module and declaring the source (preceded by the keyword `from`) and the target (preceded by the keyword `create`) model(s) which are typed by their metamodels. There can be more than one input model and output model for an ATL transformation.

The body of and ATL transformation is composed by a set of *transformation rules* and *helpers* which are stated in arbitrary order after the header section. Each rule describes how (part of) the target model should be generated from (part of) the source model, i.e. how the source model elements are matched and navigated to create and initialize the target model elements.

An ATL rule is introduced by the keyword `rule` followed by the rule's name. Rules are mainly composed of a *source* pattern and a *target* pattern. The source pattern (preceded by the keyword `from`) filters the subset of source model elements that are concerned by the rule by defining one or more source pattern element(s). In particular, an obligatory model element type has to be stated for each source pattern element as well as an optional filter (a condition expressed as an OCL expression, restricting the rule to elements of the source model that satisfy certain constraints) may be defined for the complete source pattern. On the other hand, the target pattern (preceded by the keyword `to`) describes how the target model elements are created from the source ones. Each target pattern element can have several bindings (also defined as OCL expressions) that are used to initialize the features of the target model elements.

There are two kinds of declarative rules in ATL: *matched* rules and *lazy* rules. The former are automatically matched on the source model - as their name suggests - by the ATL execution engine according to the rule's source pattern, whereas the latter has to be explicitly called from another rule giving the transformation developer more control over the transformation execution. In this chapter we focus on *matched* rules, although the methods proposed here could be extended to address other kinds of ATL rules.

On the other hand, a *helper* can be seen as an auxiliary function that enables the possibility of factorizing some ATL code used in different points of the transformation. Helpers may represent attributes which are accessible throughout the complete transformation or operations which calculate a value for a given context object and input parameters. Opposed to rules, helpers cannot produce target model elements; they can only return values which are further processed within rules.

Example 50 Let's assume that the following requirement has to be fulfilled by the transformation from `Person` to `Student`: For each `Person` instance in the source model, a `Student` instance has to be created in the target model. The full name of the student has to be set linking together the name and surname of the person. Besides, a college instance has also be created (with the college name from the person) and related to the new student.

The ATL rule that implements the above requirement (using a helper) is:

```
module Person2Student:
  create OUT: StudentMM from IN: PersonMM

 helper context PersonMM!Person def:
  getFullName(): String = self.name + ' ' + self.surname;

 rule Person2Student {
  from p: PersonMM!Person
  to s: StudentMM!Student (
   fullName <- p.getFullName(),
   college <- c
  ),
  c: StudentMM!College (
   name <- p.collegeName
  )
 }
```

8.1.2 Endogenous in-place transformations with Graph Transformation Rules

This section briefly introduces some preliminary concepts to define endogenous in-place transformations using Graph Transformations and presents a running example that will be used in Section 8.3.

Graph Transformation [56] is a declarative, rule-based technique for expressing in-place transformations based on the fact that models and metamodels can be expressed as graphs (with typed, attributed nodes and edges), and thus, manipulated using graph transformation techniques.

This formalization is specially useful to define in-place transformations in order to support model animation, simulation, optimization, execution, evolution and refactorings/redesigns. Furthermore, they are so general that also out-place transformations may be formulated with them.

Graph Transformations are now gaining increasing popularity due to their visual form (making rules intuitive) and formal nature (making rules amenable to analysis). For example, graph transformations can be used to describe the operational semantics of modelling languages for implementing a model execution engine, taking the advantage that it is possible to use the abstract syntax and sometimes even the concrete syntax of the modeling language in the rules, which then become very intuitive to the designer.

Metamodel

Graph Transformation rules describe transformations within the same metamodel (see Figure 8.6). In the remaining of this chapter we use a Domain Specific Visual Language (DSVL) to contextualize endogenous transformations.

Example 51 Figure 8.7 shows the metamodel, represented in UML, that defines the syntax of the DSVL. This metamodel includes elements of type `conveyor` that can be connected to other conveyors, to `generators` of `parts`, to `containers` or to `machines` (being mandatory that each conveyor is connected to either a machine or a container, as the *xor* constraint indicates). Conveyors can contain parts up to its maximum capacity (attribute `capacity`), which is controlled by the OCL integrity constraint defined on class `Conveyor`. Parts can either be transported in a conveyor or processed in a machine, but not both simultaneously. Containers are terminal elements that count the number of parts that have finished. Finally, production systems must contain exactly one generator, as expresses the first OCL integrity constraint.

Example 52 Figure 8.8 shows a model conformant to the previous metamodel, using the abstract syntax on the top and the visual concrete syntax at the bottom. The model contains one generator (depicted as a triangle), three conveyors (lattice boxes), one machine (coloured square), two containers (circles) and three

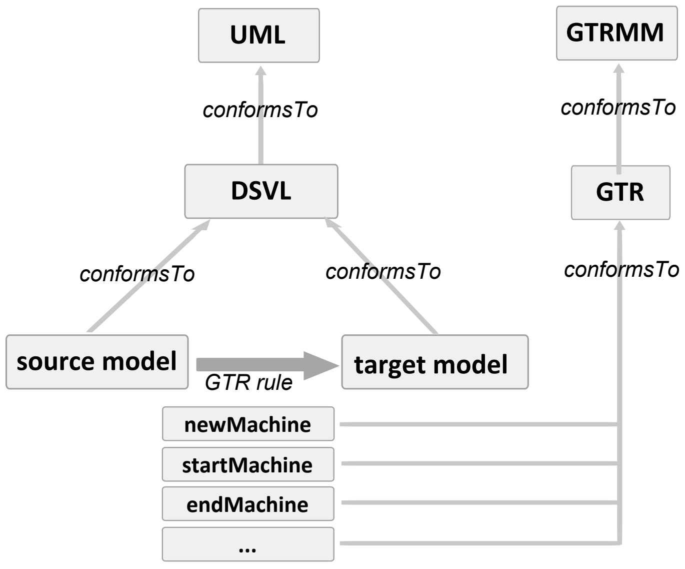
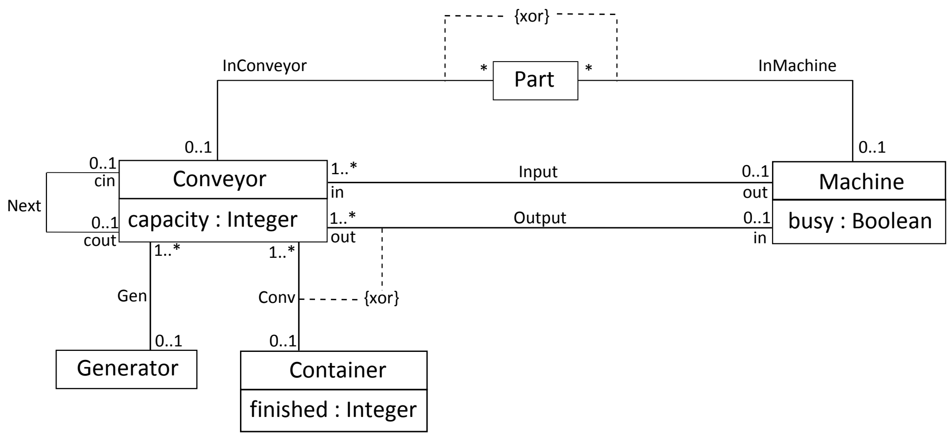


Figure 8.6. Schema of a GTR Model Transformation.



context Generator **inv** numberOfGenerators: Generator.allInstances()->size()=1
context Conveyor **inv** minimumCapacity: self.capacity >0 and self.part->size()<=self.capacity
context Machine **inv** busyMachine: if (self.busy=false) then self.part->size()=0 else self.part->size()>0 endif

Figure 8.7. Domain Specific Visual Language (DSVL) metamodel.

parts (white squares). The first conveyor contains two parts and therefore is full, while the machine is busy processing a part. All associations in the metamodel are bidirectional, but we have used arrows in the concrete syntax, which do not affect navigability. For all associations - except for those that include a part as association end - the arrow in the concrete syntax helps identifying the input and output components in the production chain.

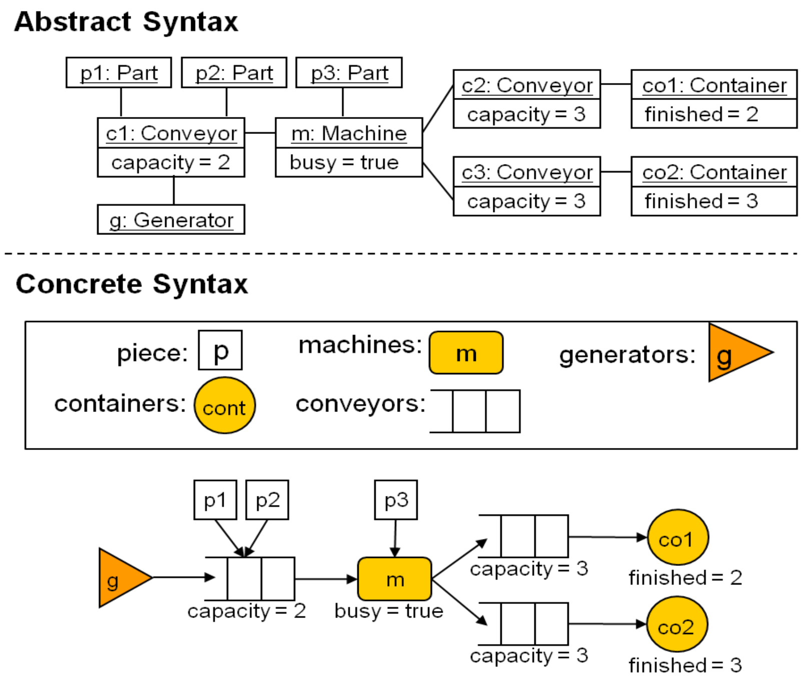


Figure 8.8. Example production system model.

Graph Transformation rules specification

A *graph grammar* is made of a set of rules and an initial graph (called *host graph*) to which the rules are applied. Each rule is made of a left hand side (LHS) and a right hand side (RHS) graph. The LHS expresses the pre-conditions for the rule to be applied, whereas the RHS contains the rule's post-conditions. The updates that are going to be carried out are implicitly defined in both sides. More precisely, the execution of a transformation rule produces the following effects: (1) all elements that only reside in the LHS are deleted; (2) all elements that only exist in the RHS are added; and (3) all elements that reside in both sides are preserved. To mark that an element in the RHS is equivalent to an element in the LHS, the elements must have the same identifier assigned.

Example 53 The rule `newMachine` (see Figure 8.9) incorporates a new machine to the plant, receiving parts initially processed by an existing overloaded machine. The LHS of this rule states that there has to exist a machine (`m1`) connected to two conveyors (`c1` and `c2`). The RHS shows that a new machine (`m2`) has been added and one of the conveyors has been linked to `m2` instead of `m1`.

In order to apply a rule to the *host graph*, a *morphism* (often also referred as *occurrence* or *match*) of the LHS has to be found in the *host graph*. If several matches are found, one is selected randomly. 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 of them is applicable.

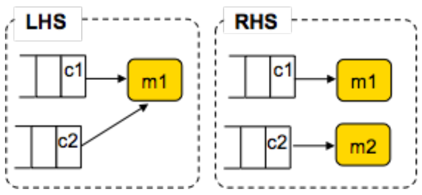


Figure 8.9. Rule `newMachine`.

Even though Graph Transformation rules are declarative, in the rest of this chapter we use a compact and operational notation used e.g. in tools like Fujava (www.fujava.de). The elements created by the rules are enclosed in a polygon labelled *new*, while the elements deleted by the rules are enclosed in a polygon labelled *del*.

Example 54 Figure 8.10 shows some rules describing the DSVL operational semantics using the compact notation. Rule `startMachine` starts the processing of a part by a free machine, and then its state changes to busy. The value of the attribute `busy` before and after applying the rule is controlled by the attribute condition and computation sections, expressed in OCL. Rule `endMachine` finishes this processing and deposits the processed part in an output conveyor, if it is not full (checked by the attribute `condition`). Rule `advance` moves parts through conveyors. Rule `generate` produces a new part in the chain. Rule `terminate` stores a part in a container and increases its attribute finished in one unit. Rule `newMachine`, as we introduced, incorporates a new machine to the plant. Finally, rule `optimize` maximizes the use of conveyors by allowing two machines to share them as output.

A major concern of a rule-based approach such as Graph Transformations is to control the application of rules. The LHS of a rule specifies what must exist in a graph to execute the rule. However, often it is required to describe what must not exist in a host graph to apply a rule. Therefore, *Negative Application Conditions* (NACs) have been introduced for Graph Transformations. A NAC is a graph that describes a forbidden sub-graph structure, i.e. the absence of specific nodes and edges must be granted. A graph transformation rule containing a NAC is executed when a match for the LHS is found and the NAC is not fulfilled. Not only one NAC can be specified for a rule, but several NACs are possible.

Example 55 Rule `disconnectGenerator` disconnects a generator. Note that this rule uses a NAC that forbids the generator that is going to be turned off is connected with more than one conveyor.

There are two main formalizations of algebraic graph transformation [153]: DPO (Double-PushOut approach) and SPO (Single-PushOut approach). 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. This condition is called *dangling edge condition*. Instead, in SPO dangling edges are

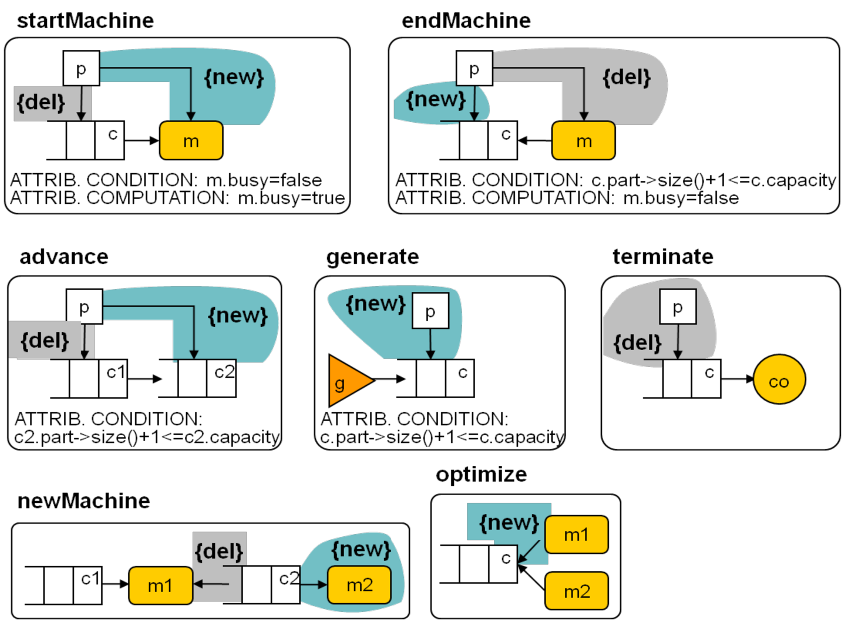

startMachine
p
{new}
{del}
c
m
ATTRIB. CONDITION: m.busy=false
ATTRIB. COMPUTATION: m.busy=true
endMachine
p
{del}
{new}
c
m
ATTRIB. CONDITION: c.part->size()+1<=c.capacity
ATTRIB. COMPUTATION: m.busy=false
advance
p
{new}
{del}
c1
c2
ATTRIB. CONDITION:
c2.part->size()+1<=c2.capacity
generate
{new}
p
g
c
ATTRIB. CONDITION:
c.part->size()+1<=c.capacity
terminate
p
{del}
c
co
optimize
{new}
m1
c
m2
newMachine
{del}
{new}
c1
m1
c2
m2

Figure 8.10. Some rules of the DSVL simulator.

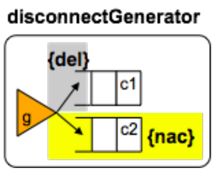


Figure 8.11. Rule `disconnectGenerator`.

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*. The method presented in Section 8.3 of this chapter supports both formalizations, although in the examples we assume SPO rules.

8.2 Verifying ATL rules

In this section we adapt the methods we presented in Chapters 6 (see Section 8.2.1) and 7 (see Section 8.2.2) to verify the weak executability and the completeness of ATL rules.

8.2.1 Weak Executability of an ATL rule

We consider an ATL rule r is *weakly executable* (WE) if it has a chance of being successfully executed. That is, if there is at least a given set of elements that matches with the source model for which the execution of the rule r generates a target model consistent with the target metamodel and its integrity constraints. Otherwise r is useless, as every time it is executed, an error arises because the target model violates some integrity constraints.

A more formal definition of this property may be found in Chapter 6.

Example 56 Rule `Person2Student` is not weakly executable since every time we create a new student and we do not associate it to any subject, we reach an erroneous target model where the minimum 1 cardinality of the association `IsEnrolledIn` (see Figure 8.5 right) is violated.

```
module Person2Student:
  create OUT: StudentMM from IN: PersonMM

 helper context PersonMM!Person def:
  getFullName(): String = self.name + ' ' + self.surname;

 rule Person2Student {
  from p: PersonMM!Person
  to s: StudentMM!Student (
   fullName <- p.getFullName(),
   college <- c
  ),
  c: StudentMM!College (
   name <- p.collegeName
  )
 }
```

As we will see later, in this case, our method reports that, in order to create a new student, we need to relate it to at least one subject within the same rule execution.

Verifying the weak executability of an ATL rule

To determine whether an ATL rule is weakly executable we have adapted the method we presented in Chapter 6 to address ATL transformations instead of action-based operations.

In this case, our lightweight static method (see Figure 8.12) takes as input the source and the target metamodels on which the ATL transformation rule is defined and the ATL rule itself. As before, our method returns either a positive answer, meaning that the ATL rule is WE, or a corrective feedback, consisting in a set of updates that should be added to the rule in order to make it WE.

When verifying ATL matched rules, the method presented in Chapter 6 may be simplified since ATL matched rules do not allow forking the execution of the rules, hence, a single execution

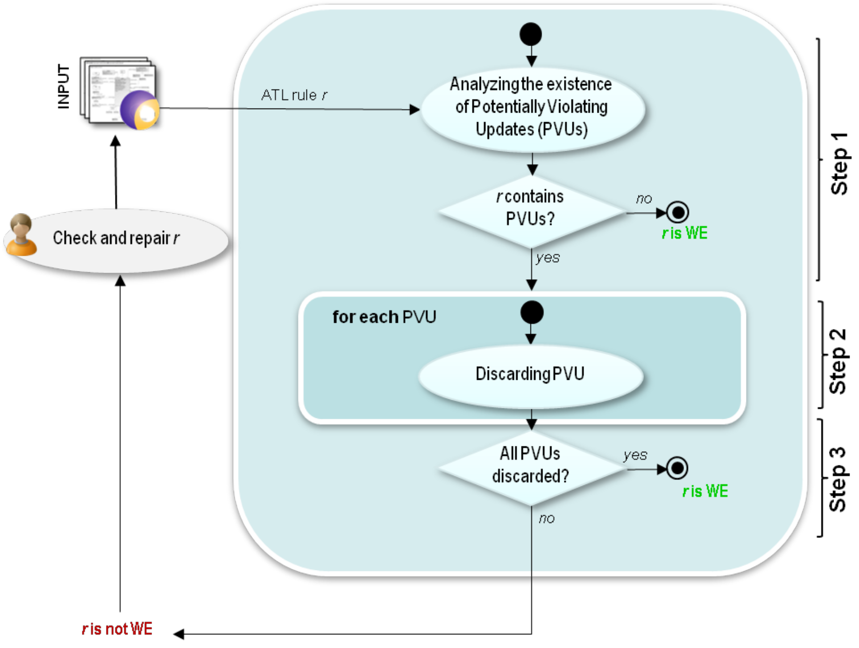


Figure 8.12. Weak executability of ATL rules method overview.

path exists. Then, the first step of the method analyzes individually each update the rule performs to see if it may violate some integrity constraint of the target metamodel. Next, the second step performs a contextual analysis of each potentially violating update to see if other updates in the rule compensate its effect to ensure that we may generate a consistent target model. Finally, the third step of the method classifies the rule depending on the results obtained in the previous step: if all potentially violating updates can be discarded we can conclude the rule is WE; otherwise, the rule is non-WE (in fact, it is non-executable) and the method returns a feedback in order to repair the rule.

In the following we describe in more detail each step, using an example to illustrate the whole process.

Step 1: Analyzing the existence of Potentially Violating Updates. First step of our verification method analyzes each update in the rule to see if its effect creates a target model element that can violate some integrity constraints of the target metamodel. If so, this update is declared as *Potentially Violating Update* (PVU) and we refer to the constraints the PVU can violate as *Susceptible Violated Constraints* (SVCs). If the rule has no PVUs, it is WE. Otherwise, we need to continue the analysis with the next step.

In order to detect the PVUs we have adapted the rules presented in Chapter 6 to automatically determine the integrity constraints of the target metamodel each update may violate. Table 8.1 shows these rules. First column (*Susceptible Violated Constraint (SVC)*) shows each constraint our method supports[18] and second column (*Potentially Violation Updates (PVUs)*)

[18]The constraints covered by this method are: Mand(`attr`,`cl`); Cmin(`cl`) and Sym(`as`). Note that additional constraints could be considered adapting the rules presented in Chapter 6.

determines the updates in the ATL rule that may violate each constraint. All PVUs are expressed textually in ATL language.

Table 8.1. Rules to determine the ATL updates that may violate each integrity constraint.

	Susceptible Violated Constraint (SVC)	*Potentially Violating Updates (PVUs)*
1	Mand(`attr`,`cl`)	`x:targetMM!cl`
		`x: targetMM!cl'`, where `cl'` is a subclass of `cl` in the target metamodel
2	Cmin(`as`,`r`)≠0	`x:targetMM!cl`, where `cl` (or one of its superclasses) participates on the association `as` with role `r'` (`r'` is the opposite role to `r` in `as`)
3	Sym(`as`)	`x:... r <- y`, where `r` represents any member end of the association `as`

In the following we discuss each row of Table 8.1:

- First row determines the updates that may violate a mandatory attribute constraint (Mand(`attr`,`cl`)). This constraint will be violated when we create an object of class `cl` (first subrow) or of a subclass of `cl` (second subrow) in the target model.
- Second row determines the update that may violate a minimum cardinality constraint of an association `as` in the role `r` when it is different to zero (Cmin(`as`,`r`)≠0). This constraint will be violated when we create a new object of class `cl` in the target model (where `cl` or one of its superclasses participates on the association `as` with role `r'`, and `r'` is the opposite role to `r` in `as`).
- Third row determines the update that may violate a symmetric constraint of a recursive association (Sym(`as`)). This constraint will be violated when we create a link of `as` in the target model.

In this first step, the above rules are applied over all the integrity constraints that appear in the target metamodel. As a result, we obtain a set of potentially violating updates that may violate the existing integrity constraints in the target metamodel.

Example 57 Applying the rules of Table 8.1 to the target metamodel of Figure 8.13 we obtain the set of possible PVUs derived from the target metamodel. Table 8.2 shows, for each SVC, the set of PVUs that may violate it.

Then, we may determine if a rule r contains PVUs by comparing the set of updates of Table 8.2 with the set of updates which appear in r. All updates in the intersection of both sets are PVUs. As explained in Chapter 6, all the instance-level parameters of the generic PVU (i.e. variables `x`, `y`, ...) may be bound to any concrete value in the rule.

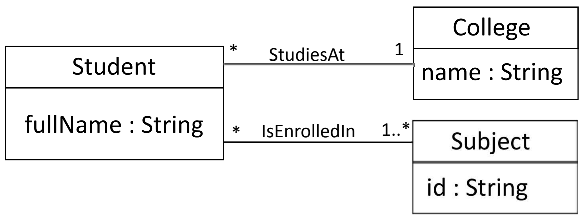


Figure 8.13. `StudentMM` (target metamodel).

Table 8.2. PVUs derived from the target metamodel of Figure 8.5 according to Table 8.1.

SVCs	*PVUs*
Mand(`fullName`, `Student`)	`x:StudentMM!Student`
Mand(`name`, `College`)	`x:StudentMM!College`
Mand(`id`, `Subject`)	`x:StudentMM!Subject`
Cmin(`StudiesAt`, `college`)=1	`x:StudentMM!Student`
Cmin(`IsEnrolledIn`, `subject`)=1	`x:StudentMM!Student`

Example 58 In the following we show the PVUs for the rule `Person2Student`. This rule contains two PVUs. First, the creation of a new student (update `s:StudentMM!Student`) may violate one mandatory constraint (when the attribute `fullName` is not initialized) and two minimum cardinality constraints (when the new student is not linked to any college or subject). Second, the creation of a new college (update `c:StudentMM!College`) may violate one mandatory constraint (when the attribute `name` is not initialized).

Potentially Violating Updates (PVUs) of the rule `Person2Student` and Susceptible Violating Constraints (SVC) they may violate:

- PVU_1: `s:StudentMM!Student`
  - $SVC_{1.1}$: Mand(`fullName`, `Student`)
  - $SVC_{1.2}$: Cmin(`StudiesAt`, `College`)
  - $SVC_{1.3}$: Cmin(`IsEnrolledIn`, `Subject`)
- PVU_2: `c:StudentMM!College`
  - $SVC_{2.1}$: Mand(`name`, `College`)

Since the rule `Person2Student` is susceptible to be non-WE (given that it contains several PVUs that may affect its executability) we must proceed with the second step of our method.

Step 2: Discarding Potentially Violating Updates. Similarly to what we explained in Chapter 6, it may happen that the context in which a PVU is executed within the transformation rule guarantees that the effect of the PVU is not going to actually violate any of its SVCs. In these cases, the PVU may be discarded.

In this second step, our method analyzes the set of PVUs returned by the previous step and tries to discard them. If all PVUs that may compromise the WE of the rule can be discarded, then it is classified as WE. If not, the rule is marked as non-WE and the corresponding corrective feedback is provided.

Note that, if a PVU may violate several SVCs, the PVU may be discarded iff it satisfies all conditions to avoid violating each SVC.

Table 8.3 describes the conditions that the ATL rule must satisfy in order to discard a specific PVU when it may violate a concrete SVC. Conditions are described in two equivalent ways: (1) as an ATL pattern; and (2) using a textual description.

Table 8.3. Necessary conditions to discard the PVUs that may affect the weak executability of an ATL rule.

PVU	*SVC*	*Conditions to discard the PVU when it may violate the SVC*	
`o:targetMM!cl`	Mand(`attr`,`cl`)	*ATL pattern*	`o:... attr<-#`
		Description	The rule includes an update to initialize the attribute `attr` of the object `o`.
	Cmin(`as`,`r`)	*ATL pattern*	`o:... r<-x`, where `x->size()`≥Cmin(`as`,`r`)
		Description	The rule includes at least Cmin(`as`,`r`) updates to create a link of `as` between the new object `o` and another object (with role r').
`o:...` r `<- x`	Sym(`as`)	*ATL pattern*	`o:... r'<-y`
		Description	The rule includes an update to create the symmetric link.

Example 59 In the following we try to discard the PVUs identified previously.

As we justified in the previous step, the rule `Person2Student` has two PVUs. According to Table 8.3, in order to discard the PVU_1 (`s:StudentMM!Student`) when it may violate the $SVC_{1.1}$ (Mand(`fullName`,`Student`)), the rule must include an update to initialize the attribute `fullName` of the student `s`. The rule contains this initialization (`s:... fullName<-p.getFullName()`), then, we can ensure the PVU_1 will not violate the $SVC_{1.1}$.

On the other hand, in order to discard the same PVU when it may violate the $SVC_{1.2}$ (Cmin(`StudiesAt`,`College`)), the rule must include at least one update to create a link of `StudiesAt` between `s` and a college. The rule contains this link creation (`s:... college<-c`), then, we can ensure the PVU_1 will not violate the $SVC_{1.2}$. Similarly, in order to discard the same PVU when it may violate the $SVC_{1.3}$ (Cmin(`IsEnrolledIn`,`Subject`)), the rule must include at least one update to create a link of `IsEnrolledIn` between `s` and a subject. The rule does not contain this link creation, then, the $SVC_{1.3}$ will be always violated for the rule `Person2Student`.

Finally, in order to discard the PVU_2 (`s:StudentMM!College`) when it may violate the $SVC_{2.1}$ (Mand(`name`,`College`)), the rule must include an update to initialize the attribute `name` from the college `c`. The rule contains this initialization (`c:... name<-p.collegeName`), then, we can ensure the PVU_2 will not violate the $SVC_{2.1}$.

Conditions to discard the PVUs of rule `Person2Student`:

- PVU_1 (`s:StudentMM!Student`), $SVC_{1.1}$ (Mand(`fullName,Student`)):
  The rule includes an update to initialize the attribute `fullname` of the object `s`.
  {satisfied by the update `s:... fullName<-p.getFullName()`}

- PVU_1 (`s:StudentMM!Student`), $SVC_{1.2}$ (Cmin(`StudiesAt,College`)):
  The rule includes at least one update to create a link of `StudiesAt` between `s` and a college.
  {satisfied by the update `s:... college<-c`}

- PVU_1 (`s:StudentMM!Student`), $SVC_{1.3}$ (Cmin(`IsEnrolledIn,Subject`)):
  The rule includes at least one update to create a link of `IsEnrolledIn` between `s` and a subject.
  {not satisfied}

- PVU_2 (`s:StudentMM!College`), $SVC_{2.1}$ (Cmin(`StudiesAt,College`)):
  The rule includes an update to initialize the attribute `name` of the object `c`.
  {satisfied by the update `c:... name<-p.collegeName`}

In summary, all PVUs may be discarded except the PVU_1, since it may violate the $SVC_{1.3}$.

Step 3: Classifying the ATL rule. Last step of our method classifies the ATL rule depending on the results obtained in the previous step. If all the PVUs may be discarded, the rule is WE.

Otherwise, the rule is classified as non-WE (and, in fact, it is non-executable). If this is the case, our method returns as feedback the conditions that should be added in the rule in order to make it WE.

Example 60 Since the rule `Person2Student` contains a PVU which cannot be discarded (see PVU_1 and $SVC_{1.3}$), this rule is non-WE.

In order to make the rule WE, the method suggests that the rule should include at least one update to create a link of the `IsEnrolledIn` association between the student `s` and a subject.

In the following we show the same rule once the suggested update (emphatized in bold type) has been added. Note that, besides the creation of a link between `s` and a subject, we have also added the proper updates to create a subject and initialize its `id` attribute.

```
module Person2Student:
  create OUT: StudentMM from IN: PersonMM

 helper context PersonMM!Person def:
  getFullName():  String = self.name + ' ' + self.surname;

 rule Person2Student {
  from p:  PersonMM!Person
  to s:  StudentMM!Student (
   fullName <- p.getFullName(),
   college <- c,
   subject <- sj
  ),
  c:  StudentMM!College (
   name <- p.collegeName
  )
  sj:  StudentMM!Subject (
   id <- ``default''
  )
 }
```

8.2.2 Completeness of ATL rule's set

We consider a set of ATL rules is complete[19] if it allows addressing all elements of the source and target metamodels participating in the ATL transformation. Then, this property may be viewed regarding two perspectives: *source-completeness* and *target-completeness*.

[19]This property can also be found in the literature as *coverage*.

Source-completeness

We consider a set of ATL rules is *source-complete* when all the elements of the source metamodel may be navigated through the execution of these rules. Otherwise, there will be elements of the source metamodel with no relevance in the transformation.

Example 61 The set composed by the single rule `Person2Student` is not source-complete since, for instance, rules to navigate the `age` attribute and the `Knows` association are not specified.

Target-completeness

We consider a set of ATL rules is *target-complete* when all the elements of the target metamodel may be created and initialized through the execution of these rules. Otherwise, there will be elements of the target metamodel that users will not be able to create/initialize since any ATL rule addresses their treatment.

Example 62 As an example, the set composed by the single rule `Person2Student` is not target-complete since, for instance, rules to create objects of type `Subject` are not specified, forbidding users to create new subjects on the target model.

A more formal definition of this property may be found in Chapter 7.

Verifying the source and target completeness of an ATL rule's set

To determine whether a set of ATL rules is complete we propose applying a four step process (see Figure 8.14). This process may be automated and integrated into a tool for editing ATL rules.

Step 1: Computing the metamodel elements (ME). First step consists in determining the metamodel elements that should be addressed by the rules. When verifying the source-completeness, the metamodel elements are all those elements of the source metamodel that should be *navigated* through the rules, i.e. classes, attributes and associations of the source metamodel. On the other hand, when verifying the target-completeness, the metamodel elements are all those elements of the target metamodel that should be *created* or *initialized*, i.e. non-abstract classes, classes which are not the supertype of a complete generalization set, non-derived attributes and non-derived associations of the target metamodel.

Example 63 First column of Tables 8.4 and 8.5 shows the metamodel elements for our running source and target metamodels respectively.

Step 2: Computing the addressed elements (AE). Second step consists in determining the addressed elements by the rules set. When verifying the source-completeness, the addressed

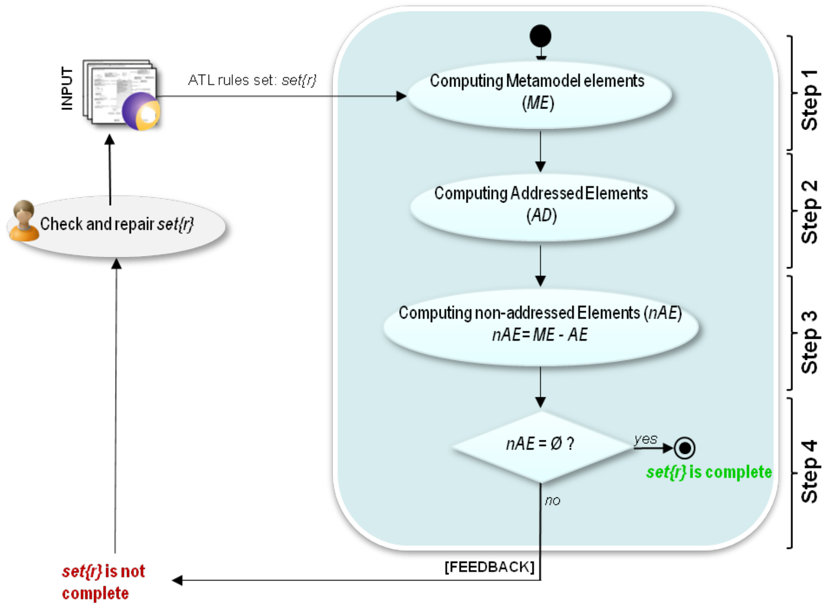


Figure 8.14. Completeness of ATL rules set method overview.

elements are those which are navigated in the rules, i.e. those elements which appear in the `from` part of some rule or those which are navigated in the `to` part of some rule. On the other hand, when verifying the target-completeness, the addressed elements are those which are created or initialized in the `to` part of some rule.

Example 64 Second column of Tables 8.4 and 8.5 shows the addressed elements (by the rule `Person2Student`) for our running source and target metamodels respectively. For each addressed element, we show the update in the rule which addresses that element.

Step 3: Computing the non-addressed elements (non-AE). Third step consists in computing the non-addressed elements, that is, those elements that should be addressed in some rule but they are not treated in any rule. In order to obtain the non-addressed elements, our method computes the difference between the set of the metamodel elements (ME) (those computed in the Step 1) and the set of addressed elements (AE) (those computed in the Step 2) and returns a (maybe empty) subset of non-addressed elements (non-AE).

Step 4: Classifying the ATL rules set. Last step of our method classifies the ATL rules set depending on the result obtained in the previous step. If the set of non-addressed elements (non-AE) is empty, the sets ME (metamodel elements) and AE (addressed elements) contain exactly the same elements (i.e. all metamodel elements are addressed by the ATL rules of the input set). Then, the ATL rules set is complete. Otherwise, this set is incomplete and our method returns as feedback the subset of updates that should be added in some rule to make the set complete.

Example 65 All elements that contain "No" in the second column of Tables 8.4 and 8.5 are non-addressed elements of our running source and target metamodels respectively.

Example 66

Tables 8.4 and 8.5 show the source and target completeness of our running example.

Table 8.4. Source-completeness example.

Metamodel elements (from the source metamodel `PersonMM`)	*Is addressed (i.e. navigated)?*
class `Person`	Yes (update `p:PersonMM!Person` of rule `Person2Student`)
attribute `name` (from `Person`)	Yes (update `fullName<-p.getFullName()` of rule `Person2Student`)
attribute `surname` (from `Person`)	Yes (update `fullName<-p.getFullName()` of rule `Person2Student`)
attribute `age` (from `Person`)	No
attribute `collegeName` (from `Person`)	Yes (update `name<-p.collegeName` of rule `Person2Student`)
association `Knows` (from `Person` to `Person`)	No

The set composed by the single rule `Person2Student` is not source-complete since there is no rule that allows to navigate the `age` attribute and the `Knows` association of the source metamodel.

Table 8.5. Target-completeness example.

Metamodel elements (from the target metamodel `StudentMM`)	*Is addressed (i.e. navigated)?*
`Student` class	Yes (update `s:StudentMM!Student` of rule `Person2Student`)
attribute `fullName` (from `Student`)	Yes (update `fullName<-p.getFullName()` of rule `Person2Student`)
class `College`	Yes (update `c:StudentMM!College` of rule `Person2Student`)
attribute `name` (from `College`)	Yes (update `name<-p.collegeName` of rule `Person2Student`)
class `Subject`	No
attribute `id` (from `Subject`)	No
association `StudiesAt` (from `Student` to `College`)	Yes (update `college<-c` of rule `Person2Student`)
association `IsEnrolledIn` (from `Student` to `Subject`)	No

On the other hand, the set composed by the same rule is not target-complete since there is no rule that allows to create objects of type `Subject`, links of `IsEnrolledIn` association neither initialize the `id` attribute from `Subject`.

In order to make the set source and target complete, new updates addressing these missing elements should be added in some rule.

8.3 Verifying Graph Transformation rules

In this section we adapt the method we presented in Chapter 6 to verify the weak executability of Graph Transformation rules. We do not present the specific method to verify the completeness of a set of Graph Transformation rules, because it does not have significant differences from the method presented in Chapter 6.

8.3.1 Weak executability of Graph Transformation rules

We consider a graph transformation rule r is *weakly executable* (WE) if it has a chance of being successfully executed. That is, if we can find at least one host graph G on which r can be applied and the direct derivation $G =>_r H$ generates a graph H consistent with the system's integrity constraints. Otherwise r is useless, as every time it is executed, an error arises because H violates some integrity constraints.

A more formal definition of this property may be found in Chapter 6.

Example 67 Rule `newMachine` is not weakly executable since every time we create a new machine and we do not associate it to any output conveyor, we reach an erroneous state where the minimum 1 cardinality of the `Input` association in the role `in` (see Figure 8.16) is violated. As we will see later, in this case, our method reports that, in order to create a new machine, we need linking the new machine with an input conveyor and updating its `busy` attribute within the same rule execution.

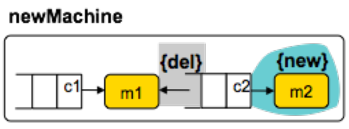
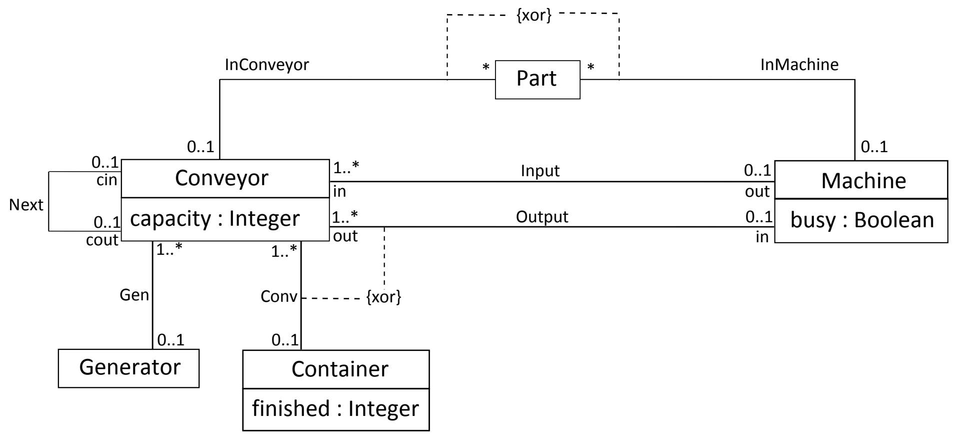


Figure 8.15. Rule `newMachine`.

Example 68 Instead, rule `startMachine` is weakly executable since we are able to find an execution scenario where we can successfully move a part from a conveyor to a machine.

Example 69 Rules `optimize` and `disconnectGenerator`, on the other hand, are not weakly executable since, although they do not violate any constraint, the single scenario in which they could be successfully applied is forbidden by the rule definition.



context Generator **inv** numberOfGenerators: Generator.allInstances()->size()=1
context Conveyor **inv** minimumCapacity: self.capacity >0 and self.part->size()<=self.capacity
context Machine **inv** busyMachine: if (self.busy=false) then self.part->size()=0 else self.part->size()>0 endif

Figure 8.16. Metamodel.

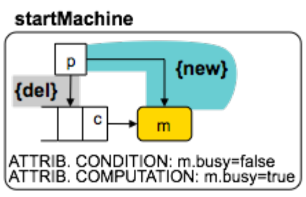

startMachine
p
{new}
{del}
c
m
ATTRIB. CONDITION: m.busy=false
ATTRIB. COMPUTATION: m.busy=true

Figure 8.17. Rule `startMachine`.

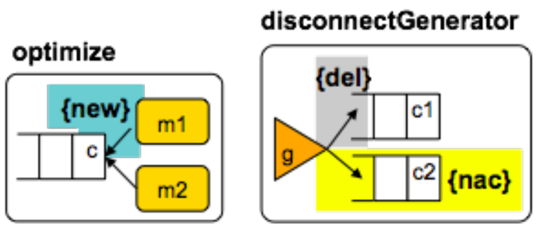


Figure 8.18. Rules `optimize` and `disconnectGenerator`.

Verifying the weak executability of a Graph Transformation rule

To determine whether a Graph Transformation rule is weakly executable we adapt the method we presented in Chapter 6 to address graph transformations instead of action-based operations.

In this case, our lightweight static method (see Figure 8.19) takes as input a metamodel on which the graph transformation rule is defined and the Graph Transformation rule itself. As before, our method returns either a positive answer, meaning that the graph transformation rule is WE, or a corrective feedback, consisting in a set of updates that should be added to the rule in order to make it WE.

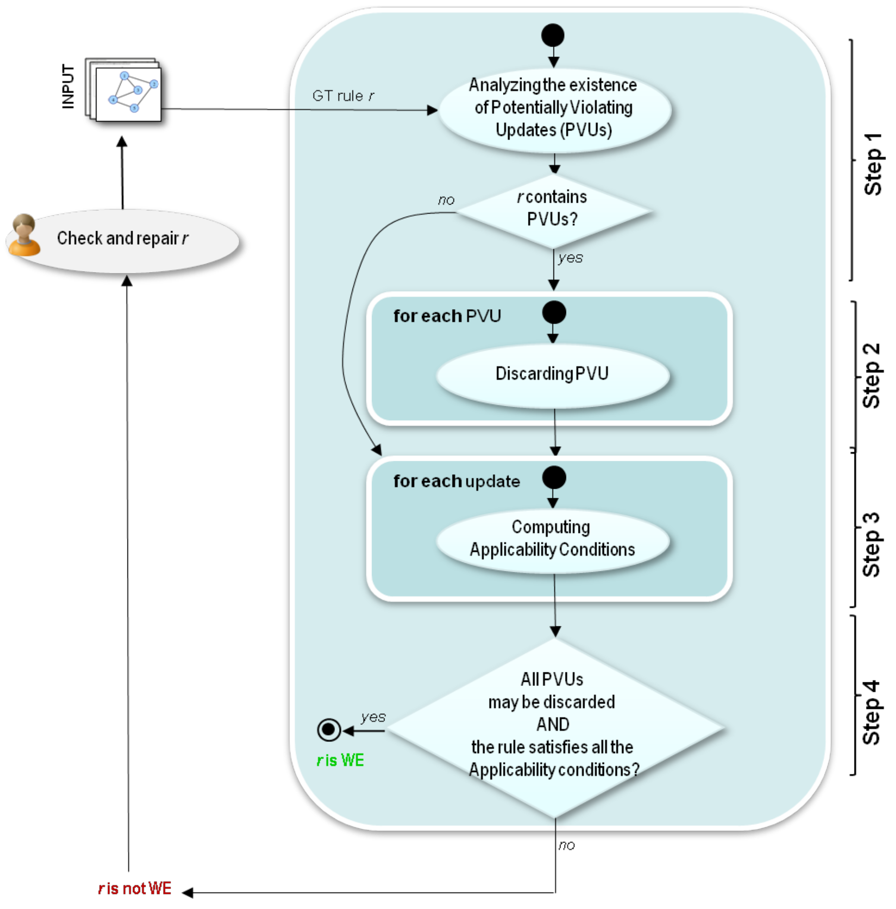

INPUT
GT rule r
Analyzing the existence of Potentially Violating Updates (PVUs)
r contains PVUs?
no
yes
Check and repair r
for each PVU
Discarding PVU
for each update
Computing Applicability Conditions
All PVUs may be discarded AND the rule satisfies all the Applicability conditions?
yes
r is WE
no
r is not WE
Step 1
Step 2
Step 3
Step 4

Figure 8.19. Weak executability of Graph Transformation rules method overview.

When verifying Graph Transformation rules, the method presented in Chapter 6 must be adapted in several ways. On the one hand, since Graph Transformation rules do not allow forking the execution of the rules, a single execution path exists, hence, there is not need to compute several alternative execution paths. Then, the first step of the method analyzes individually each update the rule performs to see if it may violate some integrity constraint of the metamodel. Next, the second step performs a contextual analysis of each potentially violating update to see if other updates in the rule compensate its effect to ensure that we may generate a consistent target model. On the other hand, afterward the above process it is necessary to add an additional third step to check that at least one of the safe scenarios in which the Graph Transformation rule will leave the system in a consistent state can actually be a match for the rule considering its LHS and NACs. Finally, fourth step of the method classifies the rule depending on the results obtained in the previous step: if all potentially violating updates can be discarded and there is a valid match in which the rule can be applied, we can conclude the rule is WE; otherwise, the rule is non-WE (in fact, it is non-executable) and the method returns a feedback in order to repair it.

Step 1: Analyzing the existence of Potentially Violating Updates

First step of our verification method analyzes each update in the rule to see if its effect can update the target model in a way that some element may violate some integrity constraints of the metamodel. If so, this update is declared as *Potentially Violating Update* (PVU) and we refer to the constraints the PVU can violate as *Susceptible Violated Constriants* (SVCs). If the rule has no PVUs, it is WE. Otherwise, we need to continue the analysis with the next step.

In order to detect the PVUs we adapt the rules presented in Chapter 6 that automatically determine the integrity constraints of the metamodel each update may violate. Table 8.6 shows these rules. First column (*Susceptible Violated Constraint (SVC)*) shows each constraint our method supports[20] and second column (*Potentially Violation Updates (PVUs)*) determines the updates in the graph transformation rule that may violate each constraint.

In the following we discuss each row of Table 8.6:

1. First row determines the update that may violate a minimum cardinality constraint of a class `cl` when it is different to zero (Cmin(`cl`)≠0). This constraint will be violated when we destroy an object of class `cl`, that is, when the number of instances of `cl` is decreased. Note that this rule only applies when Cmin(`cl`) = Cmax(`cl`) since for the rest of the situations we are always able to find a scenario in which the constraint will not be violated.

2. Second row determines the update that may violate a maximum cardinality constraint of a class `cl` when it is different to "*" (Cmax(`cl`)≠*). This constraint will be violated when we create an object of class `cl`. As before, note that this rule only applies when Cmin(`cl`) = Cmax(`cl`) since for the rest of the situations we are always able to find a scenario in which the constraint will not be violated.

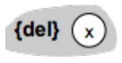
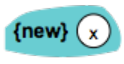
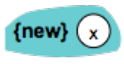
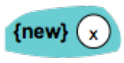
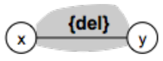
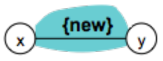
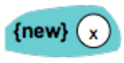
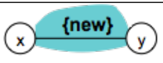
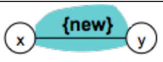
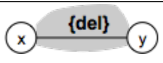
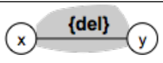
3. Third row determines the update that may violate a mandatory attribute constraint (Mand(`attr`,`cl`)). This constraint will be violated when we create an object of class `cl`.

4. Fourth row determines the updates that may violate a minimum cardinality constraint of an association `as` in the role `r` when it is different to zero (Cmin(`as`,`r`)≠0). This constraint will be violated when we create a new object of class `cl` (where `cl` participates on the association `as` with role `r'`, and `r'` is the opposite role to `r` in `as`) (first subrow). Additionally, if Cmin(`as`,`r`) = Cmax(`as`,`r`), this constraint will be also violated when we destroy a link of `as` (second subrow).

5. Fifth row determines the update that may violate a maximum cardinality constraint of an association `as` in the role `r` when it is different to "*" (Cmax(`as`,`r`)≠*). If Cmin(`as`,`r`) = Cmax(`as`,`r`), this constraint will be violated when we create a link of `as`.

[20]The constraints covered by this method are: Mand(`attr`,`cl`), Cmin(`as`,`r`), Sym(`as`), Subset(`r,as,r',as'`) (which indicates that the objects with role `r'` in the association `as'` are a subset of the objects with role `r` in the association `as`, taking the same departing instance at the opposite ends) and Xor(`as,as',cl`) (to represent that the associations `as` and `as'` share one participant of class `cl`, and any instance of the shared class may have links from only one of the associations [128]. In this thesis we restrict to *xor* constraints where the participant associations have multiplicity 0..1 in the non-shared role). Note that additional constraints could be considered adding the rules presented in Chapter 6.

Table 8.6. Rules to determine the Graph Transformation updates that may violate each integrity constraint.

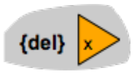
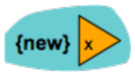
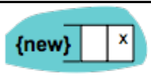
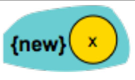
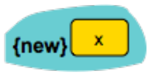
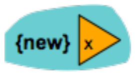
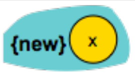
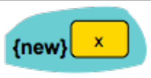
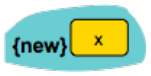
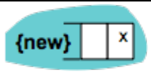
	Susceptible Violated Constraint (SVC)	*Potentially Violating Updates (PVUs)*	
		Graphical pattern	*Description*
1	Cmin(`cl`)≠0 (only applies when Cmin(`cl`) = Cmax(`cl`))	 where `x` is an instance of `cl`	Destroy an object of class `cl`
2	Cmax(`cl`)≠* (only applies when Cmin(`cl`) = Cmax(`cl`))	 where `x` is an instance of `cl`	Create a new object of class `cl`
3	Mand(`attr`,`cl`)	 where `x` is an instance of `cl`	Create a new object of class `cl`
4	Cmin(`as`,`r`)≠0	 where `x` is an instance of `cl` and `cl` participates on `as`	Create a new object of class `cl`, where `cl` participates on the association `as` with role `r′` (`r′` is the opposite role to `r` in `as`)
	Cmin(`as`,`r`)≠0 (only applies when Cmin(`as`,`r`) = Cmax(`as`,`r`))	 where the destroyed link belongs to `as`	Destroy a link of the association `as` between two objects
5	Cmax(`as`,`r`)≠* (only applies when Cmin(`as`,`r`) = Cmax(`as`,`r`))	 where the created link belongs to `as`	Create a link of the association `as` between two objects
6	Xor(`as`,`as′`,`r`)	 where `x` is an instance of `cl` and `cl` is the shared participant between `as` and `as′`	Create an object of class `cl`, where `cl` is the shared participant in the associations `as` and `as′`
		 where the created link belongs to `as`	Create a link of the association `as` between two objects
		 where the created link belongs to `as′`	Create a link of the association `as′` between two objects
		 where the destroyed link belongs to `as`	Destroy a link of the association `as` between two objects
		 where the destroyed link belongs to `as′`	Destroy a link of the association `as′` between two objects

6. Finally, sixth row determines the update that may violate a xor constraint between two associations (Xor(`as`,`as'`,`cl`)). This constraint will be violated when we create an object of class `cl` (first subrow); or when we create/destroy a link of one of the associations that participate in the constraint (second to fifth subrows).

In this first step, the above rules are applied over all the integrity constraints that appear in the metamodel. As a result, we obtain a set of potentially violating updates that may violate the integrity constraints existing in the metamodel.

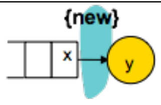
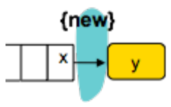
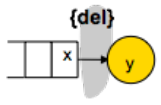
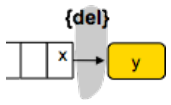
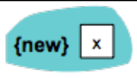
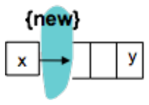
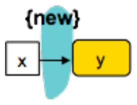
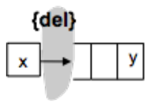
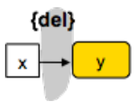
Example 70 Applying the rules of Table 8.6 to the metamodel of Figure 8.16 we obtain the set of possible PVUs. Table 8.7 shows, for each SVC, the set of PVUs that may violate it.

Table 8.7. PVUs derived from the metamodel of Figure 8.16 according to Table 8.6.

SVCs	*PVUs*	
	Graphical pattern	*Description*
Cmin(`Generator`)=1 (since Cmin(`Generator`) = Cmax(`Generator`))		Destroy an object of class `Generator`
Cmax(`Generator`)=1 (since Cmin(`Generator`) = Cmax(`Generator`))		Create a new object of class `Generator`
Mand(`capacity`,`Conveyor`)		Create a new object of class `Conveyor`
Mand(`finished`,`Container`)		Create a new object of class `Container`
Mand(`busy`,`Machine`)		Create a new object of class `Machine`
Cmin(`Gen`,`conveyor`)=1		Create a new object of class `Generator`
Cmin(`Conv`,`conveyor`)=1		Create a new object of class `Container`
Cmin(`Input`,`in`)=1		Create a new object of class `Machine`
Cmin(`Output`,`out`)=1		Create a new object of class `Machine`
Xor(`Conv`,`Output`,`conveyor`)		Create an object of class `Conveyor`

Continued on next page

Table 8.7 – continued from previous page

SVCs	*PVUs*	
	Graphical pattern	*Description*
		Create a link of the association `Conv` between a conveyor and a container
		Create a link of the association `Output` between a conveyor and a machine
		Destroy a link of he association `Conv` between a conveyor and a container
		Destroy a link of the association `Output` between a conveyor and a machine
Xor(`InConveyor`, `InMachine`, `part`)		Create an object of class `Part`
		Create a link of the association `InConveyor` between a part and a conveyor
		Create a link of the association `InMachine` between a part and a machine
		Destroy a link of the association `InConveyor` between a part and a conveyor
		Destroy a link of he association `InMachine` between a part and a machine

Then, we may determine whether a rule r contains PVUs by comparing the above set of updates with the set of updates appearing in r. All updates in the intersection of both sets are PVUs. As explained in Chapter 6, all the instance-level parameters of the generic PVU (i.e. variables `x`, `y`, ...) may be bound to any concrete value in the rule.

Example 71 In the following we show the PVUs for the rule `newMachine`.

This rule contains three PVUs (see Figure 8.20). First, creating a new machine (PVU_1, where the machine `m2` maps to the free variable `x` in the generic PVU) violates one mandatory constraint (when the attribute `busy` is not initialized) and two minimum cardinality constraints (when the new machine is not linked to any conveyor according to the `Input` and `Output` associations). Second, creating a new link of the association `Output` (PVU_2, where `c2` maps to `x` and `m2` maps to `y`) violates the xor constraint between the associations `Output` and `Conv`. Finally, destroying a link of the association `Output` (PVA_3, where `c2` maps to `x` and `m1` maps to `y`) violates the same constraint as before.

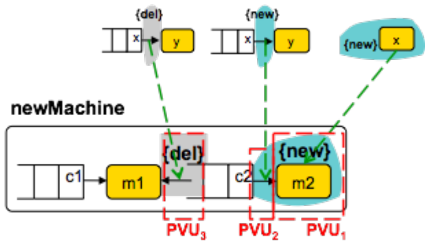


Figure 8.20. PVUs for the rule `newMachine`.

Potentially Violating Updates (PVUs) of rule `newMachine` and Susceptible Violating Constraints (SVC) they may violate:

- PVU_1: Creating a new machine `m2`
  - $SVC_{1.1}$: Mand(`busy`,`Machine`)
  - $SVC_{1.2}$: Cmin(`Input`,`in`)
  - $SVC_{1.3}$: Cmin(`Output`,`out`)
- PVU_2: Creating a new link of the association `Output` between the conveyor `c2` and the machine `m2`
  - $SVC_{2.1}$: Xor(`Conv`,`Output`,`conveyor`)
- PVU_3: Destroying a link of the association `Output` between the conveyor `c2` and the machine `m1`
  - $SVC_{3.1}$: Xor(`Conv`,`Output`,`conveyor`)

Since the rule `newMachine` is susceptible to be non-WE (given that it contains three PVUs that may affect its executability) we must proceed with the second step of our method.

Example 72 In the following we show the PVUs for the rule `startMachine`. This rule contains two PVUs (see Figure 8.21). First, the creation of a new link of the `InMachine` association (PVU_1, where `p` maps to `x` and `m` maps to `y`) violates the xor constraint between the associations `InConveyor` and `InMachine` (when the same part is already linked to a conveyor). Similarly, the destruction of an existing link of the `InConveyor` association (PVU_2, where `p` maps to `x` and `c` maps to `y`) violates the same xor constraint (when the same part is not linked to any machine).

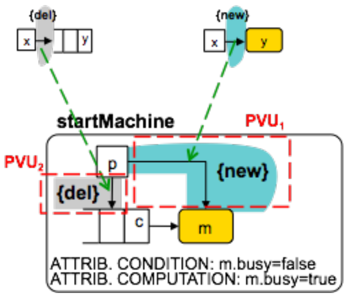

{del}
x
y
{new}
x
y
startMachine
PVU1
PVU2
p
{new}
{del}
c
m
ATTRIB. CONDITION: m.busy=false
ATTRIB. COMPUTATION: m.busy=true

Figure 8.21. PVUs for the rule `startMachine`.

Potentially Violating Updates (PVUs) of rule `startMachine` and Susceptible Violating Constraints (SVC) they may violate:

- PVU_1: Creating a new link of `InMachine` between the part `p` and the machine `m`
  $SVC_{1.1}$: Xor(`InConveyor`,`InMachine`,`part`)
- PVU_2: Destroying an existing link of `InConveyor` between the part `p` and the conveyor `c`
  $SVC_{2.1}$: Xor(`InConveyor`,`InMachine`,`part`)

Since the rule `startMachine` is susceptible to be non-WE (given that it contains two PVUs that may affect its executability) we must proceed with the second step of our method.

Example 73 Rules `optimize` and `disconnectGenerator` do not contain PVUs. Then, we can skip the second step of the method (which consists in discarding the PVUs), but we must check the applicability conditions (see Step 3) in order to conclude whether they are WE.

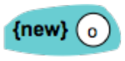
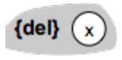
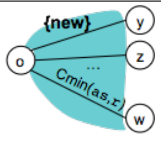
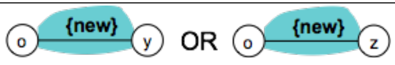
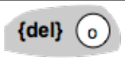
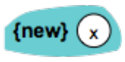
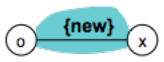
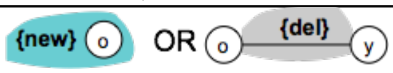
Step 2: Discarding Potentially Violating Updates

Similarly to what we explained in Chapter 6, it may be happen that the context in which a PVU is executed within the transformation rule guarantees that the effect of the PVU is not going to actually violate any of its SVCs. In these cases, the PVU may be discarded.

In this second step, our method analyzes the set of PVUs returned by the previous step and tries to discard them. As before, note that, if a PVU may violate several SVCs, the PVU may be discarded iff it satisfies all conditions to avoid violating each SVC.

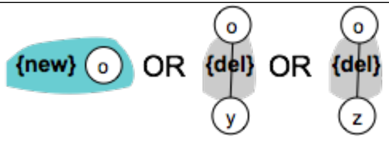
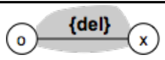
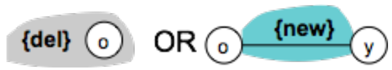
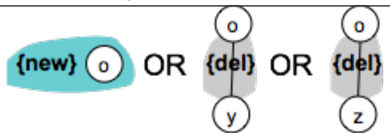
Table 8.8 describes the conditions that the Graph Transformation rule must satisfy in order to discard a specific PVU when it may violate a concrete SVC.

Table 8.8. Necessary conditions to discard the PVUs that may affect the weak executability of a Graph Transformation rule.

PVU	*SVC*	*Conditions to discard the PVU when it may violate the SVC*	
 where x is an instance of `cl`	Mand(`attr`,`cl`)	*Graphical pattern*	ATTRIB. COMPUTATION: o.attr = #
		Description	The rule includes an update to initialize the attribute `attr` of the object `o`
	Cmax(`cl`)	*Graphical pattern*	 where `x` is an instance of `cl`
		Description	The rule includes an update to destroy an object of class `cl`
	Cmin(`as`,`r`)	*Graphical pattern*	 where each link belongs to `as`
		Description	The rule includes Cmin(`as`,`r`) updates to create a link of `as` where the object `o` participates
	Xor(`as`,`as'`,`r`)	*Graphical pattern*	 where the link between `o` and `y` belongs to `as` and the link between to `o` and `z` belongs to `as'`
		Description	The rule includes an update to create a link of `as` OR of `as'` where the object `o` participates
 where `o` is an instance of `cl`	Cmin(`cl`)	*Graphical pattern*	 where `x` is an instance of `cl`
		Description	The rule includes an update to create an object of class `cl`
 where the created link belongs to `as`	Cmax(`as`,`r`)	*Graphical pattern*	 where the link between `o` and `y` belongs to `as`

Continued on next page

Table 8.8 – continued from previous page

PVU	*SVC*	*Conditions to discard the PVU when it may violate the SVC*	
		Description	The rule includes an update to create the object `o` OR to destroy a link of `as` between `o` and another object
	Xor(`as`,`as'`,`r`)	*Graphical pattern*	 where the link between `o` and `y` belongs to `as` and the link between `o` and `z` belongs to `as'`
		Description	The rule includes an update to create the object `o` OR to destroy a link of `as` in which `o` participates OR to destroy a link of `as'` in which `o` participates
 where the destroyed link belongs to `as`	Cmin(`as`,`r`)	*Graphical pattern*	 where the link between `o` and `y` belongs to `as`
		Description	The rule includes an update to destroy the object `o` OR to create a link of `as` between `o` and another object
	Xor(`as`,`as'`,`r`)	*Graphical pattern*	 where the link between `o` and `y` belongs to `as` and the link between `o` and `z` belongs to `as'`
		Description	The rule includes an update to destroy the object `o` OR to create a link of `as` in which `o` participates OR to create a link of `as'` in which `o` participates

Example 74 As we justified in the previous step, the rule `newMachine` has three PVUs. According to Table 8.8, in order to discard the PVU_1 (creating a new machine) when it may violate the $SVC_{1.1}$ (Mand(`busy`,`Machine`)), the rule must include an update to initialize the attribute `busy` of the machine `m2`. The rule does

not contain this initialization, then, the PVU_1 will always violate the $SVC_{1.1}$.

On the other hand, in order to discard the same PVU when it may violate the $SVC_{1.2}$ (Cmin(`Input`,`in`)), the rule must include at least an update to create a link of the `Input` association between `m2` and a conveyor. The rule does not contain this link creation, then, we can ensure the PVU_1 will always violate the $SVC_{1.2}$. Similarly, in order to discard the same PVU when it may violate the $SVC_{1.3}$ (Cmin(`Output`,`out`)), the rule must include at least an update to create a link of the `Output` association between `m2` and a conveyor. The rule contains this link creation, then, the PVU_1 regarding the $SVC_{1.3}$ may be discarded.

In order to discard the PVU_2 (creating a new link of the association `Output` between `c2` and `m2`) when it may violate the $SVC_{2.1}$ (Xor(`Conv,Output,conveyor`)), the rule must satisfy one of the following conditions: (1) it includes an update to create the object `c2`; or (2) it includes an update to destroy a link of the `Output` association in which `c2` participates; or (3) it includes an update to destroy a link of the `Conv` association in which `c2` participates. The rule satisfies the second condition, then, we can ensure the PVU_2 will not violate the $SVC_{2.1}$.

Similarly, in order to discard the PVU_3 (destroying a link of `Output` between `c2` and `m1`) when it may violate the $SVC_{2.1}$ (Xor(`Conv,Output,conveyor`)), the rule must satisfy one of the following conditions: (1) it includes an update to destroy the object `c2`; or (2) it includes an update to create a link of the `Output` association in which `c2` participates; or (3) it includes an update to destroy a link of the `Conv` association in which `c2` participates. The rule satisfies the second condition, then, we can ensure the PVU_3 will not violate the $SVC_{3.1}$.

Conditions to discard the PVUs of rule `newMachine`:

- PVU_1 (creating a new machine `m2`), $SVC_{1.1}$ (Mand(`busy,Machine`)):
  The rule includes an update to initialize the attribute `busy` of the machine `m2`.
  {not satisfied}

- PVU_1 (creating a new machine `m2`), $SVC_{1.2}$ (Cmin(`Input,in`)):
  The rule includes an update to create a link of `Input` where the machine `m2` participates.
  {not satisfied}

- PVU_1 (creating a new machine `m2`), $SVC_{1.3}$ (Cmin(`Output,out`)):
  The rule includes an update to create a link of `Output` where the machine `m2` participates.
  {satisfied, since the rule creates a link of `Output` between `c2` and `m2`}

- PVU_2 (creating a new link of the association `Output` between the conveyor `c2` and the machine `m2`), $SVC_{2.1}$ (Xor(`Conv,Output,conveyor`)):
  The rule includes an update to create the object `c2` OR to destroy a link of `Output` in which `c2` participates OR to destroy a link of `Conv` in which `c2` participates.
  {satisfied, since the rule destroys a link of `Output` between `c2` and `m1`}

- PVU_3 (destroying a link of the association `Output` between the conveyor `c2` and the machine `m1`), $SVC_{3.1}$ (Xor(`Conv,Output,conveyor`)):
  The rule includes an update to destroy the object `c2` OR to create a link of `Output` in which `c2` participates OR to destroy a link of `Conv` in which `c2` participates.
  {satisfied, since the rule creates a link of `Output` between `c2` and `m2`}

In summary, all PVUs may be discarded except the PVU_1, since it may violate the constraints $SVC_{1.1}$ and $SVC_{1.2}$.

Example 75 As we justified in the previous step, the rule `startMachine` has two PVUs. According to Table 8.8, in order to discard the PVU_1 (creating a new link of the association `InMachine` between `p` and `m`) when it may violate the $SVC_{1.1}$ (Xor(`InConveyor,InMachine,part`)), the rule must satisfy one of the following conditions: (1) it includes an update to create the object `p`; or (2) it includes an update to destroy a link of the `InConveyor` association in which `p` participates; or (3) it includes an update to destroy a link of the `InMachine` association in which `p` participates. The rule satisfies the second condition, then, we can ensure the PVU_1 will not violate the $SVC_{1.1}$.

Similarly, in order to discard the PVU_2 (destroying a link of `InConveyor` between `p` and `c`) when it may violate the $SVC_{2.1}$ (Xor(`InConveyor,InMachine,part`)), the rule must satisfy one of the following conditions: (1) it includes an update to destroy the object `p`; or (2) it includes an update to create a link of `InConveyor` in which `p` participates; or (3) it includes an update to destroy a link of `InMachine` in which `p` participates. The rule satisfies the second condition, then, we can ensure the PVU_2 will not violate the $SVC_{2.1}$.

Conditions to discard the PVUs of rule `newMachine`:

- PVU_1 (creating a new link of `InMachine` between the part `p` and the machine `m`), $SVC_{1.1}$ (Xor(`InConveyor,InMachine,part`)):
  The rule includes an update to create the object `p` OR to destroy a link of `InConveyor` in which `p` participates OR to destroy a link of `InMachine` in which `p` participates.
  {satisfied, since the rule destroys a link of `InConveyor` between `p` and `c`}

- PVU_2 (destroying a link of `InConveyor` between the part `p` and the conveyor `c`), $SVC_{2.1}$ (Xor(`InConveyor,InMachine,part`)):
  The rule includes an update to destroy the object `p` OR to create a link of `InConveyor` in which `p` participates OR to create a link of `InMachine` in which `p` participates.
  {satisfied, since the rule creates a link of `InMachine` between `p` and `m`}

In summary, all PVUs of the rule `startMachine` may be discarded.

Step 3: Checking applicability conditions

The discarding of all PVUs ensures that there are graphs for which the updates performed by the graph transformation rules do not induce any constraint violation. However, in addition, we have to make sure that at least one of these graphs is consistent with the LHS and NACs of the rule, that is, it can provide a valid match for the rule.

Example 76 The rule `optimize` has no PVUs but it is not weakly executable since the single scenario in which the rule could be successfully executed (the one in which the conveyor `c` is not the output of any machine before executing the rule) is forbidden by its LHS (which forces the conveyor to be the output of machine `m2` in order to be a match for the rule).

In this step we characterize the applicability conditions that guarantee the host graphs where we can successfully execute a rule also provide a valid match for the rule. Note that we do not check the general *applicability of the rule* but only whether some relevant scenarios are possible. The applicability conditions are defined as anti-patterns (i.e. a kind of graph constraints [56]) expressing conditions that cannot be found in the rule, as otherwise those patterns would exactly forbid the match that makes the rule weakly executable.

Example 77 In the previous `optimize` example, the anti-pattern would state that the LHS cannot contain a graph pattern including a link of the `Output` association between `c` and a machine.

Table 8.9 shows the conditions that describe the forbidden patterns. First column (*Update*) depicts the update that may generate a new applicability condition (where n represents the number of occurrences of the update in the rule). Second column (*When?*) indicates when the condition must be generated. Finally, last column (*Forbidden pattern*) expresses the condition by means of representing the forbidden pattern in the rule (where m represents the multiplicity of the opposite update[21]). Once instantiated for a particular rule, these patterns are presented to the user visually in the form of graph constraints.

Example 78 The rule `newMachine` must satisfy one applicability condition. Given that the rule contains the creation of a new link of the association `Output` and Cmax(`Output,in`)=1$\neq$*, the third row of Table 8.9 must be taken into account. Then, the rule cannot have more than Cmax(`Output,in`)-n+m=1-1+1=1 link of `Output` in its LHS. The rule contains one link (the link between `c2` and `m1`) in its LHS, then, the forbidden pattern does not exist and the applicability condition is satisfied.

Note that, otherwise, the destruction of the link of the association `Output` does not induce additional applicability conditions since the requisites stated in the fifth row (second column) of Table 8.9 do not hold.

Example 79 Similarly, the rule `startMachine` must satisfy one applicability condition. Given that the rule contains the creation of a new link of the association `InMachine` and Cmax(`InMachine,machine`)=1$\neq$*, the third row of Table 8.9 must be taken into account. Then, the rule cannot have more than Cmax(`InMachine,machine`)-n+m=1-1+0=0 links of `InMachine` in its LHS. The rule does not contain any link of `InMachine` in its LHS, then, the forbidden pattern does not exist and the applicability condition is satisfied.

Example 80 The rule `optimize` must satisfy one applicability condition. Given that the rule contains the creation of a new link of the association `Input` and Cmax(`Input,out`)=1$\neq$*, the third row of Table 8.9 must be taken into account. Then, the rule cannot have more than Cmax(`Input,out`)-n+m=1-1+0=0 links of `InMachine` in its LHS. The rule contains one link (the link between `m1` and `c`) in its LHS, then, the forbidden pattern appears and the applicability condition is not satisfied.

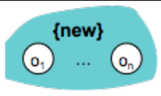
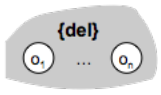
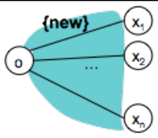
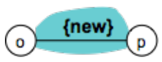
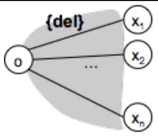
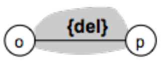
Example 81 The rule `disconnectGenerator` must satisfy one applicability condition. Given that the rule contains the destruction of a link of the association `Gen` and Cmin(`Gen,conveyor`)=1>0 and m=0<Cmin(`Gen,conveyor`)=1, the fifth row of Table 8.9 must be taken into account. Then, the rule cannot have a NAC with at least Cmin(`Gen,conveyor`)-m=1-0=1 links of the association `Gen` (excluding the link which is going to be destroyed). The rule contains a NAC with the above forbidding pattern (in particular, one link between `g` and `c2`), then, the forbidden pattern appears and the applicability condition is not satisfied.

Step 4: Classifying the Graph Transformation rule

Last step of our method classifies the Graph Transformation rule depending on the results obtained in the previous steps. If all PVUs may be discarded and the rule fulfills the applicability

[21]The opposite update of an update is the update with contrary effect in which the same element participates (e.g. the opposite update of "create an object" is "destroy an object" (and vice versa)).

Table 8.9. Forbidden patterns for each update.

Update	*When the applicability condition must hold?*	*Forbidden pattern*
 n creations of objects of class `cl` (where o_1... o_n are instances of `cl`)	Cmax(`cl`)≠*	Having more than Cmax(`cl`)-n+m objects of class `cl` in the LHS of the rule
 n deletions of objects of class `cl` (where o_1... o_n are instances of `cl`)	Cmin(`cl`)>0 and m<Cmin(`cl`)	Having a NAC with at least Cmin(`cl`)-m objects of class `cl` (excluding the objects to be destroyed)
 n creations of links of the association `as` where the object `o` participates	Cmax(`as,r`)≠* (where `r` is the opposite role to the member end `o`)	Having more than Cmax(`as,r`)-n+m links of the association `as` in the LHS of the rule
 A creation of a link of `as'` (where the link between `o` and `p` belongs to the association `as'`)	Exists a constraint of type Subset(`as`,`r`,`as'`,`r'`)	Having a NAC with a link of `as` between the objects `o` and `p`
 n deletions of links of the association `as` where the object `o` participates	Cmin(`as,r`)>0 and m<Cmin(`as,r`) (where `r` is the opposite role to the member end `o`) and the rule does not include the destruction of the object `o`.	Having a NAC with at least Cmin(`as,r`)-m links of the association `as` (excluding the links to be destroyed)
 A deletion of a link of `as'` (where the link between `o` and `p` belongs to the association `as'`)	Exists a constraint of type Subset(`as`,`r`,`as'`,`r'`)	Having a NAC with a link of `as` between the objects `o` and `p`

conditions, it is WE. Otherwise, the rule is classified as non-WE (and, in fact, it will be non-executable). If this is the case, our method returns as feedback the conditions/applicability patterns that should be added/removed to/from the rule in order to become WE.

Example 82 Although the rule `newMachine` satisfies his applicability condition (see Step 3) not all of its PVUs may be discarded (see Step 2). Then, our method classifies this rule as non-WE. In order to repair it, our method returns as feedback the set of updates the rule should include in order to become WE. In this case, two updates should be added in the rule in order to make it WE: (1) a new link of the association `Input` between the machine `m2` and a conveyor should be created in order to avoid violating the constraint Cmin(`Input,in`)=1; and (2) an attribute computation to initialize the attribute `busy` of the machine `m2` should be added to avoid violating the constraint Mand(`busy,Machine`). Note that the participant objects not created by the rule (i.e. the conveyor `x`) must be present in its LHS. Figure 8.22 shows the rule once this feedback has been added.

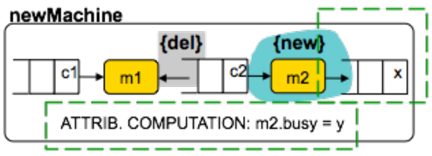


Figure 8.22. Rule `newMachine` correctly repaired.

Example 83 Otherwise, the rule `startMachine` satisfies his applicability condition (see Step 3) all its PVUs may be discarded (see Step 2). Then, our method classifies this rule as WE, meaning that there is at least one host graph in which the rule can be successfully applied.

Example 84 Although the rule `optimize` does not contain PVUs, it does not satisfy its applicability condition. Then, our method classifies this rule as non-WE. In order to repair it, our method returns as feedback the forbidden pattern that should be removed from the LHS of the rule (see Figure 8.23).

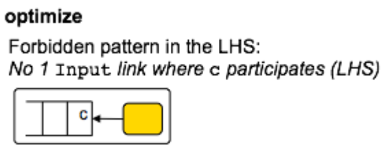

optimize
Forbidden pattern in the LHS:
No 1 Input link where c participates (LHS)
c

Figure 8.23. Feedback for rule `optimize`.

Example 85 Although the rule `disconnectGenerator` does not contain PVUs, it does not satisfy its applicability condition. Then, our method classifies this rule as non-WE. In order to repair it, our method returns as feedback the forbidden pattern that should be removed from the LHS of the rule (see Figure 8.24).

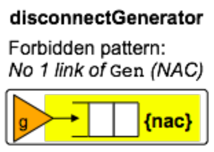


Figure 8.24. Feedback for rule `disconnectGenerator`.

8.4 Summary

Models are neither isolated nor static entities. As part of the MDE process, models may be manipulated using model transformations, which automate the translation of models between a source and a target language using a model transformation language. Model transformations are in many ways similar to traditional software artifacts. Therefore, they need to be verified as well.

With this purpose, in this chapter we have adapted the methods presented in Chapters 6 and 7 to verify model-to-model (M2M) transformations instead of action-based operations. In particular, we adapted the above methods to verify exogenous out-place transformations represented by means of ATL rules [94] and endogenous in-place transformations represented by means of Graph Transformation rules [56]. Although both formalisms are mainly declarative, they can be analyzed regarding the set of individual changes they perform and, therefore, they may be analyzed using our lightweight and static methods.

Note that, an alternative solution to tackle the same problem would be to design an exogenous M2M transformation from ATL/GTr (as a *source* language) to Alf action language (as a *target* language). Then, after applying this transformation to the input M2M transformation model we want to verify, we could directly apply the methods presented in the previous chapters of this thesis. However, in order to be independent of any other language and given that in some cases (see Section 8.3.1) the original method must be extended to cover the particularities of the new language, we preferred to adapt the original method to directly deal with ATL and Graph Transformations.

Regarding the executability property, the methods presented in this chapter are only able to check the weak executability of a M2M transformation. However, these methods could be extended in order to verify the strongly executability of a transformation adapting the ideas presented in Chapter 6. Similarly, the syntactic correctness could also be adapted to the M2M context.

Never worry about theory as long as the machinery does what it's supposed to do.
Robert A. Heinlein

9 Tool Implementation

In order to prove the feasibility of our approach, we have built a prototype tool that implements a subset of the methods proposed along this thesis. In particular, our tool focuses on the verification of the weak and the strong executability of Alf-action-based operations defined in the context of a UML class diagram. We focus on the executability property because we consider this is one of the most important (and also the more complex to be implemented) contributions of this thesis.

Then, the aim of this chapter is to explain how our method for verifying the weak and the strong executability has been internally implemented using specific technologies. It is divided into four sections: Section 9.1 provides an overview of our tool and its architecture; Section 9.2 describes how the *Alf Editor* has been implemented; Section 9.3 describes how the *Alf Verifier* has been implemented; and, finally, Section 9.4 summarizes and concludes the chapter.

9.1 General Overview and Architecture

Our tool is conceived as an Eclipse plug-in. The plug-in itself, their source code, user manuals and other information can be downloaded from [142].

Eclipse is an open source IDE platform launched by Borland, IBM, MERANT, QNX Software Systems, Rational Software, Red Hat, SuSE, TogetherSoft and Webgain in 2001. Since then, the still growing Eclipse community has become rather big. Eclipse is mainly known as an IDE for Java development, but actually Eclipse is much more. Eclipse is intended to serve as a platform for a whole variety of different tools. The "Platform Plug-in Developer Guide"

[53] describes the challenge that Eclipse should take on as: “What we all want is a level of integration that magically blends separately developed tools into a well designed suite. And it should be simple enough that existing tools can be moved to the platform without using a shoehorn or a crowbar. The platform should be open, so that users can select tools from the best source and know that their supplier has a voice in the development of the underlying platform”.

Then Eclipse is essentially an extensible platform that supports a plug-in architecture (see Figure 9.1), i.e. Eclipse offers a common and platform independent user interface that allows developers to create plug-ins supplying tools for different tasks. The platform consists of layers of plug-ins, each layer defining extensions to extension points of lower layers. Plug-ins are components that provide a certain type of service within the context of the Eclipse workbench. Extensions are the central mechanism for contributing behaviour to the platform. Plug-ins can define their own extension points for further customization.

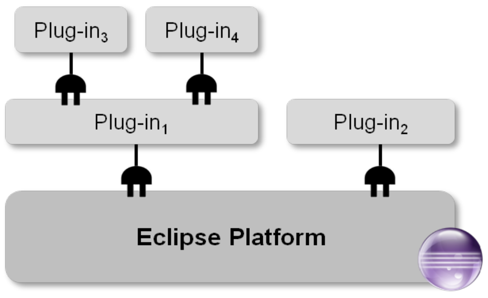


Figure 9.1. Eclipse plug-in architecture.

Moreover, Eclipse also provides a very active online support for developers, mature documentation (see [41]) and a hosting (based on Google code) to publish open source projects that build technology based on the Eclipse platform.

In the recent years, the modelling community has been involved into Eclipse with the Model Development Tools (MDT) project, a set of plug-ins that converts Eclipse into a modelling environment. Among the main functionalities provided by the MDT, there is the *UML2Tools*, a set of GMF (Graphical Modeling Framework) based editors for viewing and editing UML models; the *OCL*, an implementation of the Object Constraint Language (OCL) OMG standard for EMF (Eclipse Modeling Framework) based models; and the *BPMN2*, an open source component to provide a metamodel implementation based on the Business Process Model and Notation (BPMN) 2.0 OMG specification, among others.

The Eclipse support to UML and the wide popularity of this platform in the software development community are the main reasons why we have chosen Eclipse as the basis to develop our prototype tool. Then, our tool (see Figure 9.2) is conceived as an Eclipse plug-in. Our plug-in uses the available UML2Tools plug-in for editing part of the UML executable model we deal with.

Figure 9.3 shows the general view of our tool implementation. As a first step, the designer specifies the UML executable model (s)he wants to deal with. This executable model is

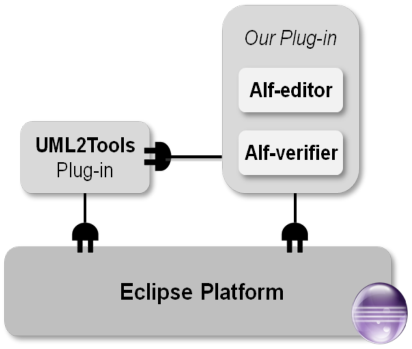


Figure 9.2. General architecture of our plug-in.

composed by:

- A structural model composed by a **UML class diagram** plus a set of **OCL integrity constraints**. The structural model is represented using the graphical modelling environment provided by the UML2Tools Eclipse plug-in [173].
- A behavioural model composed by a set of **Alf-based operations**. The behavioural model is represented using our Alf Editor (see Section 9.2).

Once the executable model is provided, the designer is able to invoke the core of our verification method (see Section 9.3). Finally, the feedback provided by the method is displayed, integrated into the Eclipse interface.

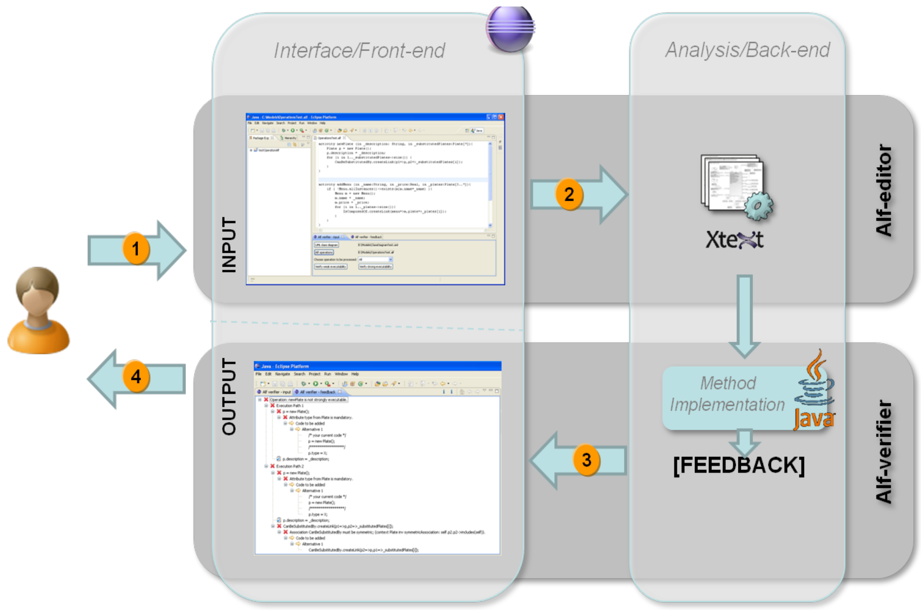


Figure 9.3. Overview of our Eclipse plug-in.

9.2 Alf Editor

In order to verify the input executable model we need to depict the structural model and the behavioural model in an internal representation that can be automatically queried by the verification method.

The structural model (i.e. the UML class diagram and the OCL integrity constraints) is drawn using the UML2Tools (see Figure 9.4), which automatically processes the model and stores it as an instance of the UML metamodel classes.

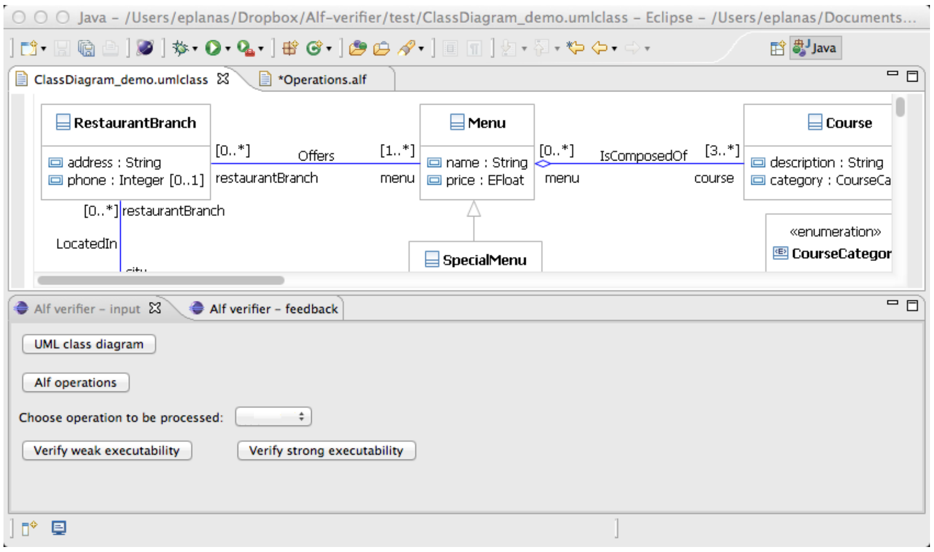


Figure 9.4. Eclipse plug-in screenshot: defining the structural model using UML2Tools.

However, when we started to develop our plug-in, no similar support existed to specify and parse the behavioural model (i.e. the Alf-based operations)[22]. Then, the first part of the plug-in developed as part of this thesis consists in an Alf textual editor that is able to define operations according to the Alf syntax and store them as Java classes.

In order to create the Alf Editor two main components are required: (1) the **Alf front-end**, to define the Alf-based operations into the Eclipse environment; and (2) the **Alf parser**, to ensure the grammatical structure of the input Alf-based operations is consistent with the Alf grammar and to instantiate these operations as Java classes.

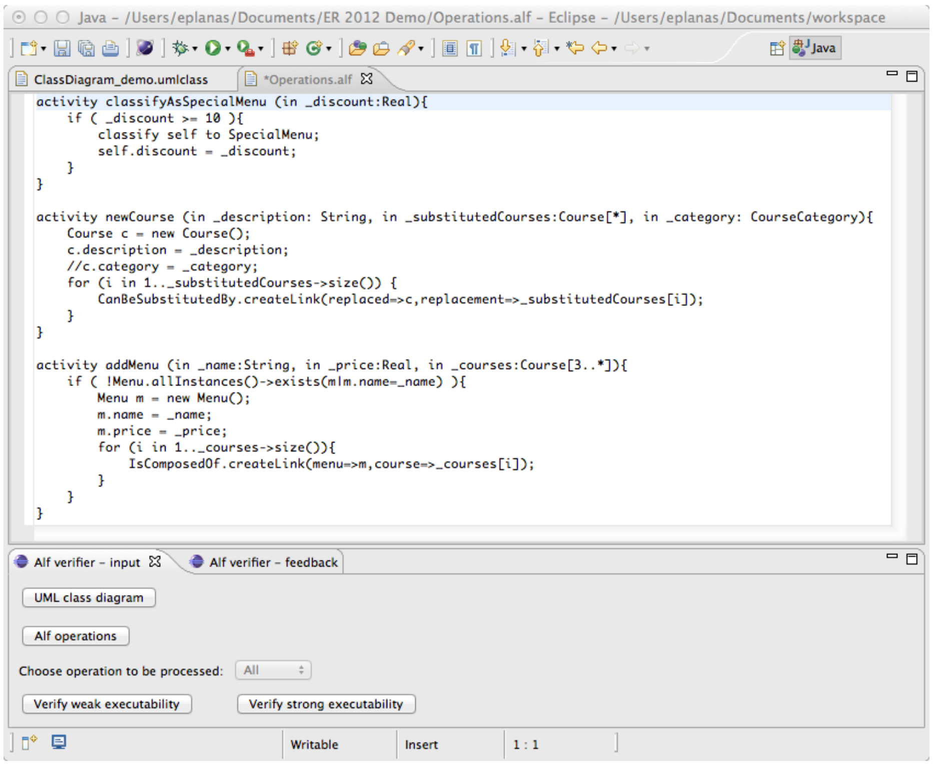
The above components has been created with the help of Xtext [186], an open source framework (as part of the Eclipse Modeling Framework (EMF)) for creating editors for custom languages (both as a Domain-Specific Language (DSL) and a general purpose programming language). Unlike standard parser generators (such as ANTLR or JavaCC), Xtext not only

[22]When we started to develop our Eclipse plug-in (at the beginning of year 2011), no support to specify Alf-based models was available. In fact, several companies and researches were working in that topic, but only one of them (the Papyrus team [130]) had a mature grammar of the Alf language. Nowadays, several initiatives like Papyrus have been emerged. Although our verification method uses our Alf Editor, it could be also integrated with other existing editors of this language.

generates a parser, but also a metamodel and a fully featured, customizable Eclipse-based IDE.

To create the above components, Xtext departs from a grammar of the language we want to deal with (in our case, the Alf grammar). With this purpose, we use the Alf grammar provided by the Papyrus team, which is consistent with the Alf standard specification [124]. The Alf grammar is composed by a set of formation rules that describe how to form valid operations according to the Alf syntax.

As a result, we obtain an Alf Editor which constitutes a new Eclipse tab in the main view (see Figure 9.5). Using this tab, the designer is able to define a set of Alf-based operations which are stored in a file with extension *.alf*. On the other hand, we have developed an engine that analyzes each operation and instantiates it as a set of Java classes that will be later used by the Alf Verifier (see Section 9.3).


Java – /Users/eplanas/Documents/ER 2012 Demo/Operations.alf – Eclipse – /Users/eplanas/Documents/workspace
Java
ClassDiagram_demo.umlclass
*Operations.alf

```
activity classifyAsSpecialMenu (in _discount:Real){
    if ( _discount >= 10 ){
        classify self to SpecialMenu;
        self.discount = _discount;
    }
}

activity newCourse (in _description: String, in _substitutedCourses:Course[*], in _category: CourseCategory){
    Course c = new Course();
    c.description = _description;
    //c.category = _category;
    for (i in 1.._substitutedCourses->size()) {
        CanBeSubstitutedBy.createLink(replaced=>c,replacement=>_substitutedCourses[i]);
    }
}

activity addMenu (in _name:String, in _price:Real, in _courses:Course[3..*]){
    if ( !Menu.allInstances()->exists(m|m.name=_name) ){
        Menu m = new Menu();
        m.name = _name;
        m.price = _price;
        for (i in 1.._courses->size()){
            IsComposedOf.createLink(menu=>m,course=>_courses[i]);
        }
    }
}
```

Choose operation to be processed: All

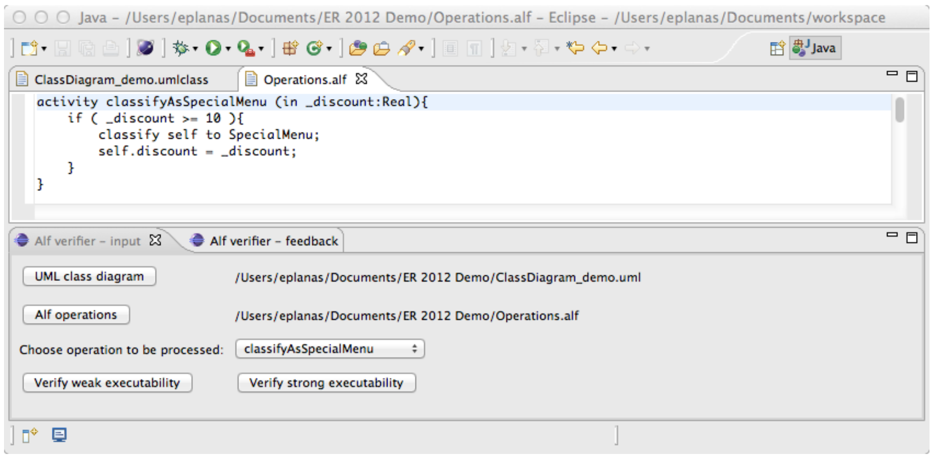
Figure 9.5. Eclipse plug-in screenshot: defining the behavioural model in the Alf Editor tab.

9.3 Alf Verifier

The Alf Verifier implements the lightweight static method presented in Chapter 6. In the next subsections we explain the input and the output of the verifier as well as the internal implementation of our method. We use the operations discussed in Chapter 6 to illustrate how the designer can verify an executable model using our tool.

9.3.1 Input

The Alf Verifier takes as input the structural and the behavioural models selected by the user through the Eclipse interface (see Figure 9.6). The user may also select what operations (s)he wants to verify (one specific or all) and the property (s)he wants to check (weak or strong executability).



Java – /Users/eplanas/Documents/ER 2012 Demo/Operations.alf – Eclipse – /Users/eplanas/Documents/workspace

```
activity classifyAsSpecialMenu (in _discount:Real){
    if ( _discount >= 10 ){
        classify self to SpecialMenu;
        self.discount = _discount;
    }
}
```

UML class diagram: /Users/eplanas/Documents/ER 2012 Demo/ClassDiagram_demo.uml

Alf operations: /Users/eplanas/Documents/ER 2012 Demo/Operations.alf

Choose operation to be processed: classifyAsSpecialMenu

Figure 9.6. Eclipse plug-in screenshot: selecting the input executable model to be verified.

9.3.2 Lightweight Static Analysis

When the user clicks the button *Verify Weak Executability* or *Verify Strong Executability* the core of our method is invoked. It is implemented as a set of Java classes that carry out all the steps our method performs. For each operation to be verified, it performs the following steps:

Step 0: Computing Execution Paths

In order to compute the execution paths, our tool represents each operation as a directed graph, where each vertex represents an action of the operation and each arc represents the precedence between actions. Once the graph is constructed, it is parsed to extract all execution paths as we explained in Chapter 6. Each path is internally represented as an *array* of *terms* (representing each action in the path) and *guards* (representing each boolean expression over the path).

Once the paths have been computed, steps 1 and 2 are applied on each execution path until a WE path is recognized (in case of verifying the weak executability) or until ensuring all paths are SE (in case of verifying the storng executability).

Step 1: Analyzing the existence of Potentially Violating Actions (PVAs)

This step is split in two tasks. First, the tool determines the actions that could violate each integrity constraint of the structural model. In order to do it, the tool queries the input structural model and applies the rules shown in the Step 1 of Chapter 6.

Second, the above set of actions is compared with the set of actions included in the terms of the execution path that is being analyzed. All actions in the intersection of both sets are considered PVAs. The PVAs are stored in a *hash table*, where each PVA acts as a *key* and their associated elements are the set of constraints (SVCs) it may violate.

Step 2: Discarding PVAs

This step implements the tables shown in the Step 2 of Chapter 6 in order to try to discard each PVA returned by the previous step. The discarded PVAs are marked as *discarded* while the rest of the PVAs are enriched with the conditions that should be satisfied in order to avoid the violations. These conditions will be returned as feedback.

Step 3: Classifying the operation

The last step classifies each operation depending on the results obtained in the previous step. If at least one execution path of the operation is WE, the operation is classified as WE. If all its execution paths are SE, the operation is classified as SE. Otherwise, the operation is classified as non-executable. If that is the case, the method returns a meaningful feedback (see Section 9.3.3) to help the user to repair the operations.

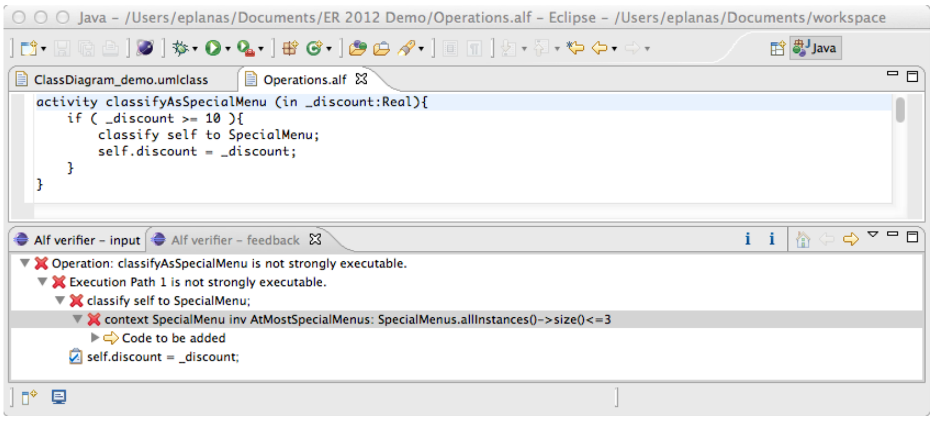
9.3.3 Feedback

As we explained in Chapter 6 our method returns a feedback that (in case of the verified operation is non-executable) it points out *why* (i.e. which actions in the operation may violate some integrity constraints of the structural model) and *how* the designer can fix these errors (providing a set of possible repairing alternatives that should be included in the operation). This feedback is expressed in terms of the Alf language and it is displayed into the Eclipse interface (see Figure 9.7)).

As an example, consider the operation `classifyAsSpecialMenu` introduced in Chapter 6, which classifies a menu as special menu.

```
activity classifyAsSpecialMenu(in _discount:Real) {
  if ( _discount ≥ 10 ) {
   classify self to SpecialMenu;
   self.discount = _discount;
  }
}
```

As we have seen in Chapter 6, this operation is not strongly executable since as the feedback informs (see Figure 9.7) it violates the constraint Cmax(`SpecialMenu`) = 3 of the structural model.

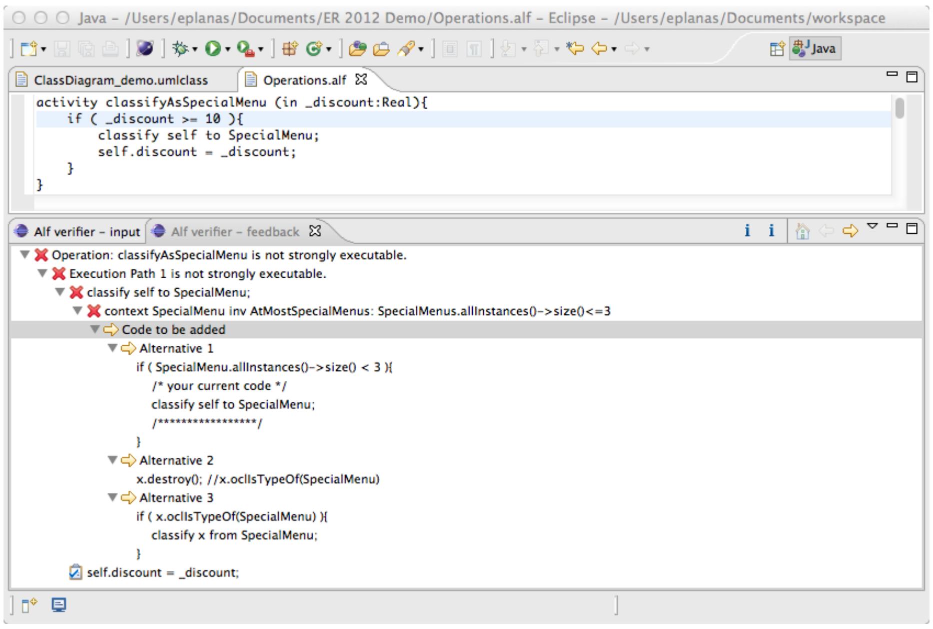

Java – /Users/eplanas/Documents/ER 2012 Demo/Operations.alf – Eclipse – /Users/eplanas/Documents/workspace
Java
ClassDiagram_demo.umlclass
Operations.alf
Alf verifier – input
Alf verifier – feedback

```
activity classifyAsSpecialMenu (in _discount:Real){
    if ( _discount >= 10 ){
        classify self to SpecialMenu;
        self.discount = _discount;
    }
}
```

```
▼ ✗ Operation: classifyAsSpecialMenu is not strongly executable.
   ▼ ✗ Execution Path 1 is not strongly executable.
      ▼ ✗ classify self to SpecialMenu;
         ▼ ✗ context SpecialMenu inv AtMostSpecialMenus: SpecialMenus.allInstances()->size()<=3
            ▶ ⇨ Code to be added
      ✓ self.discount = _discount;
```

Figure 9.7. The feedback shows *why* the operation `classifyAsSpecialmenu` is not SE.

Besides, the feedback suggests (see Figure 9.8) several repairing alternatives to avoid this violation. Note that some of these alternatives include the PVA (inside a comment), meaning that the code to be added must appear before/after the PVA. If the PVA is not included, it means that place where the suggested coded must be added does not matter.

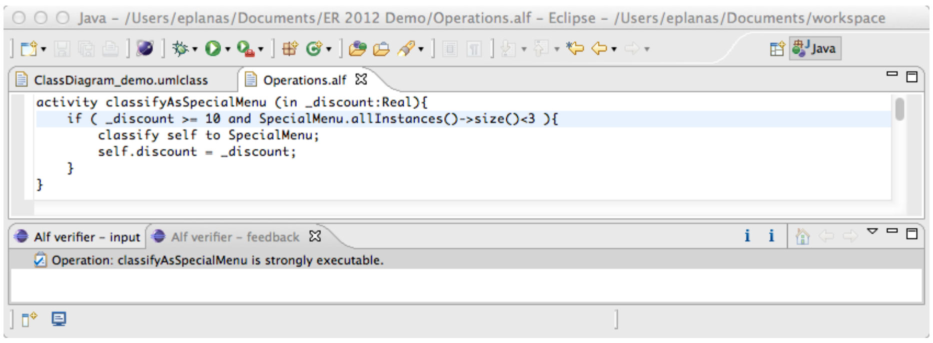

Java – /Users/eplanas/Documents/ER 2012 Demo/Operations.alf – Eclipse – /Users/eplanas/Documents/workspace
Java
ClassDiagram_demo.umlclass
Operations.alf
Alf verifier – input
Alf verifier – feedback

```
activity classifyAsSpecialMenu (in _discount:Real){
    if ( _discount >= 10 ){
        classify self to SpecialMenu;
        self.discount = _discount;
    }
}
```

```
▼ ✗ Operation: classifyAsSpecialMenu is not strongly executable.
   ▼ ✗ Execution Path 1 is not strongly executable.
      ▼ ✗ classify self to SpecialMenu;
         ▼ ✗ context SpecialMenu inv AtMostSpecialMenus: SpecialMenus.allInstances()->size()<=3
            ▼ ⇨ Code to be added
               ▼ ⇨ Alternative 1
                  if ( SpecialMenu.allInstances()->size() < 3 ){
                     /* your current code */
                     classify self to SpecialMenu;
                     /****************/
                  }
               ▼ ⇨ Alternative 2
                  x.destroy(); //x.oclIsTypeOf(SpecialMenu)
               ▼ ⇨ Alternative 3
                  if ( x.oclIsTypeOf(SpecialMenu) ){
                     classify x from SpecialMenu;
                  }
      ✓ self.discount = _discount;
```

Figure 9.8. The feedback suggests *how* the operation `classifyAsSpecialmenu` may be fixed.

When the designer adds one of the suggested repairing alternatives to the operation, it becomes strongly executable (see Figure 9.9). But it may be the case that the added code generates new PVAs making the operation non-executable again. If that is the case, the feedback should be added iteratively until we obtain a correct operation.


Java – /Users/eplanas/Documents/ER 2012 Demo/Operations.alf – Eclipse – /Users/eplanas/Documents/workspace
Java
ClassDiagram_demo.umlclass
Operations.alf
Alf verifier – input
Alf verifier – feedback

```
activity classifyAsSpecialMenu (in _discount:Real){
    if ( _discount >= 10 and SpecialMenu.allInstances()->size()<3 ){
        classify self to SpecialMenu;
        self.discount = _discount;
    }
}
```

Operation: classifyAsSpecialMenu is strongly executable.

Figure 9.9. Operation `classifyAsSpecialmenu` correcly repaired.

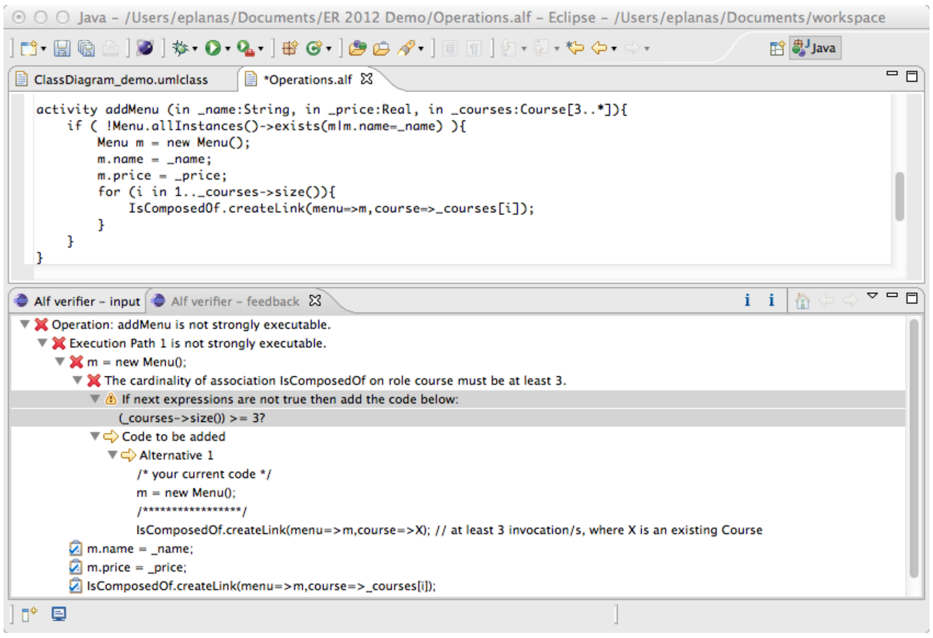
As we commented in Chapter 6 our method may require the user intervention to disambiguate some situations that cannot be computed by our lightweight method. As an example, when verifying the operation `addMenu` the user intervention is required to determine if the inequality `_courses->size() >= 3` is true (see Figure 9.10). In this case, since the parameter `_courses` contains at least three courses, the above inequality is always true, making the operation strongly executable. Otherwise, in case that the inequality was not true, as the feedback points out (see Figure 9.10), the designer should add the suggested code, in this case, ensuring that the menu `m` is linked with at least three courses.

```
activity addMenu(in _name:String, in _price:Real, in _courses:Course[3..*])
{
  if ( !Menu.allInstances()→exists(m|m.name=_name) ) {
   Menu m = new Menu();
   m.name = _name;
   m.price = _price;
   for ( i in 1.._courses→size() ) {
    IsComposedOf.createLink(menu=>m,course=>_courses[i]);
   }
  }
}
```

9.4 Summary

In this chapter we have explained the fundamentals of the prototype tool we developed to prove the feasibility of our approach. We implemented this prototype as a plug-in integrated into the Eclipse platform, one of the most popular development platforms in the software development community.

Our plug-in focuses on the implementation of the method presented in Chapter 6 to verify the weak/strong executability of Alf-based operations, since it is one of the most important contributions of this thesis. However, the rest of the properties presented in this thesis (syntactic


Java – /Users/eplanas/Documents/ER 2012 Demo/Operations.alf – Eclipse – /Users/eplanas/Documents/workspace
Java
ClassDiagram_demo.umlclass
*Operations.alf
Alf verifier – input
Alf verifier – feedback

```
activity addMenu (in _name:String, in _price:Real, in _courses:Course[3..*]){
    if ( !Menu.allInstances()->exists(m|m.name=_name) ){
        Menu m = new Menu();
        m.name = _name;
        m.price = _price;
        for (i in 1.._courses->size()){
            IsComposedOf.createLink(menu=>m,course=>_courses[i]);
        }
    }
}
```

```
Operation: addMenu is not strongly executable.
  Execution Path 1 is not strongly executable.
    m = new Menu();
      The cardinality of association IsComposedOf on role course must be at least 3.
        If next expressions are not true then add the code below:
          (_courses->size()) >= 3?
        Code to be added
          Alternative 1
            /* your current code */
            m = new Menu();
            /*****************/
            IsComposedOf.createLink(menu=>m,course=>X); // at least 3 invocation/s, where X is an existing Course
    m.name = _name;
    m.price = _price;
    IsComposedOf.createLink(menu=>m,course=>_courses[i]);
```

Figure 9.10. The tool requires the user intervention when verifying the operation `addMenu`.

correctness and completeness) could also be integrated in our prototype.

On the other hand, this implementation could also be adapted and integrated into a tool for editing ATL rules (in order to verify the weak/strong executability of an ATL rule) or into a tool for editing Graph Transformation rules (in order to verify the weak/strong executability of a Graph Transformation rule).

Supposing is good, but finding out is better.

Mark Twain

10 Experimentation

According to the Design-Science Research (DSR) paradigm proposed by Hevner et al. [83], the evaluation of the designed artifacts produced as a result of the research process is crucial.

The evaluation of the designed artifacts may rely on several methodologies available in the knowledge base such as *observation* (case studies, field studies, ...), *analysis* (static analysis, architecture analysis, optimization, ...), *experimentation* (controlled experiments, simulation, ...), *testing* (functional black box, structural white box, ...), etc.

Despite widespread interest in evaluating the produced artifacts, in the software engineering field there is still a little guidance on which methodologies are suitable to which research problems, and how to choose amongst them. In [163], Easterbrook et al. provide a basis for both understanding and selecting from the variety of methods applicable to empirical software engineering. Following the criteria suggested by these authors, we selected *experimentation* as the method to evaluate several features of the result of our research (i.e. our verification framework and our prototype tool). An experiment is an investigation of a testable hypothesis where one or more independent variables are manipulated to measure their effect on one or more dependent variables. This methodology has been largely used in software engineering [13, 95, 170].

This chapter is structured in two sections. Section 10.1 describes our first experiment to evaluate the *relevance* of our verification framework and Section 10.2 describes our second experiment to evaluate the *efficiency* of our prototype tool.

10.1 Experiment 1: Evaluating the relevance of our verification framework

The purpose of the first experiment was to evaluate the *relevance* of the verification framework proposed in this thesis. In order to justify the need for verification frameworks, we show that writing correct executable operations by hand and without any kind of support (such as verification or code generation tools) is not easy.

Our first hypothesis can be formulated as: “writing executable operations by hand is error prone”.

In the next subsections we explain how this experiment was designed (see Section 10.1.1) as well as its results and conclusions (see Section 10.1.2).

10.1.1 Design of the experiment

Participants

A total of one hundred people (14% female and 86% male) participated in our experiment. All the participants were grad students of a Software Engineering course of the Computing Engineering Bachelor taught at the Open University of Catalonia (UOC). All the students had a good background about modelling in UML and good programming skills (using object oriented languages such as Java or C++). Besides, most of the students had a job related to his studies.

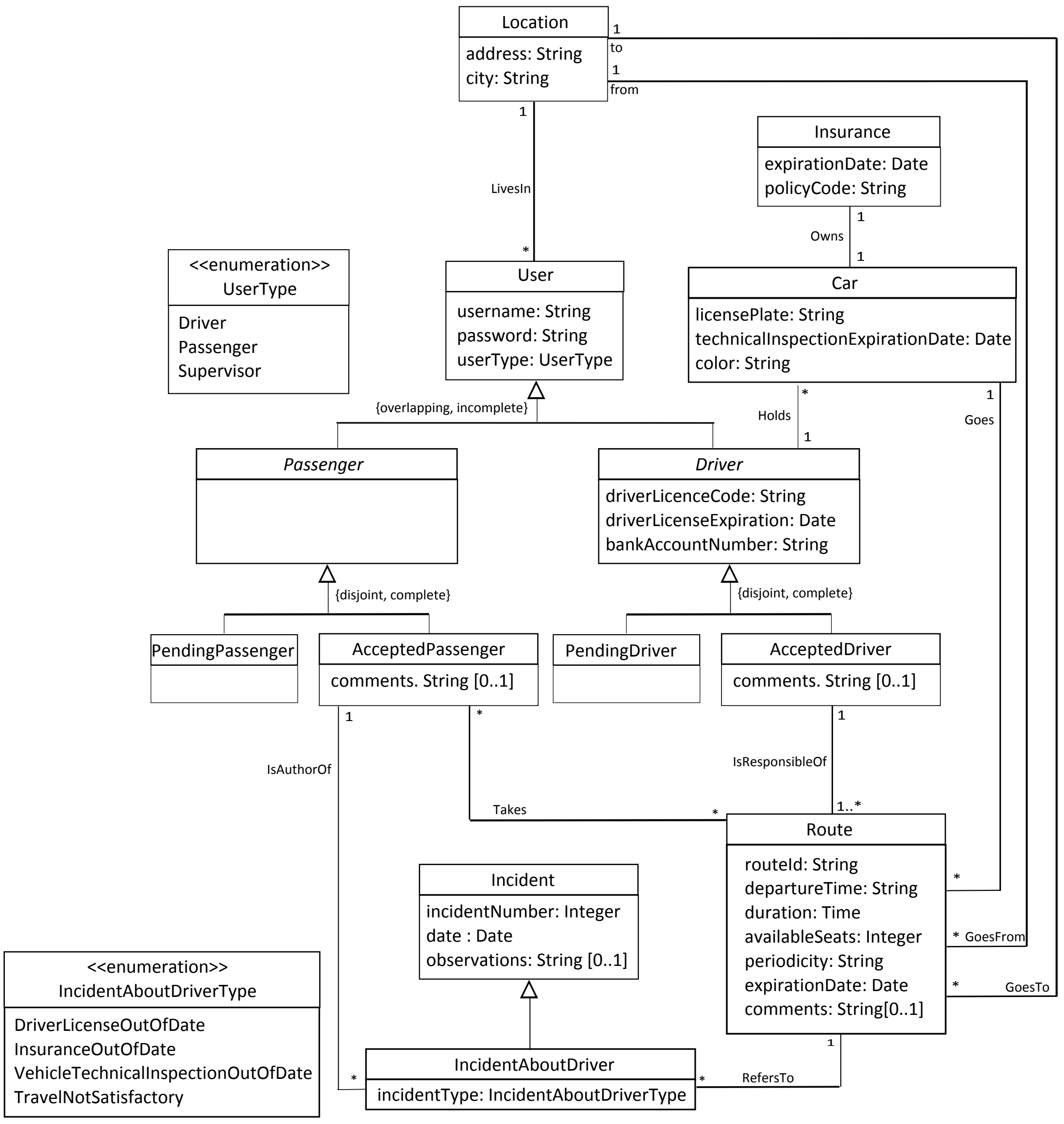
Tasks

The participants were asked to carry out two tasks. Both tasks were part of a mandatory activity (which also contained other tasks) that all students had to deliver to pass the course.

As a starting point, the students were provided with a UML class diagram (see Figure 10.1) representing a car sharing system. This class diagram contains information about the *drivers* and the *passengers* registered in the system (both of them must be accepted by the *supervisors* of the system before being able to use the whole functionalities) and the *cars* (together with its *insurance*) that each driver holds. Each driver is responsible of at least one *route*. Routes may have associated *incidences*, that store information about unpleasant trips. Although the original class diagram contained more elements and integrity constraints, here we show a simplified version for the purpose of Task 1.

Based on the above class diagram, two tasks were formulated:

Task 1: Designing correct operations. The first task that was asked to the participants was about designing a set of action-based operations. In the definition of the task we put special



context User **inv** : User.allInstances()->isUnique(username)
context Location **inv** : Location.allInstances()->isUnique(address)
context Insurance **inv** : Insurance.allInstances()->isUnique(policyCode)
context Car **inv** : Car.allInstances()->isUnique(licencePlate)
context Route **inv** : Route.allInstances()->isUnique(routeId)
context Incident **inv** : Incident.allInstances()->isUnique(incidentNumber)
context IncidentAboutDriver **inv**: self.acceptedPassenger.route ->includes(self.route)
context Route **inv**: self.acceptedDriver.car->includes(self.car)

ID(username,User)
ID(address,Location)
ID(policyCode,Insurance)
ID(licencePlate,Car)
ID(routeId,Route)
ID(incidentNumber,Incident)

Figure 10.1. Starting point for Task 1.

emphasis in the *correctness* of the operations, saying that "a correct operation is invoked in a consistent scenario and leaves the system in a new scenario also consistent with the class diagram and all its integrity constraints" (i.e. the operations should be *executable*). However, we do not explicitly said that the operations should be syntactically correct, since we consider this is an intrinsic property that any designer/programmer knows.

Prior to develop the task, the participants were introduced to action-based operations and Alf action language. As an example, we provided the participants with the `createSupervisor` operation (in the context of class `User`), which creates a new supervisor in the system:

```
activity createSupervisor(in _username: String, in _password: String, in
_location: Location) {
  if ( !User.allInstances()->exists(u|u.username=_username) ) {
   User u = new User();
   u.username = _username;
   u.password = _password;
   u.userType = ``Supervisor'';
   LivesIn.createLink(user=>u,location=>_location);
  }
}
```

We suggested the use of the Alf action language, but the participants could choose to use another action language or any imperative language based on pseudocode.

As part of this task, the operations to be designed were the following:

- Operation `createIncidenceAboutDriver` (in the context of class `Passenger`): When a passenger feels a travel has not been satisfactory, using this operation (s)he is able to create a new incidence about the driver who is responsible of the route. The type of the new incidence must be `TravelNotSatisfactory`.

  One acceptable solution to accomplish this requirement is:

```
activity createIncidenceAboutDriver(in _incidentNumber: Integer, in
_date: Date, in _route: Route) {
  //Create the incidence and initialize its mandatory attributes
  IncidentAboutDriver i = new IncidentAboutDriver();
  i.incidentNumber = _incidentNumber;
  i.date = _date;
  i.incidentType = ``TravelNotSatisfactory'';
  //Link the new incidence to a Route and a Passenger
  RefersTo.createLink(incident=>i,route=>_route);
  IsAuthorOf.createLink(passenger=>self,incident=>i);
}
```

- Operation `replaceCar` (in the context of class `Car`): When a driver decides to eliminate one of its cars, using this operation s(he) is able to remove this car of the system and replace all the routes assigned to the removed car to another existing car belonging to the same driver.

  One acceptable solution to accomplish this requirement is:

```
activity replaceCar(in _newCar: Car) {
  //Assign all the routes of the deleted car to the new car
  Routes[*] routesList = self.route;
  for int i=1 to routesList->size() {
   Goes.destroyLink(car=>self,route=>routesList[i]);
   Goes.createLink(car=>_newCar,route=>routesList[i]);
  }
  //Destroy the object (car) itself
  self.destroy();
}
```

Remember that, in Alf, the links in which a destroyed object participates and the objects owned by this object are destroyed along with the object itself. Then, in this case, the insurance of the destroyed car and the link between this car and its insurance is destroyed during the destruction of the car.

- Operation `admitDriver` (in the context of class `PendingDriver`): Using this operation, a supervisor can accept a driver (i.e. classify it from class `PendringDriver` to `AcceptedDriver`). Note that, in order to satisfy the class diagram multiplicities, the accepted driver must be responsible of at least one route.

  One acceptable solution to accomplish this requirement is:

```
activity admitDriver(in _routeId: String, in _departureTime: String,
in _duration: Time, in _availableSeats: Integer, in _expirationDate:
Date, in _from: Location, in _to: Location, in _car: Car) {
  classify self from PendingDriver to AcceptedDriver;
  Route r = new Route();
  r.routeId = _routeId;
  r.departureTime = _departureTime;
  r.duration = _availableSeats;
  r.periodicity = _periodicity;
  r.expiration = _expiration;
  GoesFrom.createLink(route=>r,from=>_from);
  GoesTo.createLink(route=>r,to=>_to);
  IsResponsibleOf.createLink(driver=>self,route=>_route);
  Goes.createLink(car=>_car,route=>r);
}
```

  Note that in this solution a new route is created to be linked with the accepted driver.

Task 2: Designing a complete operation's set. The second task that was asked to the participants was about providing the signature (context, name and parameters) of a set of operations. In the definition of the task we put special emphasis in the *completeness* of the operations set, saying that "the set of operations should allow to create/delete all the elements of the class diagram and its relationships as well as to modify the value of its attributes". In order to prevent the task was too tedious, we delimited the class diagram in which this task was focused on (see Figure 10.2).

Then, the expected solution was the following:

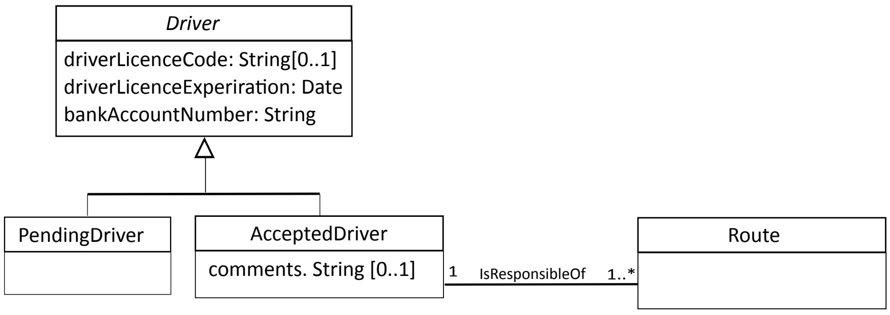

Driver
driverLicenceCode: String[0..1]
driverLicenceExperiration: Date
bankAccountNumber: String
PendingDriver
AcceptedDriver
comments. String [0..1]
1
IsResponsibleOf
1..*
Route

Figure 10.2. Starting point for Task 2.

```
(context PendingDriver) activity createPendingDriver(in _driverLicenseCode:
String, in _driverLicenseExpiration: Date, in _bankAccountNumber:
String): PendingDriver
(context AcceptedDriver) activity createAcceptedDriver(in
_driverLicenseCode: String, in _driverLicenseExpiration: Date, in
_bankAccountNumber: String, in _comments: String[0..1]): AcceptedDriver
(context Route) activity createRoute(in _responsible: AcceptedDriver):
Route
(context Driver) activity setDriverLicenseCode(in _newLicenseCode: String)
(context Driver) activity setDriverLicenseExpiration(in
_newLicenseExpiration: String)
(context Driver) activity setBankAccountNumber(in _newBankAccountNumber:
String)
(context AcceptedDriver) activity setComments(in _newComments: String)
(context PendingDriver) activity destroyPendingDriver()
(context AcceptedDriver) activity destroyAcceptedDriver()
(context Route) activity destroyRoute()
(context PendingDriver) activity admitDriver()
(context PendingDriver) activity rejectDriver()
(context Driver) activity relateDriverWithRoute(in _route: Route)
(context Driver) activity unrelateDriverWithRoute(in _route: Route)
```

Experiment environment

Given the characteristics of the course, the students worked together in mixed groups of five people. Then, we had twenty groups of five students each. As a result, we obtained twenty solutions to be analyzed.

The resolution of the complete activity (composed by these two tasks and nine more tasks to evaluate the contents of the course) lasted for three weeks. During this time period, the students were able to work collaboratively, ask questions to the teacher and search any kind of information to help resolving the proposed tasks.

10.1.2 Results of the experiment

As a result of our first experiment, we obtained twenty solved activities to be analyzed (one for each group of students). From Task 1, we obtained sixty operations (three operations for each group) to be analyzed individually. From Task 2, we obtained twenty sets of operations (one set for each group) to be analyzed together.

Our analysis was focused on checking whether the operations obtained from Task 1 were *syntactically correct* and *executable*. On the other hand, it was also focused on checking whether the sets of operations obtained from Task 2 were *complete*. Since not all operations were specified using the Alf action language, the operations were manually examined.

Syntactic correctness

First of all, we analyzed whether the sixty operations obtained from Task 1 were syntactically correct. Surprisingly, only 45% of the examined operations were syntactically correct (see Figure 10.5). It means that a significant number of them contained syntactical errors. We classified these errors into several categories (see Figure 10.3):

- *Structural errors.* In this category we classified those errors that were related with the structure of the language. For instance, to miss the `return` sentence in an operation which is not void, to use variables that have not been defined before, to miss semicolons, to write too many parentheses or wrong use of the language constructors (i.e. `if` sentences, `for each` sentences, and so on). These errors represent a 37% of the syntax errors analyzed in our experiment.

- *Typographical errors.* In this category we classified those mistakes made during the typing process. These errors represent a 21% of the syntax errors analyzed in our experiment.

- *Wrong/missing arguments in actions.* In this category we classified those errors that were directly related with the syntax of the actions of the action language used to specify the operations. For instance, to miss parameters in an action (e.g., to use `IsResponsibleOf.createLink(route=>r)` instead of `IsResponsibleOf.createLink(driver=>self,route=>r)`), to use inconsistent parameters in an action (e.g., to use `IsResponsibleOf.createLink(driver=>self,car=>c)`) or to exchange the order of the parameters inside an action (for instance, to use `classify self from AcceptedDriver to PendingDriver` in order to accept a pending driver), among others. These errors represent a 42% of the syntax errors analyzed in our experiment.

Given that syntactic correctness is a pre-requisite for the rest of the correctness properties we address in this thesis, we manually corrected the non-syntactically correct operations before verifying its executability.

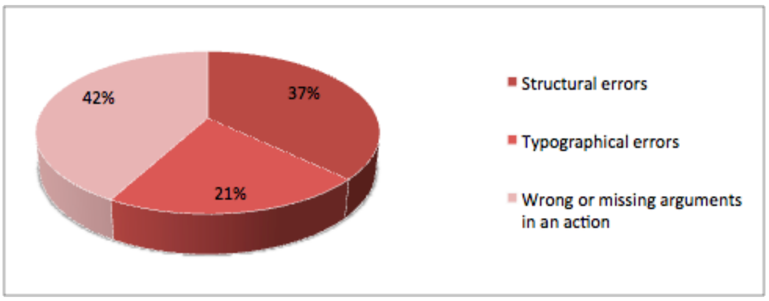


Figure 10.3. Classification of the syntactic errors found in Task 1.

Executability

Second, we analyzed whether the sixty operations obtained from Task 1 were executable (classifying them into weakly and strongly executable) or not (classifying them as non-executable). We concluded that 68% of the operations were non-executable and only 32% were executable (in particular, 32% where weakly executable while 26% of them were also strongly executable[23]) (see Figure 10.4). It means that, although the statement of the Task 1 puts special emphasis on the correctness of the designed operations, most of the operations designed by the students left the system in a state which was inconsistent with the class diagram and/or some of its its integrity constraints.

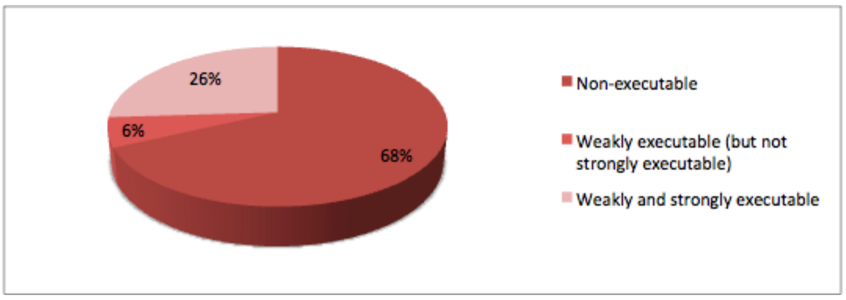


Figure 10.4. Executability of the analyzed operations.

Completeness

Finally, we analyzed whether the twenty sets of operations obtained from Task 2 were complete. We concluded that 90% of the analyzed sets were incomplete (i.e. only 10% of them were complete) (see Figure 10.5). It means that, although the statement of the Task 2 puts special emphasis on the completeness of the set of designed operations, almost all sets of operations did not allow to create/delete all the elements of the class diagram and its relationships as well as to modify the value of its attributes.

[23]Note that, since weak executability is a pre-requisite for strong executability, all strongly executable operations are also weakly executable.

Conclusions

To sum up, as can be seen in Figure 10.5, 45% of the operations obtained from Task 1 of our experiment were syntactically correct (then, 55% of them were not syntactically correct), 32% were weakly (but not strongly) executable and 26% were weakly and strongly executable (then, 68% were non-executable). On the other hand, only 10% of the sets of operations obtained from Task 2 were complete (then, 90% were incomplete).

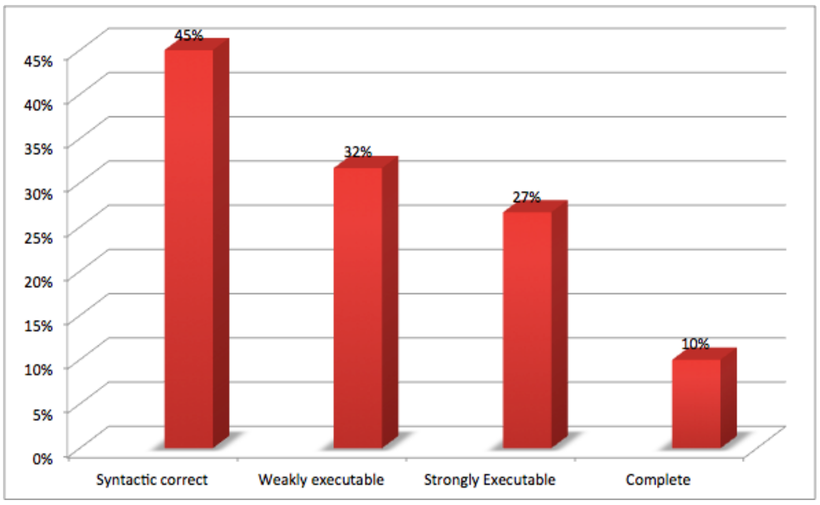


Figure 10.5. Correctness of the analyzed operations.

As a conclusion, this experiment evidences that designers are not able to easily write correct executable operations by themselves. We believe this conclusion clearly supports the need for methods able to evaluate the quality of the operations. Besides, to be useful, those methods should be easily integrated in the modelling tools used by practitioners.

With this purpose, all the methods developed as part of this thesis try to help the designers during the design of the behaviour that composes an executable model and can be helpful to detect and correct the problems observed during this experiment.

10.2 Experiment 2: Evaluating the efficiency of the lightweight and static methods

Traditional formal verification methods (as model checking) usually suffer from the so-called state explosion problem, meaning that the size of the underlying state space grows exponentially in terms of the size of the model to be verified. This trade-off compromises the efficiency of these methods. As a consequence, they tend to be used to verify a limited subset of systems (mainly critical systems). In contrast, lightweight and static methods are generally more efficient and, thus, they are suitable to be integrated into the development tools to be used during the development process.

In order to assesses whether lightweight static methods are efficient (considering efficiency as *the ability to accomplish a purpose with a reasonable time*), the purpose of the second experiment was to evaluate the *efficiency* of our prototype tool for verifying the executability of Alf-based operations (see Chapter 9).

Our second hypothesis can be formulated as: "our lightweight static tool to verify the correctness of Alf-based operations is efficient, i.e. it is able to perform the verification in a reasonable time".

10.2.1 Design of the experiment

Sample

Most of the input models used in this second experiment were small UML/Alf executable models. In particular, this experiment took as input two UML/Alf executable models that has been introduced along this thesis: (1) the restaurant chain system, introduced in Chapter 2; and (2) the car sharing system, introduced in the Section 10.1 of this chapter. In order to measure the efficiency of our tool, we execute it using several input models and computing its execution time.

In order to guarantee that our tool is also efficient when using large models, we also used a large model artificially created for this purpose.

Variables

As we said, the objective of the experiment was to evaluate the efficiency of our prototype tool. In order to evaluate this feature, we measured the time consumed during the verification process. In particular, we focus on the time consumed when verifying whether the operations were strongly executable since, in general, it is worse than the time consumed when verifying the weak executability.

During the experiment we take into account the following variables:

1. *Size of the input structural model (class diagram).* The number of classes, attributes, associations, generalizations and integrity constraints (considering both the graphical constraints and the textual ones) of the input class diagram.
2. *Size of the input operation to be verified.* The number of actions, conditionals and loops of the input operation.
3. *Running time.* The time (in milliseconds) that takes the verification. This running time contemplates the whole process, i.e. parsing the structural and behavioural models, com-

puting the execution paths of the operation to be verified, computing its PVAs, discarding the PVAs, classifying the operation and returning the corresponding feedback.

Experiment environment

The experiment was run on a Windows XP laptop, with an Intel Pentium processor, 1.73 GHz, and 1.0 Gb RAM.

10.2.2 Results of the experiment

Table 10.1 details the performance results for the experiments we conducted with our Eclipse plug-in. We verified eight Alf-based operations using our prototype. For those operations that were not executable, we verify them again until reached an executable operation. Then, "`op` (1)" makes reference to the first time the operation `op` was verified, while "`op` (2)" makes reference to the second time.

A brief description of each variable may be found in Section 10.2.1.

Conclusions

As can be seen in Table 10.2.1, the median running time is 3704 milliseconds (i.e. 3.7 seconds). All the executions using small executable models (see rows 1 to 9 of Table 10.2.1) take less than 4 seconds, whilst the large model (see row 10 of Table 10.2.1) takes almost 5 seconds. However, in both cases we consider this is a reasonable execution time (compared to those verification methods that simulate the behaviour and may take several minutes).

Although we would like to evaluate the performance of our tool using big models from a real environment, we believe these preliminary results points out the efficiency of our lightweight static tool. Besides, even that we have only used an artificial large model, our experiment shows the scalability of our method.

As a conclusion, we believe the proposed method is efficient, and then, it is suitable to be integrated in the development process tools used by the designers.

Table 10.1. Performance results for our prototype tool.

	Operation (iteration)	*Class diagram size*	*Operation size*	*Running time*
1	`newCourse` (1)	6 classes, 8 attributes, 4 associations, 1 generalization, 13 constraints	3 actions, 1 loop	3641 ms
2	`newCourse` (2)	6 classes, 8 attributes, 4 associations, 1 generalization, 13 constraints	3 actions, 1 loop	3593 ms
3	`addMenu` (1)	6 classes, 8 attributes, 4 associations, 1 generalization, 13 constraints	4 actions, 1 conditional, 1 loop	3579 ms
4	`classifyAs-SpecialMenu` (1)	6 classes, 8 attributes, 4 associations, 1 generalization, 13 constraints	2 actions, 1 conditional	3522 ms
5	`classifyAs-SpecialMenu` (2)	6 classes, 8 attributes, 4 associations, 1 generalization, 13 constraints	2 actions, 1 conditional	3563 ms
6	`createSupervisor` (1)	13 classes, 28 attributes, 7 associations, 3 generalizations, 46 constraints	5 actions	3546 ms
7	`createIncidence-AboutDriver` (1)	13 classes, 28 attributes, 7 associations, 3 generalizations, 46 constraints	6 actions	3516 ms
8	`replaceCar` (1)	13 classes, 28 attributes, 7 associations, 3 generalizations, 46 constraints	4 actions, 1 loop	3531 ms
9	`admitDriver` (1)	13 classes, 28 attributes, 7 associations, 3 generalizations, 46 constraints	11 actions	3578 ms
10	`scalabilityTest` (1)	100 classes, 200 attributes, 40 associations, 10 generalizations, 100 constraints	100 actions, 10 conditionals, 5 loops	4969 ms

He who sees things grow from the beginning will have the best view of them.
Aristotle

11 Related Work

Verification of the correctness of software models has been a topic extensively addressed in the literature. The work related to this thesis can be analyzed regarding three dimensions: (1) the domain, i.e. the type of model to be verified; (2) the correctness properties to be verified; and (3) the type of method employed to perform the verification.

In this chapter we review the previous work on verifying models according to the above dimensions. Firstly, Section 11.1 describes the three dimensions and briefly cites the related works. Then, Section 11.2 reviews the most representative works wrt the above dimensions and compares our proposal with them.

11.1 Dimensions of the related work

According to the three axis of the framework proposed in Chapter 4 (domain, property and method), the related work can be analyzed regarding three perspectives (see Figure 11.1):

- **Domain**. Refers to the kind of model to be verified.
- **Property**. Refers to the correctness property to be verified.
- **Method**. Refers to the type of method employed to perform the verification.

In the rest of this section we briefly describe these dimensions.

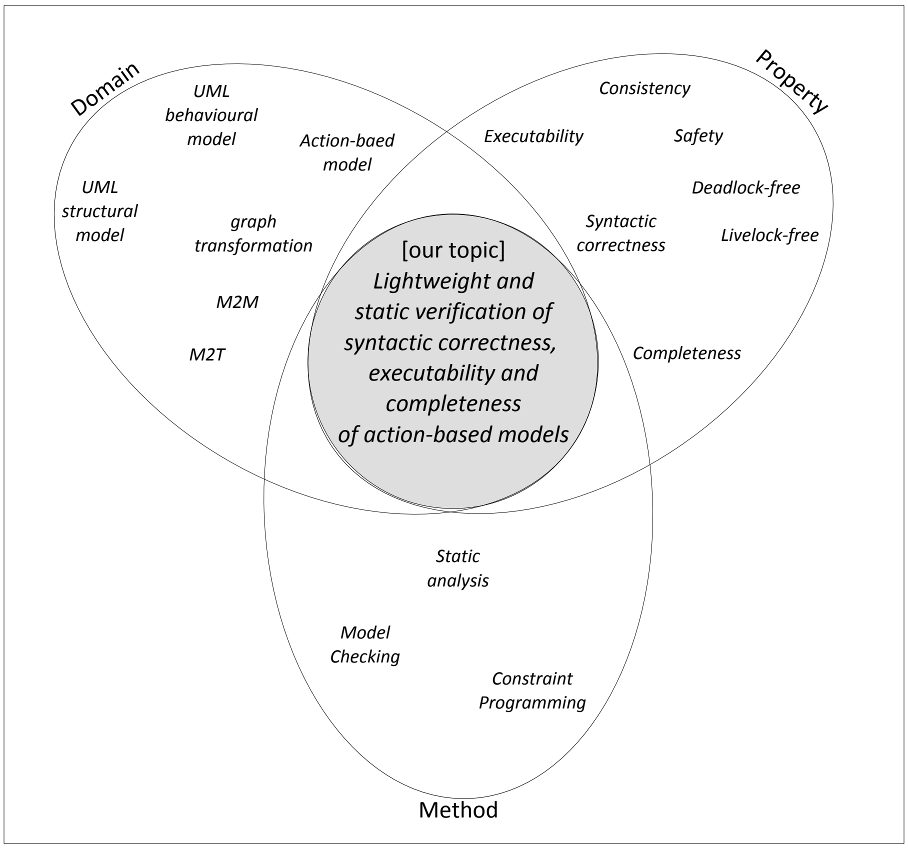


Figure 11.1. Related Work dimensions.

11.1.1 Domain

The *domain* dimension refers to the kind of model to be verified.

In the software modelling context, the focus of the verification may be the structural model or the behavioural model. Regarding the first group, there is a long tradition of methods devoted to the problem of verifying structural models. For instance, [102, 148, 150] verify several correctness properties over UML class diagrams.

Regarding the second group, there is also a broad set of research proposals devoted to the problem of verifying behavioural models. For instance, in the UML context, there are works focusing on verifying statechart diagrams [104, 105, 129], sequence diagrams [74], activity diagrams [3, 1, 22, 59], operations [33, 70, 149], xUML models [75, 185], or on verifying the consistent interrelationship between them [2, 44, 71], among others. On the other hand, in the model transformation context, there are also works focusing on the verification of M2M transformations described by means of QVT [178, 8], ATL [178, 179], Xtend [178], or graph transformation rules [31, 152], among others.

In Section 11.2 we review in detail the most related works.

11.1.2 Property

The *property* dimension refers to the correctness property to be verified.

Along the history of software engineering, several properties of software models have been addressed. In the following subsections we review the origin of the correctness properties studied in this thesis and we briefly introduce other properties that has been studied in the literature.

Syntactic Correctness

Syntactic correctness has its origin in programming languages. Syntactic correctness is a very well-known property that all programs written in a specific programming language must satisfy. In this context, compilers [6] are the responsible of checking whether the code of a program is correctly written in terms of the syntax of the programming language. The compiler recognizes well-formed and ill-formed programs, reporting errors, if any.

In a higher level of abstraction, syntactic correctness may also be checked in models, model transformations and any high-level representation conforming to a language (or metamodel). As an example, the UML specification [126] provides several Well-Formedness Rules (WFR) to ensure the syntactic correctness of models.

The most popular modelling tools used today (such as MagicDraw, ArgoUML, Poseidon or Eclipse UML2Tools) include some basic syntactic checks focused on the UML class diagram (e.g. to check that associations have at least two association ends). At most, some of them provide also syntactic checks over OCL integrity constraints (e.g. to check that the structure of the constraint is consistent with the OCL language). Regarding the tools that also allow the inclusion of actions (like Papyrus [130], which includes an Alf editor) they only include some basic syntactic checks. The current version of Papyrus, for instance, includes an Alf parser (to check the Alf code fulfills the Alf syntax) but it does not yet include advanced analysis to check the issues we propose in Chapter 5.

Executability

Executability has also been briefly studied in programming languages [81].

In a higher level of abstraction, as syntactic correctness, executability may be checked in models, model transformations and any high-level representation conforming to a language (or metamodel). As an example, [33] and [149] verify the executability of declarative operations represented by means of the OCL language. In Section 11.2 we review these works in more detail.

Completeness

The meaning of *completeness* is so broad that it can be applied to many different contexts. Olivé argues that “a conceptual model must be complete” meaning that “a complete conceptual model includes all knowledge relevant to the Information System” [127]. According to this definition, Tort et al. [169] propose a Test Driven Development (TDD) approach to develop complete conceptual models.

The above definition refers to the external quality of models. However, the notion of *completeness* we adopt in this thesis refers to the internal quality of models. Aligned with our notion, for instance, [178] verifies the completeness of a M2M transofrmation. In Section 11.2 we review this work in more detail.

Other properties

Besides the correctness properties studied along this thesis, an extensive list of additional properties over behavioural models has been studied. For instance, there are works focusing on checking properties such as: *deadlocks-free* [3, 22, 105, 129], *livelocks-free* [22, 105, 129], *safety* [74, 75, 104], *liveness* [22, 74, 104], *applicability* [33], *consistency* [2, 71], or *domain-specific properties* [8, 152, 179, 185], among others.

In Section 11.2 we describe these properties and the main works that address them.

11.1.3 Method

Finally, the *method* dimension refers to the type of method employed to perform the verification.

As we explained in Chapter 3, a variety of methods can be used to analyze a model. They can be classified into static/dynamic and formal/non-formal/lightweight.

As we will explain in Section 11.2, most of the related works [3, 2, 1, 33, 31, 71, 104, 105, 129, 149, 152] require translating the input model into a formalism where a dynamic and formal verification method (such as a solver, a model checker or a theorem prover) is available. Only few works [100, 178, 179] use static analysis to verify the models.

11.2 Comparing the Related Works

In this section we compare the works related with this thesis. According to the first dimension (*domain*), we classify the related approaches into those that focus on UML models (see Section 11.2.1) and those that focus on model to model transformations (see Section 11.2.2).

11.2.1 Verification of UML Models

A lot of research has been devoted to the problem of V&V (verify and validate) UML models. As we introduced, many works focus on the structural model [102, 148, 150], however, the works more related to our are those which focus on the behavioural model.

Although there is no work addressing exactly the same problem of our focus (i.e. verifying the syntactic correctness, the executability and the completeness of UML action-based operations), a lot of research has been done to address similar problems. In this section we review the related works that have at least one dimension (kind of model, property verified or verification method) in common with our work.

Table 11.1 classifies the related works that deal with the verification of UML behavioural models and positions our work in relation with them. For each approach, we include the following information:

- *Work.* References the work.
- *Model.* Indicates the kind of behavioural model targetted.
- *Supported Constraints.* Indicates whether OCL integrity constraints are considered when analyzing the models.
- *Include Actions?.* Indicates whether UML actions can be added to specify fine-grained details of the model.
- *Property.* Enumerates the main correctness properties addressed by the work.
- *Method.* Indicates the basic method employed during the verification.
- *Repairing Feedback.* Indicates whether the approach returns some kind of repairing feedback beyond a simple yes/no answer (or a counterexample).

As can be seen in Table 11.1, there are lots of works targetting the verification of several types of UML behavioural diagrams (statechart diagrams, activity diagrams, etc.). The most related items of each work are colored in Table 11.1.

Lilius et al. [105, 129] propose the *vUML* tool, a verification tool that uses the SPIN model checker [84] to detect basic error types on **UML statechart diagrams**. The detected errors are: *deadlocks* (a deadlock is a common problem in concurrent models and it is produced when is not possible to dispatch any event in any object), *livelocks* (an object contains a livelock if none of the states marked with the “progress” stereotype are visited infinitely often during a normal execution of the model), *reaching a state marked as invalid* (invalid states are introduced by the designer and represent an error condition on the statechart; the verification will fail if a state marked as invalid is reached), *violating a constraint of an state* (if a constraint does not hold, the verification fails), *sending an event to a terminated object* (it is an error to send events to a terminated object) and *queue overruns* (each object has a finite input queue associated that holds the events sent to the object until they are processed; sending an event to a full queue produces an error). Similarly, Latella et al. [104] translate UML statechart diagrams to PROMELA (the specification language of the SPIN model checker [84]) to allow the designer to automatically verify basic correctness properties such as: *safety* (the statechart diagram is

Table 11.1. UML related methods comparison.

Work	*Domain*			*Property*	*Method*	*Repairing Feedback?*
	Model	*Supported Constraints*	*Include UML Actions?*			
Lilius et al. [105, 129]	Statechart diagrams	None	No	Deadlocks, livelocks, etc.	Model checking	No
Latella et al. [104]	Statechart diagrams	None	No	Safety, liveness	Model checking	No
Grosu et al. [74]	Sequence diagrams	None	No	Safety, liveness	Model checking	No
Eshius et al. [59]	Activity diagrams	None	No	Safeness, etc.	Model checking	No
Bouabana-Tebibel et al. [22]	Activity diagrams	None	No	Deadlocks, livelocks, liveness, etc.	Model checking	No
Brosch et al. [26]	Statechart diagrams and sequence diagrams	No	No	Consistency	Model Checking	No
Gogolla et al. [70]	Declarative operations	Yes (all)	No	Validation checks	Animation	No
Cabot et al. [33]	Declarative operations	Yes (all)	No	WE, SE etc.	Constraint programming	No
Queralt et al. [149]	Declarative operations	Yes (subset)	No	WE etc.	Query containment	No
Abdelhalim et al. [3, 1]	Activity diagrams	None	Yes (fUML)	Deadlock free	Model checking	No
Abdelhalim et al. [2]	Statechart diagrams, activity diagrams	None	Yes (fUML)	Consistency	Model checking	No
Graw et at. [71]	Statechart diagram, sequence diagrams	None	Yes (fUML)	Consistency	Model checking	No
Hansen et al. [75]	xUML model	None	Yes (xUML)	Safety	Model checking	No
Xie et al. [185]	xUML model	None	Yes (xUML)	Domain-specific properties	Model checking	No
Bousee et al. [23]	SysML statechart diagram	Yes	Yes (Alf)	Safety	Theorem proving	No
Lai et al. [100]	Activity diagram	No	Yes (Alf)	Basic redundancies	Static analysis	No
our work	imp-OP	Subset	Yes (Alf)	Syntactic correctness, weak and strong executability, completeness	Static analysis	Yes

free from invalid transitions) and *liveness* (all valid transitions are possible). Likewise, Grosu et al. [74] study the *safety* and *liveness* properties over **UML sequence diagrams**.

Eshius et al. [59] translate **UML activity diagrams** to NuSMV (a symbolic model verifier [35]) to allow the designer to automatically verify a subset of the correctness properties that can be expressed in PLTL-X (a variant of the Past Linear Temporal Logic [107]) such as: *safeness* (an activity is safe if each node can be active at most once at the same time). Similarly, Bouabana-Tebibel et al. [22] translate UML activity diagrams to Object Petri Nets [92] and analyze the Petri Nets using a model checker to allow the designer to verify properties such as: *deadlock-free* (there are no UML states that prevent any activity to be invoked eventually), *livelock-free* (there are no loops of activities on the diagram) and *quasi-liveness* (guarantees that each UML activity can be invoked eventually). Furthermore, Brosch et al. [26] translate UML statechart diagrams and sequence diagrams to PROMELA [84] to allow the designer to automatically verify the consistency between both diagrams.

Other works, as our work, are focused on **UML operations**. For instance, the USE tool by Gogolla et al. [70] receives a UML class diagram and a set of declarative operations and it is able to validate the structure/behaviour according to the designer expectations (such as the consistency of UML models and the independence of OCL constraints) through animation and certification. Similarly, Cabot et al. [33] translate declarative operations (expressed by means of OCL pre- and postconditions) into a Constraint Satisfaction Problem (CSP) [172] to allow the designer to automatically verify several correctness properties such as: *weak executability* (an operation is weakly executable if its 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) and *strong executability* (an operation is strongly executable if, for every legal input satisfying the precondition, there is a legal output that satisfies the postcondition). Besides, they verify other properties such as: *applicability* (an operation is applicable if the precondition is satisfiable, i.e. if there is an input where the precondition evaluates to true), *redundant precondition* (the precondition of an operation is redundant if it is true for any legal input), *correctness preserving* (an operation is correctness preserving if, given a legal input, each possible output satisfying the postcondition is also legal), *immutability* (an operation is immutable if, for some input, it is possible to execute the operation without modifying the initial snapshot) and *determinism* (an operation 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). In the same line, Queralt et al. [149] translate declarative operations into logic to allow the designer to automatically verify several correctness properties such as: *applicability* and *weak executability.*

The above works by Cabot et al. [33] and Queralt et al. [149] are the closest to our work since they verify the **executability** of operations defined at the model level. In the following we itemize the similarities and differences of both works with respect to ours:

- A significant difference wrt our work is that both the above works [33, 149] depart from declarative operations specified by means of OCL pre and postconditions, instead of using imperative specifications[24] based on actions.

[24] Although our aim is not to compare declarative and imperative specifications, it is worth to note that there are methods that transform declarative operations into imperative ones [29].

- Regarding the expressivity of the input model, a strong point of [33] wrt our work is that they deal with more expressive models since they do not have any structural limitation, i.e. they consider general OCL constraints (annotated to the class diagram) during his analysis. On the other hand, Queralt et al. [149], as our work, take into account only a subset of OCL integrity constraints. In particular, they are not able to deal with those OCL expressions that include arithmetic operations, since they cannot be expressed in the logic representation they use to perform the verification.

- Regarding the method used to perform the verification, both works use dynamic and formal methods. In particular, Cabot et al. [33] use a Constraint Programming (CP) solver to search for a state of the domain in which all the constraints that define the problem are satisfied at the same time. On the other hand, Queralt et al. [149] use the Constructive Query Containment Method (CQC-Method) [63] to construct a state that fulfills a goal (i.e. the property) and satisfies all the constraints in the model. Both methods may be classified as dynamic and formal according to the classification we presented in Chapter 3. It means that both methods require to execute the model to search (in CP) or to construct (in CQC) the proper state. Although both methods show off about its efficiency, none of them describe its time complexity.

- A weak point wrt our work is that both works [33, 149] do not provide any repairing feedback. When the checked property is not satisfied, Cabot et al. [33] return a counterexample represented by means of a UML object diagram. Similarly, Queralt et al. [149] return a binary response to indicate whether the input test (designed to validate/verificate a specific property) is satisfied or not. Although both approaches point out the source of the problem, they do not indicate how the user can correct the error. This is because of the focus of the analytical methods is on exploring the model in order to find a solution (but not on finding errors).

- Finally, regarding the completeness of the method, this feature is not guaranteed by Cabot et al. [33] since they restrict the domains of the variables in order to guarantee decidability. It means they may fail to find an existing solution (i.e. as our work -when the user does not intervene-, they can return false positives, that is, they may conclude that a property is not satisfied although it is actually satisfied outside the search space explored during the verification). On the contrary, the method proposed by Queralt et al. [149] is complete (i.e. when an example for a property exists, it will always be found) but, in change, it is semi-decidable since it does not guarantee termination in all cases (since the query containment problem for the general case of queries and databases that the CQC method can cover is undecidable [62]). However, they have identified the cases in which their method may not terminate, so the designer is aware of this fact before performing the verification.

As can be seen in Table 11.1, although there are a lot of works addressing the verification of UML behavioural models, only a subset of these works support the inclusion of **UML actions** as part of the specification of their input behavioural UML diagrams (even when this is indeed allowed by the UML standard). For instance, Abdelhalim et al. [3, 1] translate UML activity diagrams (which include fUML abstract actions) into the Communicating Sequential

Processes (CSP) modelling language [158] to allow the designer to check whether the model is *deadlock-free* using a model checker. In another work [2] the same authors take as input a UML statechart diagram and a UML activity diagram (which includes fUML abstract actions) for each state of the statechart. Then, these diagrams are translated into a CSP (Communicating Sequential Processes) formalization in order to check the *consistency* between each UML statechart diagram and its corresponding fUML activity diagram using a model checker. Similarly, Graw et at. [71] transform UML statechart diagrams and UML sequence diagrams (both including the abstract syntax of UML 2.0 Action Semantics) into compositional Temporal Logic of Actions (cTLA) [101] formalization language in order to verify several *consistency checks* using a model checker.

Other works allow the inclusion of actions in the context of **xUML models**. For instance, Hansen et al. [75] translate xUML models into the process algebra mCRL2 [73] in order to verify *safety properties* in a domain-specific scenario (such as interlockings in the railway industry, to ensure that trains neither collide nor derail) using symbolic model checking. Similarly, Xie et al. [185] translate xUML models into S/R (the input language of the COSPAN model checker [76]) to allow the designer to verify *domain-specific properties* specified in logic.

Although there are several works [1, 3, 2, 71, 73, 75] addressing the verification of UML executable models including actions, only two recent works [23, 100] are aligned with the new standard **Alf action language** (all the above works use a non-standard action language to specify the concrete syntax of actions). Bousse et al. [23] translate SysML [66] statechart diagrams (including Alf actions) to the B formalization [159] in order to check $safety$ properties using theorem prover techniques. On the other hand, Lai et al. [100] propose a preliminary framework to verify basic *redundancies* over Alf activities (such as detecting unused activities or unused class members). The verification relies on static analysis and uses a library of suspicious patterns to detect the above anomalies. However, none of these works check any of the properties we deal with in this thesis.

As also can be seen in Table 11.1, most of the related works require translating the input model into a formalism where a **dynamic verification** method (such as a solver, a model checker or a theorem prover) is available. For instance, Cabot et al. [33] translate the model into a Constraint Satisfaction Problem (CSP) [172] and then use a Constraint Programming solver. Queralt et al. [149] translate the model into logic and then use a the Constructive Query Containment Method (CQC-Method) [63]. Bousse et al. [23] translate the model into the B formalization [159] in order to use theorem proving techniques. Other broad set of works translates the model into the specification language of a model checker in order to check the input property. For instance, Lilius et al. [105, 129], Latella et al. [104] and Brosch et al. [26] translate the model to PROMELA (the specification language of the SPIN model checker [84]); Abdelhalim et al. [3, 2, 1] translate the model into the Communicating Sequential Processes (CSP) [158] (the specification language of the Failures-Divergences Refinement tool (FDR2) [64]); Graw et at. [71] transform the model into compositional Temporal Logic of Actions (cTLA) [101] (the specification language of the TLC (Temporal Logic Checker) [187]).

Model checkers work by generating and analyzing all the potential executions at run-time and evaluating whether for each (or some) execution scenario the given property is satisfied. Even with the several optimizations available (as partial order reduction or state compression), methods based on model checking suffer from the state-explosion problem (i.e. the number of potential executions to analyze grows exponentially) compromising the efficiency of the method. Therefore, in general, it is not possible to explore all possible executions. This implies that the results provided by these methods may be not complete, i.e. the absence of a solution cannot be used as a proof, because a property may be satisfied outside the search space explored during the verification.

However, the above formal dynamic methods are more potent wrt our lightweight static method. It means that these methods are allowed to verify arbitrary properties (previously formalized in the proper language) considering a more expressive model (allowing all types of integrity constraints) and without requiring the user intervention.

As suggested in [58], static and dynamic analysis can interact. In this sense, we believe the lightweight static methods presented in this thesis could be used to perform a first correctness analysis, basic to ensure a fundamental quality level on action-based operations. Then, designers could proceed with a more detailed analysis adapting the current approaches presented above to the verification of behaviours specified with Alf. For instance, example execution traces that lead to an error state would help designers to detect particular scenarios not yet appropriately considered.

Finally, we would like to remark that most of the cited methods just provide a binary response (showing whether the model satisfies the given property or not) and, at most some provide example execution traces that do (not) satisfy the property (i.e. a counterexample). However, none clearly identify the source of the problems nor assist the designer to repair them. Instead, a clear benefit of our method is the kind of feedback provided, that helps the designer repairing her models.

11.2.2 Verification of M2M Transformations

Since quality of model transformation is a more recent topic, less work has been done in this direction. However, relevant efforts have been recently done to explore this area.

Table 11.2 classifies the related works that deal with the verification of M2M transformations and positions our work in relation with them. For each approach, we include the following information:

- *Work.* References the work.
- *Domain.* Indicates the kind of behavioural model targetted.
- *Property.* Enumerates the main correctness properties addressed by the work.
- *Method.* Indicates the basic method employed during the verification.
- *Repairing Feedback.* Indicates whether the approach returns some kind of feedback beyond a simple yes/no answer.

As before, the most related items of each work are colored in the Table 11.2.

In the context of **exogenous out-place transformations** (i.e. using transformation languages), Anastasakis et al. [8] formalize QVT [122] model transformations in Alloy [90] (a

Table 11.2. M2M related methods comparison.

Work	*Domain*	*Properties*	*Method*	*Repairing Feedback?*
Anastasakis et al. [8]	QVT M2M Transformations	Domain-specific properties	Simulation	No
Vieira et al. [179]	ATL M2M Transformations	Domain-specific properties	Static Analysis	No
Vallecillo et al. [176]	ATL M2M Transformations	Domain-specific and syntax related properties	Testing	No
Amstel et al. [178]	ATL, QVTo and Xtend M2M Transformations	Covering (i.e. completeness) et al.	Static analysis	Yes
Cabot et al. [31]	Graph Transformation Rules	WE, SE, et al.	Constraint Programming	No
Rivera et al. [152]	Graph Transformation Rules	Domain-specific properties	Model Checking	No
our work	ATL Transformations, Graph Transformation Rules	WE, SE, completeness	Static Analysis	Yes

first-order relation specification language). Then, the Alloy analyzer simulates these transformations to explore the potential combinations of target models that can be generated by the given transformation rules. Besides, several types of *domain-specific assertions* can be used to verify that a target model will always be well-formed given the transformation rule. Note that using assertions one could specify our *executability* property over QVT transformations. However, this property should be redefined for each domain. Similarly, Vieira et al. [179] provide an API to manipulate ATL [94] transformation elements in order to assist the designer during the inspection of the model transformations. On the other hand, Vallecillo et al. [176] propose a testing based method to check the correctness of ATL transformations. To do that, first they define the requirements that a transformation has to fulfill (i.e. its expected behaviour). Afterward, they generate several input test models, which are then automatically transformed into output models and checked against the set of requirements defined for the transformation, using the USE tool [70].

However, the most related work dealing with exogenous out-place M2M transformations is one by Van Amstel et al. [178]. In this work, the authors provide several metamodel *completeness* visualization techniques (they use the term *coverage* instead of *completeness*) that can be used to visually analyze the relation between a model transformation and the metamodels in which it is defined on. The first property, *metamodel coverage*, analyzes what elements (metaclasses, attributes and references) of the input/output metamodels are covered by the transformation. An input metamodel element is covered if it serves as input for a transformation function in the transformation, while an output metamodel element is covered if it is generated as output by a transformation function in the transformation. The second property, *metamodel coverage relation*, trace the relation between a transformation element and the metamodel element it covers. The result of the checked properties is visually presented using two helpful visualization techniques. When verifying the first property, the metaclasses and references that are covered by the transformation are colored, while the attributes that are covered are underlined. When verifying the second property, a line between a metaclass and a transformation element exists if and only if the metaclass is covered by the transformation element. Both visualizations may be zooming and navigated in order to understand in detail the completeness of a model transformation and to find errors during development. Similarly to our method, this work uses static analysis to check the previous properties. With respect to our method, this work has the advantage that it uses visualization techniques that help the designer to comprehend the feedback. Even though this work also deals with other correctness properties (*metrics for measuring model transformations*, and *structure and trace analysis*), it does not check the executability neither the syntactic correctness of M2M transformations.

In the context of **endogenous in-place transformations** (i.e. using graph transformation languages), Rivera et al. [152] translate graph transformation rules into a Maude specification [40] (a rewriting logic-based language with formal analysis support). Then, using the tools and techniques that Maude provides, they perform simulation, reachability and model-checking to analyze *specific-domain properties* of such models.

However, the most related work dealing with graph transformations is one by Cabot et al. [31]. In this work, the authors translate graph transformation rules into OCL declarative operations and then use their *UML2CSP* tool [32] (which translates the OCL invariants into CSP) to allow the designer to automatically verify the *weak/strong executabilily* of graph transformation rules among other properties. However, even though the results of their analysis are presented in a graphical front-end tool, they do not provide any kind of feedback to help the designers repair her erroneous rules.

12 Conclusions

A young martial artist kneeling before the Master Sensei in a ceremony to receive a hard-earned black belt. After years of relentless training, the student has finally reached a pinnacle of achievement in the discipline.
"Before granting the belt, you must pass one more test", says the sensei. "I am ready", responds the student, expecting perhaps one final round of sparring. "You must answer the essential question: What is the true meaning of the black belt?" "The end of my journey", says the student. "A well-deserved reward for all my hard work." The sensei waits for more. Clearly, he is not satisfied. Finally, the sensei speaks. "You are not yet ready for the black belt. Return in one year."
A year later, the student kneels again in front of the sensei. "What is the true meaning of the black belt?" asks the sensei. "A symbol of distinction and the highest achievement in our art", says the student. The sensei says nothing for many minutes, waiting. Clearly, he is not satisfied. Finally, he speaks. "You are still not ready for the black belt. Return in one year."
A year later, the student kneels once again in front of the sensei. And again the sensei asks: "What is the true meaning of the black belt" "The black belt represents the beginning – the start of a never-ending journey of discipline, work, and the pursuit of an ever-higher standard", says the student. "Yes. You are now ready to receive the black belt and begin your work."

The Parable of the Black Belt

Executable models play a key role in many development methods by facilitating the immediate simulation/implementation of the software system under development. Given its increasing importance and the impact of their correctness on the final quality of software systems derived from them, the existence of methods to verify the correctness of such models is becoming crucial. The aim of this thesis has been to propose a framework for assessing and improving the quality of executable models based on actions. To define the scope of this thesis, in Chapter 1 we formulated the main research question as follows: *How can the quality of executable models be improved?*

The aim of this chapter is to summarize the contributions of this thesis regarding the above research question (Section 12.1) and to provide directions for further research related to the topic of this thesis (Section 12.2).

12.1 Contributions

In Chapter 1 we decomposed the main research question into four specific research questions. In this section we reexamine these research questions in order to summarize the contributions of this thesis.

12.1.1 Quality Properties for Executable Models

In Chapter 2 we defined an *executable model* as "a model with a behavioural specification detailed enough so that it can be systematically implemented or executed in the production environment". Previously, in Chapter 1 we defined the quality of a model as "the degree to which a set of internal properties (also called quality goals) is present". In order to study the quality of executable models, in Chapter 1 we set out the first research question (RQ_1): *How can the quality of executable models be decomposed into quality properties?*

In Chapter 4 we proposed a set of *correctness properties* that we believe all executable models (consisting of a set of action-based operations) should accomplish to ensure their internal quality. Some of these properties were previously studied in the literature (for instance, to verify the source code of a program). However, in this thesis we redefined its meaning to deal with action-based operations defined within a model.

Then, the correctness properties we addressed in this thesis can be summarized as:

- **Syntactic correctness** (see Chapter 5). An action-based operation is *syntactically correct* if all the statements in the operation conform to the syntax of the language in which it is described (i.e. the Alf action language [124]). According to the Lindland et al. framework for evaluating the quality of models [106], syntactic correctness works at the syntactic level of model's correctness, then, ensuring this property improves the *syntactic quality* of the executable model we are analyzing.
- **Executability** (see Chapter 6). The *executability* of an action-based-operation is its ability to be executed without breaking the integrity constraints defined in the structural model. Executability can be studied regarding two levels of correctness. On the one hand, an action-based operation is *weakly executable* when there is a chance that a user may successfully execute the operation, i.e. when there is at least an initial state of the system for which the execution of the actions included in the operation evolves this state to a new system state that satisfies all integrity constraints of the structural model. On the other hand, an action-based operation is *strongly executable* when it is always successfully executed, i.e. when every time a user executes the operation, the effect of the actions included in it evolves the initial state of the system to a new system state that satisfies all integrity constraints of the structural model. According to [106], executability works at the semantic level of model's correctness, then, ensuring this property improves the *semantic quality* of the executable model we are analyzing.
- **Completeness** (see Chapter 7). A set of action-based operations is *complete* when all the possible changes on the system state can be performed through the execution of these operations. Otherwise, there will be parts of the system that users will not be able to modify since no available behaviour address their modification. According to [106], as executability, completeness works at the semantic level of model's correctness, then, ensuring this property improves the *semantic quality* of the executable model we are analyzing.

The more lower the quality goal, the more fundamental the correctness property is. It means that the syntactic quality ensures a $fundamental$ quality level that all executable models should guarantee; and the semantic quality ensures a *significant* quality level that all executable models should also guarantee. According to this classification, we believe that *syntactic correctness*, *executability* and *completeness* are mandatory properties that all correct executable models should guarantee.

12.1.2 Lightweight methods for verifying Executable Models

In Chapter 3 we introduced several types of methods that can be employed for checking the quality of models. Then, our second research question (RQ_2) was: *What methods can be employed to support the verification of the quality properties of executable models?*

In this thesis we proposed specific methods for verifying the proposed correctness properties. Chapters 5 to 7 describe in detail each method. In summary, we would like to remark that all the proposed methods are lightweight, i.e. they directly reason over a model formalized in Alf language, they are based on a static analysis of the model (i.e. no execution of the behaviour is required) and they provide meaningful feedback. This leads on a set of methods that can be easily integrated in the current software development processes and CASE tools.

As a trade-off, our lightweight static methods are not able to verify other arbitrary properties that can be logically formulated. Besides, they may require (only when verifying executability) the user intervention in order to return a more precise result. For these reasons, we believe the methods presented in this thesis could be used to perform a first correctness analysis, basic to ensure a fundamental quality level on action-based operations. Then, designers could proceed with a more detailed analysis adapting other methods (such as model checking) to perform a more complete verification.

In order to prove the feasibility of the methods proposed in this thesis, we have built a prototype tool (see Chapter 9) that implements the methods for verifying the weak and strong executability of an executable model consisting in a UML class diagram and a set of Alf-based operations.

12.1.3 Feedback

In order to guarantee the usefulness of the methods presented in this thesis, we set out the third research question (RQ_3): *What kind of feedback can help the designer to improve her executable models?*

Actually, one of the goals of our verification framework is to provide a useful feedback to help the designers improve her models. To achieve this goal, all the verification methods proposed in this thesis return either a positive answer, meaning that the model achieves the checked property, or a corrective feedback otherwise. In order to make the corrective feedback understandable to the designer, it is always expressed in the same language used to express the input model (i.e. Alf actions, ATL sentences or graph transformation elements).

In this sense, we would like to remark that all the related works studied (see Chapter 11) only provide a binary response (whether the model satisfies the property or not) and, at most, some of them provide (counter)example execution traces that do (not) satisfy the property. None of them identify the source of the problems nor assist the designer to repair them, which is one of the goals of our proposal.

12.1.4 Application in Model Transformations

The properties presented in this thesis can also serve to analyze the quality of other types of models. As a consequence, the methods presented in this thesis may be applied in other contexts. This led to the fourth research question (RQ_4): *What kinds of executable models can be verified using these methods?*

Model transformations automate the translation of models between a source and a target language using a model transformation language (such as ATL or graph transformation rules). Model transformations are in many ways similar to traditional software artifacts (for instance, they may change according to the user expectations, they may be reused, and so on). Therefore, they need to be verified as well. In Chapter 8 we adapted our lightweight and static methods to the context of model to model transformations. In particular, we redefined our *weak executability* and *completeness* properties in the context of model transformations and we redesigned the previous methods to verify these properties in the context of M2M transformations.

12.2 Directions for Further Research

The work proposed in this thesis can be extended in many ways. In this section we present several directions for further research in this area, according to the three dimensions of our framework.

Domain

Regarding the kind of model to be verified, the executable models addressed in this thesis could be extended. In order to be more expressive, new types of OCL constraints could be considered. Adding constraints imply to identify its OCL pattern and to extend the methods proposed in this thesis to address these constraints. However, note that - as we justified in Chapter 6 - not all possible constraints can be added to our method.

On the other hand, the methods presented in this thesis could also be applied in other types of executable models. In the context of UML, for instance, other behavioural diagrams including actions to define the low level behaviour (such as activity diagrams or statechart diagrams) could be analyzed in terms of the properties addressed in this thesis. Also BPM models could be analyzed concerning similar properties.

Properties

Regarding the properties to be verified, the definition of some properties proposed in this thesis could be extended. In particular, as we stated in Chapter 7, the meaning of the *completeness* property could be extended to consider not only the modification actions appearing in the model but also the reading actions.

Besides, other semantic properties could be added in our verification framework. One example is the *applicability* of an operation, considering that an operation is applicable if it may be invoked, i.e. if the required parameters to invoke the operation may be created executing other operations existing in the behavioural model. In other words, an operation is applicable if its execution requires an empty initial state or a state that may be built executing other existing operations. This property, as the rest of the properties proposed in this thesis, may be statically analyzed reasoning over the input/output parameters of the operations.

Method

Regarding the method employed to perform the verification, the verification framework presented in this thesis could be complemented with other dynamic formal methods to allow the verification of more complex and specific properties. In this way, for instance, two concrete research lines could be addressed:

1. The first line consists in providing an automatic translation between our Alf-based operations and OCL pre- and postconditions in order to be able to use the UML2CSP tool [32]. Using this tool, the designers could directly check additional properties such as *applicability* (an Alf-based operation is applicable if it may be invoked) or *determinism* (an Alf-based operation 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) among others.
2. The second line consists in providing an automatic translation between our Alf-based operations and the input language of a model checker (such as PROMELA [84] or Maude [40]). Then, the designers could also perform simulation, reachability and model checking to analyze specific-domain properties over the Alf-based behavioural model. However, note that this translation requires expertise on the formal notation.

On the other hand, regarding the kind of feedback provided by our methods, its representation could be improved by using visualization techniques like those proposed in [178].

The study of the proposed further work will help to extend our verification framework to be more complete. The verification framework should integrate all the studied properties and its respective verification methods (both lightweight and no-lightweight) in all the addressed domains. Then, a set of guidelines should be proposed to help the designers choose the most appropriate verification method for the model they have defined, depending on the target property and the verification trade-offs (expressiveness, completeness, efficiency,...) they are ready to accept.

In summary, given the increasing importance of executable models in the most relevant software development methods used today, the assessment of the correctness of such models is a research topic that should be studied in more deep.

Bibliography

[1] Abdelhalim, I., Schneider, S., and Treharne, H. An integrated framework for checking the behaviour of fUML models using CSP. *International Journal on Software Tools for Technology Transfer* (2002, DOI: 10.1007/s10009-012-0243-0).

[2] Abdelhalim, I., Schneider, S., and Treharne, H. Towards a Practical Approach to Check UML/fUML Models Consistency using CSP. In *ICFEM* (2011), pp. 33–48.

[3] Abdelhalim, I., Sharp, J., Schneider, S. A., and Treharne, H. Formal Verification of Tokeneer Behaviours Modelled in fUML using CSP. In *ICFEM* (2010), pp. 371–387.

[4] Agarwal, B., Tayal, S., and Gupta, M. *Software Engineering and Testing*. Infinity Science Series. Jones & Bartlett Publishers, Incorporated, 2009.

[5] Agesen, O. Constraint-Based Type Inference and Parametric Polymorphism. In *SAS* (1994), pp. 78–100.

[6] Aho, A., Lam, M., Sethi, R., and Ullman, J. *Compilers: Principles, Techniques, and Tools*, vol. 1009. Pearson/Addison Wesley, 2007.

[7] Aiken, A. Introduction to Set Constraint-Based Program Analysis. *Sci. Comput. Program. 35*, 2 (1999), 79–111.

[8] Anastasakis, K., Bordbar, B., and Kster, J. M. Analysis of Model Transformations via Alloy. In *MoDeVVa* (2007), pp. 47–56.

[9] Arsac, J., and Kodratoff, Y. Some Techniques for Recursion Removal from Recursive Functions. *ACM Trans. Program. Lang. Syst. 4*, 2 (Apr. 1982), 295–322.

[10] Atkinson, C., and Kühne, T. Model-Driven Development: A Metamodeling Foundation. *IEEE Software 20*, 5 (2003), 36–41.

[11] Ayer, A. *Language, Truth and Logic*. Penguin Books, 1936.

[12] Bansiya, J., and Davis, C. G. A Hierarchical Model for Object-Oriented Design Quality Assessment. *IEEE Trans. Software Eng. 28*, 1 (2002), 4–17.

[13] Basili, V. R. The role of experimentation in software engineering: past, current, and future. In *Proceedings of the 18th international conference on Software engineering* (Washington, DC, USA, 1996), ICSE '96, IEEE Computer Society, pp. 442–449.

[14] Beck, K. Simple Smalltalk Testing: With Patterns. Tech. rep., First Class Software, Inc., 1989.

[15] Bertolino, A., and Marré, M. Automatic Generation of Path Covers Based on the Control Flow Analysis of Computer Programs. *IEEE Trans. Softw. Eng. 20*, 12 (Dec. 1994), 885–899.

[16] Bishop, P. G. Estimating residual faults from code coverage. In *SAFECOMP* (2002), pp. 163–174.

[17] Boehm, B. W., and Basili, V. R. Software Defect Reduction Top 10 List. *IEEE Computer 34*, 1 (2001), 135–137.

[18] Boehm, B. W., Brown, J. R., and Lipow, M. Quantitative Evaluation of Software Quality. In *ICSE* (1976), pp. 592–605.

[19] Boehm, B. W., McClean, R. K., and Urfrig, D. B. Some Experience with Automated Aids to the Design of Large-Scale Reliable Software. *SIGPLAN Not. 10*, 6 (Apr. 1975), 105–113.

[20] Bollobás, B. *Modern Graph Theory.* Graduate Texts in Mathematics. Springer, 1998.

[21] Booch, G. The Promise, the Limits, the Beauty of Software, 2007.

[22] Bouabana-Tebibel, T., and Belmesk, M. An Object-Oriented Approach to Formally Analyze the UML 2.0 Activity Partitions. *Information & Software Technology 49*, 9-10 (2007), 999–1016.

[23] Bousse, E., Mentré, D., Combemale, B., Baudry, B., and Takaya, K. Aligning SysML with the B Method to Provide V&V for Systems Engineering. In *Model-Driven Engineering, Verification, and Validation 2012 (MoDeVVa 2012)* (Innsbruck, Autriche, Sept. 2012).

[24] Brambilla, M., Cabot, J., and Wimmer, M. *Model Driven Software Engineering in Practice.* Morgan & Claypool, 2012.

[25] Brooks, F. P. *The Mythical Man-Month : Essays on Software Engineering*, anniversary ed. Addison-Wesley Pub. Co, Aug. 1995.

[26] Brosch, P., Egly, U., Gabmeyer, S., Kappel, G., Seidl, M., Tompits, H., Widl, M., and Wimmer, M. Towards Scenario-Based Testing of UML Diagrams. In *TAP* (2012), pp. 149–155.

[27] Brown, D. *An Introduction to Object-Oriented Analysis: Objects and UML in plain English.* Wiley, 2002.

[28] Burch, J. R., Clarke, E. M., McMillan, K. L., and Dill, D. L. Sequential Circuit Verification Using Symbolic Model Checking. In *DAC* (1990), pp. 46–51.

[29] Cabot, J. From Declarative to Imperative UML/OCL Operation Specifications. In *ER* (2007), pp. 198–213.

[30] Cabot, J. The New Executable UML Standards: fUML and Alf. `http://modeling-languages.com/new-executable-uml-standards-fuml-and-alf/`, 2011.

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

[32] Cabot, J., Clarisó, R., and Riera, D. UMLtoCSP: a tool for the Formal Verification of UML/OCL models using Constraint Programming. In *ASE* (2007), pp. 547–548.

[33] Cabot, J., Clarisó, R., and Riera, D. Verifying UML/OCL Operation Contracts. In *IFM* (2009), pp. 40–55.

[34] Chusho, T. Test Data Selection and Quality Estimation Based on the Concept of Essential Branches for Path Testing. *IEEE Trans. Softw. Eng. 13*, 5 (May 1987), 509–517.

[35] Cimatti, A., Clarke, E. M., Giunchiglia, F., and Roveri, M. NUSMV: A New Symbolic Model Checker. *STTT 2*, 4 (2000), 410–425.

[36] Clarke, E. M., and Emerson, E. A. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs* (1981), pp. 52–71.

[37] Clarke, E. M., Emerson, E. A., and Sistla, A. P. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Trans. Program. Lang. Syst. 8*, 2 (1986), 244–263.

[38] Clarke, E. M., and Wing, J. M. Formal Methods: State of the Art and Future Directions. *ACM Comput. Surv. 28*, 4 (1996), 626–643.

[39] Clarke, Jr., E. M., Grumberg, O., and Peled, D. A. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[40] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Talcott, C. L., Eds. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic* (2007), vol. 4350 of *Lecture Notes in Computer Science*, Springer.

[41] Clayberg, E., and Rubel, D. *Eclipse Plug-ins*, 3 ed. Addison-Wesley Professional, 2008.

[42] Click, C., and Cooper, K. D. Combining analyses, combining optimizations. *ACM Trans. Program. Lang. Syst. 17*, 2 (1995), 181–196.

[43] Costal, D., Gómez, C., Queralt, A., Raventós, R., and Teniente, E. Improving the definition of general constraints in UML. *Software and System Modeling 7*, 4 (2008), 469–486.

[44] Costal, D., Sancho, M.-R., and Teniente, E. Understanding Redundancy in UML Models for Object-Oriented Analysis. In *CAiSE* (2002), pp. 659–674.

[45] COUSOT, P. Abstract Interpretation Based Formal Methods and Fture Challenges. In *Informatics* (2001), pp. 138–156.

[46] COUSOT, P. Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. *Theor. Comput. Sci. 277*, 1-2 (2002), 47–103.

[47] COUSOT, P., AND COUSOT, R. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Aproximation of Fixpoints. In *POPL* (1977), pp. 238–252.

[48] CRAIGEN, D., CRAIGEN, D., CANADA, O., CANADA, O., GERHART, S., GERHART, S., RALSTON, T., AND BROWN, R. H. *An International Survey of Industrial Applications of Formal Methods Volume 2 Case Studies.* 1993.

[49] CUADRADO, J. S., MOLINA, J. G., AND TORTOSA, M. M. RubyTL: un Lenguaje de Transformación de Modelos Extensible. In *DSDM* (2006).

[50] DINH-TRONG, T. T., GHOSH, S., FRANCE, R. B., HAMILTON, M., AND WILKINS, B. UMLAnT: an Eclipse plugin for animating and testing UML designs. In *ETX* (2005), pp. 120–124.

[51] DROMEY, R. G. A Model for Software Product Quality. *IEEE Trans. Software Eng. 21*, 2 (1995), 146–162.

[52] EASTERBROOK, S. M., LUTZ, R. R., COVINGTON, R., KELLY, J., AMPO, Y., AND HAMILTON, D. Experiences using lightweight formal methods for requirements modeling. *IEEE Trans. Software Eng. 24*, 1 (1998), 4–14.

[53] ECLIPSE. Platform Plug-in Developer Guide. `http://www.eclipse.org/documentation/`.

[54] ECLIPSE. Modeling Platform / Eclipse Con Europe Nov 2 2011. `http://wiki.eclipse.org/ModelingPlatform/EclipseConEuropeNov2_2011`, 2011.

[55] EDELKAMP, S., LEUE, S., AND LLUCH-LAFUENTE, A. Directed Explicit-State Model Checking in the Validation of Communication Protocols. *STTT 5*, 2-3 (2004), 247–267.

[56] EHRIG, H., EHRIG, K., PRANGE, U., AND TAENTZER, G. *Fundamentals of algebraic graph transformation.* Springer-Verlag, 2006.

[57] ENDRES, A. An Analysis of Errors and Their Causes in System Programs. *SIGPLAN Not. 10*, 6 (Apr. 1975), 327–336.

[58] ERNST, M. D. Static and dynamic analysis: synergy and duality. In *PASTE* (2004), p. 35.

[59] ESHUIS, R. Symbolic Model Checking of UML Activity Diagrams. *ACM Trans. Softw. Eng. Methodol. 15*, 1 (2006), 1–38.

[60] FAGAN, M. E. Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal 15*, 3 (1976), 182–211.

[61] Fagan, M. E. Advances in Software Inspections. *IEEE Trans. Software Eng. 12*, 7 (1986), 744–751.

[62] Farré, C., Teniente, E., and Urpí, T. The constructive method for query containment checking. In *DEXA* (1999), pp. 583–593.

[63] Farré, C., Teniente, E., and Urpí, T. Checking Query Containment with the CQC method. *Data Knowl. Eng. 53*, 2 (2005), 163–223.

[64] Formal Systems Oxford. *FDR 2.91 manual*, 2010.

[65] Freedman, D. P., and Weinberg, G. M. *Handbook of Walkthroughs, Inspections, and Technical Reviews: Evaluating Programs, Projects, and Products*, 3rd ed. Dorset House Publishing Co., Inc., New York, NY, USA, 2000.

[66] Friedenthal, S., Moore, A., and Steiner, R. *A Practical Guide to SysML: Systems Modeling Language*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[67] Garousi, V., Briand, L. C., and Labiche, Y. Control Flow Analysis of UML 2.0 Sequence Diagrams. In *Proceedings of the First European conference on Model Driven Architecture: foundations and Applications* (Berlin, Heidelberg, 2005), ECMDA-FA'05, Springer-Verlag, pp. 160–174.

[68] Garvin, D. What does product quality really mean? *Sloan Management Review 1*, 26 (1984), 469–486.

[69] Genero, M., Piattini, M., and Chaudron, M. R. V. Quality of UML models. *Information & Software Technology 51*, 12 (2009), 1629–1630.

[70] Gogolla, M., Büttner, F., and Richters, M. USE: A UML-based specification environment for validating UML and OCL. *Sci. Comput. Program. 69*, 1-3 (2007), 27–34.

[71] Graw, G., and Herrmann, P. Transformation and Verification of Executable UML Models. *Electr. Notes Theor. Comput. Sci. 101* (2004), 3–24.

[72] Groce, A., and Joshi, R. Extending Model Checking with Dynamic Analysis. In *VMCAI* (2008), pp. 142–156.

[73] Groote, J. F., Mathijssen, A., Reniers, M. A., Usenko, Y. S., and van Weerdenburg, M. The Formal Specification Language mCRL2. In *MMOSS* (2006).

[74] Grosu, R., and Smolka, S. A. Safety-Liveness Semantics for UML 2.0 Sequence Diagrams. In *ACSD* (2005), pp. 6–14.

[75] Hansen, H. H., Ketema, J., Luttik, B., Mousavi, M. R., van de Pol, J., and dos Santos, O. M. Automated Verification of Executable UML Models. In *FMCO* (2010), pp. 225–250.

[76] Hardin, R., Har'El, Z., and Kurshan, R. COSPAN. In *8th International Conf. on Computer Aided Verification* (1996).

[77] Harel, D. Biting the Silver Bullet - Toward a Brighter Future for System Development. *IEEE Computer 25*, 1 (1992), 8–20.

[78] Harrold, M. J., and Soffa, M. L. Interprocedual Data Flow Testing. *SIGSOFT Softw. Eng. Notes 14*, 8 (Nov. 1989), 158–167.

[79] Heintze, N. Set-Based Analysis of ML Programs. In *LISP and Functional Programming* (1994), pp. 306–317.

[80] Heitmeyer, C. On the need for practical formal methods. In *In Formal Techniques in RealTime and Real-Time Fault-Tolerant Systems, Proc., 5th Intern. Symposium (FTRTFT'98* (1998), Springer Verlag, pp. 18–26.

[81] Hermenegildo, M. V., Puebla, G., Bueno, F., and López-García, P. Integrated program debugging, verification, and optimization using abstract interpretation (and the ciao system preprocessor). *Sci. Comput. Program. 58*, 1-2 (2005), 115–140.

[82] Herrington, J. *Code Generation in Action.* Manning Publications Co., Greenwich, CT, USA, 2003.

[83] Hevner, A. R., March, S. T., Park, J., and Ram, S. Design Science in Information Systems Research. *MIS Quarterly 28*, 1 (2004), 75–105.

[84] Holzmann, G. *Spin Model Checker, the: Primer and Reference Manual*, first ed. Addison-Wesley Professional, 2003.

[85] IEEE. Standard for Software Reviews, IEEE Std 1028-1997. *Electronics*, March (1998).

[86] IEEE. Standard for Software Verification and Validation, IEEE Std 1012-1998. *Electronics* (1998).

[87] International Standards Organization (ISO). *ISO Standard 9000-2000: Quality Management Systems: Fundamentals and Vocabulary*, 2000.

[88] International Standards Organization (ISO), International Electrotechnical Commission (IEC). *ISO Standard 9126: Software Product Quality*, 2001.

[89] Jackson, D. Alloy: a Lightweight Object Modelling Notation. *ACM Trans. Softw. Eng. Methodol. 11*, 2 (Apr. 2002), 256–290.

[90] Jackson, D. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, 2006.

[91] Jackson, D., and Wing, J. Lightweight Formal Methods. *ACM Comput. Surv. 28*, 4es (1996), 21.

[92] Jensen, K., and Kristensen, L. *Coloured Petri Nets: Modelling and Validation of Concurrent Systems.* Springer, 2009.

[93] Jones, C. B. Formal Methods Light. *ACM Comput. Surv. 28*, 4es (1996), 20.

[94] Jouault, F., and Kurtev, I. Transforming Models with ATL. In *MoDELS Satellite Events* (2005), pp. 128–138.

[95] Juristo, N., and Moreno, A. M. *Basics of Software Engineering Experimentation*, 1st ed. Springer Publishing Company, Incorporated, 2010.

[96] Kennedy Carter. *UML ASL Reference Guide*, 2003.

[97] Kern, C., and Greenstreet, M. R. Formal Verification in Hardware Design: a survey. *ACM Trans. Design Autom. Electr. Syst. 4*, 2 (1999), 123–193.

[98] Kolovos, D. S., Paige, R. F., and Polack, F. The Epsilon Transformation Language. In *ICMT* (2008), pp. 46–60.

[99] Krogstie, J., Sindre, G., and Jørgensen, H. D. Process Models Representing Knowledge for Action: a Revised Quality Framework. *EJIS 15*, 1 (2006), 91–102.

[100] Lai, Q., and Carpenter, A. Defining and verifying behaviour of domain specific language with fUML. In *Proceedings of the Fourth Workshop on Behaviour Modelling - Foundations and Applications* (New York, NY, USA, 2012), BM-FA '12, ACM, pp. 1:1–1:7.

[101] Lamport, L. The Temporal Logic of Actions. *ACM Trans. Program. Lang. Syst. 16*, 3 (1994), 872–923.

[102] Lano, K., Clark, D., and Androutsopoulos, K. UML to B: Formal Verification of Object-Oriented Models. In *IFM* (2004), pp. 187–206.

[103] Larsen, P. G., Fitzgerald, J. S., and Wolff, S. Are formal methods ready for agility? a reality check. In *FM+AM* (2010), pp. 13–25.

[104] Latella, D., Majzik, I., and Massink, M. Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-Checker. *Formal Asp. Comput. 11*, 6 (1999), 637–664.

[105] Lilius, J., and Paltor, I. vUML: A Tool for Verifying UML Models. In *ASE* (1999), pp. 255–258.

[106] Lindland, O. I., Sindre, G., and Sølvberg, A. Understanding Quality in Conceptual Modeling. *IEEE Software 11*, 2 (1994), 42–49.

[107] Manna, Z., and Pnueli, A. *The Temporal Logic of Reactive and Concurrent Systems - Specification.* Springer, 1992.

[108] Marc J. Balcer, C. B., and Epperson, D. That Action Language (plus an extra L). Tech. rep., 1989.

[109] McMillan, K. L. A Methodology for Hardware Verification Using Compositional Model Checking. *Sci. Comput. Program. 37*, 1-3 (2000), 279–309.

[110] Mellor, S. J., Scott, K., Uhl, A., and Weise, D. *MDA Distilled: Principles of Model-Driven Architecture*, vol. 88. Addison-Wesley, 2004.

[111] Mellor and Balcer. *Shlaer-Mellor Action Language*, 1997.

[112] Milicev, D. On the Semantics of Associations and Association Ends in UML. *IEEE Trans. Software Eng. 33*, 4 (2007), 238–251.

[113] Mohagheghi, P., Dehlen, V., and Neple, T. Definitions and approaches to model quality in model-based software development - A review of literature. *Information & Software Technology 51*, 12 (2009), 1646–1669.

[114] Moody, D. L. Metrics for Evaluating the Quality of Entity Relationship Models. In *ER* (1998), pp. 211–225.

[115] Moody, D. L. Theoretical and practical issues in evaluating the quality of conceptual models: current state and future directions. *Data Knowl. Eng. 55*, 3 (2005), 243–276.

[116] Muchnick, S. S. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[117] Myers, G. J., and Sandler, C. *The Art of Software Testing.* John Wiley & Sons, 2004.

[118] Nelson, H. J., and Monarchi, D. E. Ensuring the Quality of Conceptual Representations. *Software Quality Journal 15*, 2 (2007), 213–233.

[119] Nelson, H. J., Poels, G., Genero, M., and Piattini, M. A Conceptual Modeling Quality Framework. *Software Quality Journal 20*, 1 (2012), 201–228.

[120] Nielson, F., Nielson, H. R., and Hankin, C. *Principles of Program Analysis.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[121] Nugroho, A., and Chaudron, M. R. V. Evaluating the Impact of UML Modeling on Software Quality: An Industrial Case Study. In *MoDELS* (2009), pp. 181–195.

[122] Object Management Group (OMG). *Query/View/Transformation (QVT), version 1.0*, April 2008.

[123] Object Management Group (OMG). *Object Constraint Language Specification (OCL), version 2.0*, February 2010.

[124] Object Management Group (OMG). *Concrete Syntax for UML Action Language (Action Language for Foundational UML - ALF), Beta 2*, December 2011.

[125] Object Management Group (OMG). *Semantics of a Foundational Subset for Executable UML Models (FUML), version 1.1*, February 2011.

[126] Object Management Group (OMG). *Unified Modeling Language (UML) Superstructure Specification, version 2.4.1*, August 2011.

[127] Olivé, A. Conceptual Schema-Centric Development: A Grand Challenge for Information Systems Research. In *CAiSE* (2005), pp. 1–15.

[128] Olivé, A. *Conceptual Modeling of Information Systems.* Springer-Verlag, 2007.

[129] Paltor, I., and Lilius, J. Formalising UML State Machines for Model Checking. In *UML* (1999), pp. 430–445.

[130] Papyrus. *http://www.papyrusuml.org (last visit: February 2013).*

[131] Peffers, K., Tuunanen, T., Rothenberger, M. A., and Chatterjee, S. A Design Science Research Methodology for Information Systems Research. *J. of Management Information Systems 24*, 3 (2008), 45–77.

[132] Perseil, I. ALF formal. *ISSE 7*, 4 (2011), 325–326.

[133] Pilskalns, O., Andrews, A. A., Knight, A., Ghosh, S., and France, R. B. Testing UML Designs. *Information & Software Technology 49*, 8 (2007), 892–912.

[134] Pipino, L., Lee, Y. W., and Wang, R. Y. Data Quality Assessment. *Commun. ACM 45*, 4 (2002), 211–218.

[135] Planas, E. A Framework for Verifying UML Behavioral Models. In *CAiSE Doctoral Consortium* (2009).

[136] Planas, E., Cabot, J., and Gómez, C. Verificación de la Ejecutabilidad de Operaciones definidas con Action Semantics. In *Talleres de las Jornadas de Ingeniería del Software y Bases de Datos (DSDM - JISBD)* (2008), pp. 62–71.

[137] Planas, E., Cabot, J., and Gómez, C. Verifying Action Semantics Specifications in UML Behavioral Models. In *CAiSE* (2009), pp. 125–140.

[138] Planas, E., Cabot, J., and Gómez, C. Verifying Action Semantics Specifications in UML Behavioral Models (Extended Version). Available from: http://www.lsi.upc.edu/dept/techreps/llistat_detallat.php?id=1044.

[139] Planas, E., Cabot, J., and Gómez, C. Lightweight Verification of Executable Models. In *ER* (2011), pp. 467–475.

[140] Planas, E., Cabot, J., and Gómez, C. Two Basic Correctness Properties for ATL Transformations: Executability and Coverage. In *MtATL* (2011), pp. 1–9.

[141] Planas, E., Cabot, J., Gómez, C., Guerra, E., and de Lara, J. Lightweight Executability Analysis of Graph Transformation Rules. In *VL/HCC* (2010), pp. 127–130.

[142] Planas, E., and Sanchez-Mendoza, D. Alf-verifier: A lightweight tool for verifying UML-Alf executable models. `http://code.google.com/a/eclipselabs.org/p/alf-verifier/`, 2012.

[143] Planas, E., Sanchez-Mendoza, D., Cabot, J., and Gómez, C. Alf-verifier: An Eclipse Plugin for Verifying Alf/UML Executable Models. In *ER Workshops* (2012), pp. 378–382.

[144] Plat, N., van Katwijk, J., and Toetenel, H. Application and Benefits of Formal Methods in Software Development. *Softw. Eng. J. 7*, 5 (Sept. 1992), 335–346.

[145] Porter, A., Siy, H., Toman, C. A., and Votta, L. G. An Experiment to Assess the Cost-Benefits of Code Inspections in Large Scale Software Development. In *Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering* (New York, NY, USA, 1995), SIGSOFT '95, ACM, pp. 92–103.

[146] Porter, A. A., and Votta, L. G. Comparing Detection Methods For Software Requirements Inspections: A Replication Using Professional Subjects. *Empirical Software Engineering 3*, 4 (1998), 355–379.

[147] Project Technology Inc. *BridgePoint Object Action Language Reference Manual, Mentor Graphics*, 2009.

[148] Queralt, A., and Teniente, E. Reasoning on UML Class Diagrams with OCL Constraints. In *ER* (2006), pp. 497–512.

[149] Queralt, A., and Teniente, E. Reasoning on UML Conceptual Schemas with Operations. In *CAiSE* (2009), pp. 47–62.

[150] Queralt, A., and Teniente, E. Verification and Validation of UML Conceptual Schemas with OCL Constraints. *ACM Trans. Softw. Eng. Methodol. 21*, 2 (2012), 13.

[151] Riehle, D., Fraleigh, S., Bucka-Lassen, D., and Omorogbe, N. The Architecture of a UML Virtual Machine. In *OOPSLA* (2001), pp. 327–341.

[152] Rivera, J. E., Guerra, E., de Lara, J., and Vallecillo, A. Analyzing Rule-Based Behavioral Semantics of Visual Modeling Languages with Maude. In *SLE* (2008), pp. 54–73.

[153] Rozenberg, G., Ed. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations* (1997), World Scientific.

[154] Rumbaugh, J., Jacobson, I., and Booch, G. *The Unified Modeling Language Reference Manual*, 2. ed. Addison-Wesley, Boston, MA, 2005.

[155] Sage, A., and Rouse, W. *Handbook of Systems Engineering and Management.* Wiley Series in Systems Engineering and Management. John Wiley & Sons, 2009.

[156] Saiedian, H. An Invitation to Formal Methods. *Computer 29*, 4 (Apr. 1996), 16–17.

[157] Schäfer, T., Knapp, A., and Merz, S. Model Checking UML State Machines and Collaborations. *Electr. Notes Theor. Comput. Sci. 55*, 3 (2001), 357–369.

[158] Schneider, S. *Concurrent and Real Time Systems: The CSP Approach (Worldwide Series in Computer Science).* John Wiley & Sons, Sept. 1999.

[159] Schneider, S. *The B-Method: An Introduction.* Cornerstones of Computing. Palgrave Macmillan, Oct. 2001.

[160] Schrr, A. Specification of Graph Translators with Triple Graph Grammars. In *Proc. of the 20th Int. Workshop on Graph-Theoretic Concepts in Computer Science (WG '94), Herrsching (D* (1995), Springer.

[161] Seidewitz, E. Programming in UML: Why and How. 2nd Joint Eclipse / OMG Symposium.

[162] Selic, B. The Pragmatics of Model-Driven Development. *IEEE Softw. 20*, 5 (Sept. 2003), 19–25.

[163] Shull, F., Singer, J., and Sjøberg, D. I. *Guide to Advanced Empirical Software Engineering.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[164] Solheim, I., and Neple, T. Model Quality in the Context of Model-Driven Development. In *MDEIS* (2006), pp. 27–35.

[165] Spivey, J. M. *The Z Notation: a Reference Manual.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[166] Stamelos, I., Angelis, L., Oikonomou, A., and Bleris, G. L. Code Quality Analysis in Open Source Software Development. *Inf. Syst. J. 12*, 1 (2002), 43–60.

[167] Stephen J. Mellor, M. J. B. *Executable UML: A Foundation for Model-Driven Architecture.* Addison-Wesley, 2002.

[168] Thalheim, B. *Entity Relationship Modeling - Foundations of Database Technology.* Springer, 2000.

[169] Tort, A., Olivé, A., and Sancho, M.-R. An Approach to Test-Driven Development of Conceptual Schemas. *Data Knowl. Eng. 70*, 12 (2011), 1088–1111.

[170] Travassos, G. H., Santos, P. S. M. d., Mian, P. G., Neto, A. C. D., and Biolchini, J. An environment to support large scale experimentation in software engineering. In *Proceedings of the 13th IEEE International Conference on on Engineering of Complex Computer Systems* (Washington, DC, USA, 2008), ICECCS '08, IEEE Computer Society, pp. 193–202.

[171] Travassos, G. H., Shull, F., and Carver, J. Working with UML: A Software Design Process Based on Inspections for the Unified Modeling Language. *Advances in Computers 54* (2001), 35–98.

[172] Tsang, E. Foundations of Constraint Satisfaction, 1993.

[173] UML2Tools. *http://www.eclipse.org/modeling/mdt/?project=uml2tools (last visit: February 2013).*

[174] Unhelkar, B. *Process Quality Assurance for UML-Based Projects.* Addison-wesley Object Technology Series. Addison-Wesley, 2003.

[175] Unhelkar, B. *Verification and Validation for Quality of UML 2.0 Models.* Wiley Series in Systems Engineering and Management. John Wiley & Sons, 2005.

[176] Vallecillo, A., Gogolla, M., Burgueño, L., Wimmer, M., and Hamann, L. Formal Specification and Testing of Model Transformations. In *SFM* (2012), pp. 399–437.

[177] Valmari, A. The State Explosion Problem. In *Petri Nets* (1996), pp. 429–528.

[178] van Amstel, M., and van den Brand, M. G. J. Model Transformation Analysis: Staying Ahead of the Maintenance Nightmare. In *ICMT* (2011), pp. 108–122.

[179] Vieira, A., and Ramalho, F. A Static Analyzer for Model Transformations. In *Third International Workshop on Model Transformation with ATL (MtATL)* (2011).

[180] Wand, Y., and Wang, R. Y. Anchoring Data Quality Dimensions in Ontological Foundations. *Commun. ACM 39*, 11 (1996), 86–95.

[181] Wang, R. Y., and Strong, D. M. Beyond Accuracy: What Data Quality Means to Data Consumers. *J. of Management Information Systems 12*, 4 (1996), 5–33.

[182] Weber, F., Wunram, M., Kemp, J., Pudlatz, M., and Bredehorst, B. Standardisation in Knowledge Management – Towards a Common KM Framework in Europe, 2002.

[183] Wexelblat, R. *History of programming languages.* ACM monograph series. Academic Press, 1981.

[184] Wing, J. M. A Specifier's Introduction to Formal Methods. *IEEE Computer 23*, 9 (1990), 8–24.

[185] Xie, F., Levin, V., and Browne, J. C. Model Checking for an Executable Subset of UML. In *ASE* (2001), pp. 333–336.

[186] Xtext. *www.xtext.org/ (last visit: February 2013).*

[187] Yu, Y., Manolios, P., and Lamport, L. Model Checking TLA+ Specifications. In *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods* (London, UK, UK, 1999), CHARME '99, Springer-Verlag, pp. 54–66.