

VISUAL CONTRACTS (VCS) - ENRICHING GRAPHICAL SYSTEMIC MODELS TO SUPPORT DIAGRAMMATIC REASONING IN SYSTEM DESIGN

THÈSE N° 3972 (2007)

PRÉSENTÉE LE 7 DÉCEMBRE 2007

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE DE MODÉLISATION SYSTÉMIQUE
SECTION DES SYSTÈMES DE COMMUNICATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

José Diego de la CRUZ GARCIA

Ingénieur en électronique et télécommunication de l'Université de Cauca, Colombie
et de nationalité colombienne

acceptée sur proposition du jury:

Prof. P. Thiran, président du jury
Prof. A. Wegmann, directeur de thèse
Prof. C. Atkinson, rapporteur
Prof. C. Petitpierre, rapporteur
Prof. Y. Pigneur, rapporteur



ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Lausanne, EPFL

2008

*Caminante, son tus huellas
el camino y nada más;
caminante, no hay camino,
se hace camino al andar.*
—Antonio Machado

*A todos aquellos que me enseñaron el camino correcto;
Pero también a aquellos que me mostraron
todo aquello que no debo hacer...*

Con su ejemplo, me han ayudado a forjar el camino

A la memoria de mi padre

Mamita, espero que estés orgullosa

Abstract

Model-Driven Engineering (MDE) harbors the promise of developing software-based systems with little or no coding. Instead of coding, it is envisioned that software engineers build models that are automatically translated into code. The modeling notation of choice for model-driven methods is the Unified Modeling Language (UML).

In UML, modelers are forced to separately model structure (class diagrams), behavior (activity, sequence diagrams), state (statecharts), and integrity constraints (OCL). This separation of models makes it difficult to understand the overall behavior of the resulting system.

We propose a visual modeling notation, called visual contracts, for system specifications, which incorporate the four aspects (i.e., behavioral, structural, state and constraints) in a single diagram. From a UML point of view, this is a combination of an activity diagram, a class diagram, a statechart, and OCL code. Proposing a unified, diagrammatic notation for contracts requires advances in the following three dimensions: visual, formal and system-centric.

We examine the current state of research in modeling notations for system specifications, in contracts and in visual notations. We then describe the main contribution of the thesis, the concept of Visual Contracts. Visual Contracts condense the four components of the specification –i.e. behavior, structure, state, and constraints—in a compact form. Visual Contracts contain all the elements required for a specification. Visual Contracts are based on set theory. They are formalized in Alloy. As this is a complementary approach to traditional, analytic specification techniques, we are able to express features that are difficult to express using notations such as UML.

We consider that visual contracts, as a complement to UML, can be one of the aspects that can help model-driven approaches to fulfill their promise.

Keywords:

systemic modeling, system specifications, contract modeling, visual specifications, UML, enterprise architecture, formal methods

Version abrégée

L'Ingénierie dirigée par les modèles (IDM) promet de transformer le développement de logiciels en s'affranchissant de la programmation. Dans la vision d'IDM la programmation est remplacée par la construction de modèles qui sont ensuite traduits automatiquement en programmes. La notation de choix pour IDM est le Unified Modeling Language (UML).

Avec UML les modélisateurs sont forcés de séparer la structure (diagramme de class), le comportement (diagramme d'activité ou de séquence), l'état (diagramme d'état) et les contraintes d'intégrité (OCL). Cette séparation rend difficile la compréhension du globale du system.

Nous présentons une notation graphique, appelé contrat visuel, pour la spécification de systèmes qui incorpore les quatre aspects susmentionnés (comportement, structure, état et contraintes) dans un seul diagramme. Du point de vue d'UML, ceci revient à fusionner le diagramme d'activité, le diagramme de classe, le diagramme d'état et le code OCL. Proposer une notation graphique unifiée requiert des avancées dans les aspects suivants : l'aspect graphique, l'aspect formel et l'aspect systémique.

Nous examinons l'état de la recherche dans les outils de modélisation pour les spécifications de systèmes, dans les contrats, et dans les notations graphiques. Nous décrivons ensuite la contribution principale de la thèse, le contrat visuel. Un contrat visuel condense les quatre composants de la spécification, le comportement, la structure, l'état et les contraintes dans une forme compacte. Un contrat visuel contient tous les éléments nécessaires pour une spécification. Les contrats visuels sont basés sur la théorie ensemble. Ils sont formalisés en Alloy. Etant complémentaires aux approches traditionnelles, analytiques pour la spécification de systèmes, nous sommes capables d'exprimer des aspects difficiles à exprimer avec des notations tels qu'UML.

Nous considérons que les contrats visuels, en tant que complément à UML, est un des aspects qui peuvent aider à réaliser la promesse d'IDM.

Mot-clés :

modélisation systémique, spécification des systèmes, modélisation de contrats, spécification visuelle, UML, architecture d'entreprise, méthodes formelles

Table of Contents

Table of Contents	i
Table of Figures	iv
Table of Tables	vii
Acknowledgements	ix
1 Introduction	1
1.1 The Problem	1
1.1.1 Requirements for the Modeling Notation	2
1.2 Visual Contracts	2
1.3 Contributions	3
1.4 Outline of this Thesis	4
PART I – Contracts in Software Engineering, Visual Modeling and Systemic Modeling	5
2 System Modeling in Software Engineering	7
2.1 Unified Modeling Language (UML)	7
2.1.1 Advantages	9
2.1.2 Limitations	10
2.2 Object-Process Method (OPM)	11
2.2.1 Advantages	12
2.2.2 Limitations	13
2.3 Limitations in Current System Modeling	13
2.3.1 Incompleteness & Uncertainty	13
2.3.2 Compartmentalization	14
2.3.3 System and IT-System do not Interact	14
2.3.4 Modeling of Change	15
2.3.5 Granularity	15
2.3.6 Context Modeling	15
2.3.7 Analysis and Validation Required	16
2.3.8 Cognitive Constraints	16
2.3.9 Declarative Approach vs. Imperative Approach	17
2.4 Summary	18
3 Software Contracts as Specification Artifacts	21
3.1 Terminology	21
3.1.1 System	21
3.1.2 Service	22
3.1.3 Contract	22
3.1.4 Interpretation of a Contract	22
3.1.5 Assertions: Preconditions and Postconditions	23
3.1.6 Assertions: Invariants	23
3.1.7 Configuration	24
3.1.8 Interface Contract	25
3.1.9 Usefulness of a Contract	25
3.2 What is in a contract?	26
3.2.1 Historical Remarks	26
3.2.2 Operation Schemas	27
3.2.3 Meyer’s “Design by Contract”, 1988	28
3.2.4 Wirfs-Brock et al, 1990	29
3.2.5 Fusion, 1994	30

3.2.6	Catalysis, 1999	31
3.2.7	Kobra, 2002	32
3.2.8	ANZAC, 2002	33
3.2.9	Diagrammatic Approaches to Contracts	34
3.3	Summary	35
4	Creating Visual Models	39
4.1	Problems for the Creation of a Notation for Modeling	39
4.1.1	Choosing What to Represent	39
4.1.2	Choosing How to Represent	41
4.2	Techniques for Reasoning with Diagrams	43
4.2.1	Visual Formalisms	44
4.2.2	Expressing Constraints Visually	44
4.2.3	Limitations	50
4.2.4	Discussion	54
PART II - Visual Contracts for System Modeling		57
5	Systemic Modeling of Systems	59
5.1	SEAM as a System Modeling Approach	59
5.1.1	Main Modeling Concepts	59
5.2	Modeling Heuristics	60
5.2.1	Definition of Context	60
5.2.2	System / Environment Complementarity	61
5.2.3	Behavior / State-Structure Complementarity	61
5.2.4	Whole / Composite Complementarity	63
5.2.5	System Identity = Myself	64
5.2.6	Discussion	65
5.3	Summary	65
6	Visual Contracts	67
6.1	Semantics of Visual Contracts	67
6.2	Primitives for Visual Contracts	68
6.2.1	Basic Elements	69
6.2.2	Behavioral Description	71
6.3	Visual Contract	73
6.3.1	Execution of Visual Contracts	74
6.4	Discussion	75
6.5	Summary	77
7	Set-Associations in Detail	79
7.1	Algebra for Sets Associations	79
7.1.1	Operations on Set-Associations	80
7.1.2	Set-Associations as Contexts of Existence	80
7.1.3	Algebra of Contexts of Existence	82
7.1.4	Set-Associations as Collections	83
7.2	Specifying Behaviors	84
7.2.1	Transactional vs. Non-Transactional	84
7.3	Design Heuristics and Patterns in Visual Contracts	85
7.4	Summary	87
8	The Language for Visual Contracts	89
8.1	Primitives for Building Visual Contracts	89
8.2	Rules and Guidelines for Building Visual Contracts	89
8.2.1	Basic Elements	89
8.2.2	Relational Elements	91

8.2.3	Behavioral Elements.....	92
8.2.4	Temporal Elements.....	92
8.3	Metamodel.....	92
8.3.1	Basic Elements.....	92
8.3.2	Relational Elements.....	93
8.3.3	Behavioral Elements.....	95
8.3.4	Temporal Elements.....	96
8.3.5	Complete Metamodel.....	98
8.4	Summary.....	98
9	Translating the Visual Contracts to Alloy.....	99
9.1	Translating the Visual Contracts.....	99
9.1.1	Alloy Specification Language.....	100
9.1.2	Representation of Information Objects.....	100
9.1.3	Representing Time Statically.....	102
9.1.4	Representing Time in Operations that Change Collection Members Only	102
9.1.5	Representing Time in Operations that Change the State of the Objects.....	103
9.1.6	Representing Time for Execution of Sequential VCs.....	106
9.1.7	The VCML Representation.....	106
9.1.8	Results.....	107
9.2	Summary.....	109
PART III – Praxis of Visual Contracts		111
10	Case Studies.....	113
10.1	Example: A User Login.....	114
10.2	Example: the Plane Boarding Control (PlaBoCo) System.....	116
10.2.1	Definition of Information Objects.....	116
10.2.2	Operations Init & CheckIn.....	117
10.2.3	Operation Board.....	118
10.3	VideoStore PORT(Point-Of-Rent Terminal).....	122
10.3.1	Successful Scenario.....	123
10.3.2	Non-Successful Scenarios.....	126
10.4	BookStore.....	129
10.5	Example: the Job Yellow Pages Website.....	132
10.6	Summary.....	134
PART IV – Closing Thoughts		135
11	Conclusions.....	137
11.1	Limitations of Visual Contracts.....	138
11.2	Future Work.....	138
Bibliographical References		141

Table of Figures

Figure 1. A partial UML specification for action Board for a system that controls the boarding of passengers to a plane.....	8
Figure 2. OPM diagram that represents a process that interacts with several objects	11
Figure 3. OPM final diagram with all information required for a complete system model. From [DORI, D. 2002a], page 430	12
Figure 4. The contract establishes the responsibilities of the elements participating in each interaction. There is always a client and a server for a given service; the service must be declared as available in the interface.....	25
Figure 5. Use of contracts as a validation tool of the system model.....	26
Figure 6. 3-D box that specifies the way a reservation can be done using a library IT system. It represents how users of the library can loan copies of books. From [Kent, S. and Gil, J.].....	35
Figure 7. Model vs. reality. Expanded version of figure 5, that makes explicit the impact of modeling language in the validation process.....	40
Figure 8. The connection between sentences and facts is provided by the semantics of the language. Adapted from [RUSSELL, S. and NORVIG, P. 1995], page 158.....	42
Figure 9. Example of a simple Petri Net.....	45
Figure 10. Static structure diagrams: UML class, E-R (Entity-Relationship)	46
Figure 11. Arrow diagrams integrate additional semantic information. The decorations add some constraints over the functions and relationships (the arrows). From [DISKIN, Z.] p. 6.....	48
Figure 12. Spider Diagrams, from [HOWSE, J., <i>et al</i> , STAPLETON, G., <i>et al</i> 2004]	49
Figure 13. Constraint Diagrams [GIL, J., <i>et al</i> 2001].....	50
Figure 14. a) Simple and symmetrical Venn diagram with four contours. b) The simple symmetrical Venn diagram of five contours. c) Adelaide, a symmetrical Venn diagram of seven contours [GIL, J.Y., <i>et al</i> 2000].....	50
Figure 15. Use of regions in order to map complex configurations. The syntax of regions (left) is simpler than semantics in Euler-Venn diagrams (right) [GIL, J.Y., <i>et al</i>], Page 126.....	51
Figure 16. Intersecting contexts in Venn-Euler Diagrams using regions [GIL, J.Y., <i>et al</i> 2000], Page 126.....	51
Figure 17. Node-link interpreted as a table. On the left, the relational approach, adopted by E-R and class diagrams, where the observer has the whole knowledge. On the right, the relativist, systemic approach we use.	52
Figure 18. UML object, activity, state, and class diagrams for (a) before and (b) after action op1. The effect of change performed by operation op1 is not evident.....	52
Figure 19. Static structure diagrams: UML object diagrams (2 snapshots).....	53
Figure 20. The objects in the model track the real models in reality, and trace one part of their properties and behavior.....	61
Figure 21. The action changes the state. This change can be seen as changes in the structure (cardinalities) and state of the objects themselves.....	62
Figure 22. Action A is an action as a whole, Actions A1, A2 and A3, plus the ordering constraints constitute the action “as a composite”	64
Figure 23. Lifecycle composition with actions.....	64
Figure 24. Symbol representing the identity of the system in SEAM	69
Figure 25. Representation of an Information Object or Property in SEAM.....	69
Figure 26. Representation of a stateful Information Object. State information appears in each of the attributes.....	69

Figure 27. UML object diagram for 4 passengers and the state diagram for the class Person	70
Figure 28. Representation of 4 passengers (a) UML-like model with instance identifiers, (b) SEAM-like representation using instance cardinalities + explicit IO state information, c) SEAM-like representation with state information implicit on the set-association	70
Figure 29. Action changes cardinality of set-associations. On the left side the initial (a) and final (b) conditions. In (c), SEAM notation for representing changes in cardinalities..	72
Figure 30. Action changes state of instances in set-associations.....	72
Figure 31. Action changes cardinality of both SAs for passenger lists. A transfer has been made, meaning also a change of state of the respective instances of IO Person.	73
Figure 32. Annotated version of Visual Contract for action Board. Please refer to chapter 9 for a complete description of this illustration.....	74
Figure 33. A partial UML specification for action Board, equivalent to figure 32.	76
Figure 34. Making explicit the state information of instances belonging to a set-association. The IO Person has two possible states. The two diagrams at the bottom illustrate the implicit and explicit use of state information in set-associations.....	87
Figure 35. Basic elements in the Visual Contract metamodel	93
Figure 36. Relational elements in the metamodel of Visual Contracts. First partial view. .	94
Figure 37. Relational elements in the Visual Contract metamodel. Second partial view....	95
Figure 38. Behavioral elements in the metamodel of Visual Contracts.....	96
Figure 39. Complete Visual Contract metamodel.....	97
Figure 40. Strategy of translation from Visual Contracts to Alloy.....	99
Figure 41. Visual Contract symbol and Alloy specification for an Information Object....	100
Figure 42. Extended notation for the Information Object, and Alloy equivalent for this extended notation. State information is included	101
Figure 43. SEAM notation for property definitions in the specification of a system.....	101
Figure 44. The Visual Contract of action <code>initAction</code> and the corresponding Alloy code ..	103
Figure 45. Precondition for action <code>actionC</code>	104
Figure 46. Post-condition for action <code>actionC</code>	104
Figure 47. Visual contract for action <code>actionC</code> . It illustrates the operators «select», «change», and «transfer».....	105
Figure 48. Mapping between model and the corresponding XML schema	107
Figure 49. Mapping between Visual Contract for action Board and the corresponding VCML output.	108
Figure 50. Results of simulating the action Board in the Alloy analyzer. At Time0 the passengers (Person0, Person1, and Person3) have already checked-in. At Time1 two of them (Person0, Person1) effectively embark on the plane. Capacity of the plane is 3	108
Figure 51. Logging action after a first failed attempt.....	114
Figure 52. Logging action after a second failed attempt	114
Figure 53. Logging action after a third, final failed attempt.....	115
Figure 54. Logging action finished after three attempts. A response is generated.	115
Figure 55. Visual Contract for failed logging action.....	115
Figure 56. Visual Contract for successful logging action.....	116
Figure 57. Aggregate Visual Contract for successful & failed logging action.....	116
Figure 58. SEAM notation for data definitions for the PlaBoCo	117
Figure 59. The SEAM contract of <code>Init</code> : the cardinality of the <code>passenger_List</code> SA changes	118
Figure 60. Precondition for action Board	119
Figure 61. Post condition of action Board.....	119
Figure 62. Visual Contract for action Board. It illustrates the «change» operator	120
Figure 63. SEAM notation for property definitions in the PlaBoCo specification.....	120

Figure 64. Complete translation example for action Init. From left to right: Visual Contract, VCML, and Alloy model. The border lines indicate the correspondence among the models of steps 2 and 3.	121
Figure 65. Visual Contract domain model.....	122
Figure 66. Composite Visual Contract for a successful loan made using the PORT system	123
Figure 67. Visual Contract for action Create_OneLoan.....	123
Figure 68. Visual Contract for operation Check_OneMember.....	124
Figure 69. Visual Contract for operation Check_MultipleVideo	125
Figure 70. Resulting Visual Contract for the set of operators Create_OneLoan, Check_OneMember, Check_MultipleVideo, and Commit	125
Figure 71. Composite Visual Contract for the PORT system, including error handling and compensation measures	126
Figure 72. Visual Contract for operation Cancel_Create_OneLoan.....	127
Figure 73. Visual Contract for action Cancel_Check_OneMember	127
Figure 74. Visual Contract for operation Cancel_Check_MultipleVideo	128
Figure 75. Visual Contract for the composite action of figures 64 to 66.....	128
Figure 76. Global Visual Contract for successful and unsuccessful scenarios of the PORT system. It was extracted from figures 70 and 75	129
Figure 77. SEAM notation for a sale operation.....	129
Figure 78. SEAM notation for a sale operation, expanded with the domain models for each working object.....	130
Figure 79. Informal Visual Contract for the operation saleAction. Ad hoc operators are used to link information objects that are exchanged.	130
Figure 80. Visual Contract for the operation saleAction. The transfer among the two systems is shown explicitly	131
Figure 81. Visual Contract for the operation saleAction. The changes are local and the transfer is done via connectors among the parameters that the systems exchange.....	131
Figure 82. Class diagram for JobYellowPages.....	132
Figure 83. First interpretation of class diagram in terms of Visual Contracts for JobYellowPages	133
Figure 84. Set-theoretical <i>ad hoc</i> interpretation of set-associations of figure 75	133
Figure 85. Correct modeling of structural model of JobYellowPages using the Visual Contracts notation.....	134

Table of Tables

Table 1. Classification of UML notations	9
Table 2. Comparison between UML and OPM.....	19
Table 3. Elements for description of contracts.....	36
Table 4. Comparison of the different approaches for contractual specification.....	37
Table 5. A classification of visual formalisms based on the aspect they illustrate	44
Table 6. Comparison of different notations and their relative adequacy to representing behavior and/or structure.....	54
Table 7. Comparison of the different approaches for contractual specification.....	56
Table 8. Featuring the visual perception dimension of Visual Contracts.....	76
Table 9. Logical operations of Set-Associations. Note that cardinality must be always zero or positive ($m \geq n$)	81
Table 10. Default behavior for deletion of set-associations that do not include actions. The red dotted arrows represent the several deletion processes, the guards are named after the highest context that is being deleted.....	82
Table 11. Default behavior for deletion of set-associations that include actions. The red dotted arrows represent the several deletion processes, the guards are named after the highest context that is being deleted.....	83
Table 12. SEAM notation elements required for Visual Contracts.....	90

Acknowledgements

I would like to express my sincere gratitude to all the people that contributed to the making of this work.

First of all, I want to express my appreciation to Mme. Catherine Vinckenbosch and Mme. Annette Jaccard for their support and their generosity. They made my stay in Switzerland possible, and also assisted me at the personal level and also for the funding of my research during all these years. I must also thank the professor Eduardo Sanchez, as he invited me to Switzerland and made all this adventure viable.

I want to express my everlasting gratitude to my advisor, the Professor Alain Wegmann. He gave me the wonderful opportunity to explore new ideas and new paradigms in the LAMS (Laboratory of Systemic Modeling). I really appreciated sharing with him ideas about systemic modeling and formal methods, and being able to propose solutions to the beautiful problems that his methodology aims to resolve in the field of Enterprise Architecture.

I want to thank Dr. Gil Regev, my mentor and my support at the LAMS. Gil knew how to convince me of finishing my Ph.D. I really appreciate all the time and projects we did together, and especially having the possibility to use his systemic approach for problem solving.

I want to thank my colleagues at the LAMS. I will miss the quality work, programming, discussions and joint papers with Lam-Son Lê, and the incredible and intense discussions with Irina Rychkova about semantics. When I started my research at LAMS, I had the opportunity to share with Andrey Naumenko, Pavel Balabko, Otto Preiss and Guy Genilloud; they are responsible for the foundations of the work that I have continued through my research. Thank you all for your contribution, your support and the generous exchange of ideas.

On the scientific side, I want to acknowledge the people that have contributed in different forms to my work. First of all, I would like to thank Professors Thomas Baar and Collin Atkinson for their continuous interest on my work. I would like to express my gratitude to the visitors that accepted to give part of their time to nurture my theory: the Professors Donald Gause, Robert France Dirk Beyer, and Bran Selic, and Mr. Henry Peyret. I thank the members of the jury, the Professors Claude Petitpierre and Yves Pigneur, for their time and advice that helped me refine the ideas of this dissertation.

In a more personal register, I would like to thank Dr. Claudio Bruschini, Hassina Bounif, Dr. Hector Restrepo, Walter Pineda, David Portabella, Dr. Fabio Porto, Dr. Eugenio Tamura, and Professor Olivier Jolliet for their support and advice, as well as to Professors Jean-Dominique Decotignie and Humbert Kirmann for their generosity; thanks to Lianick Houmngny and Loïc Schulé for their practical contribution to my thesis, and to Mauro Cherubini and Alessandro Fachini for their technical expertise. I want to express my gratitude to my colleagues of *Ensures Consulting*, who supported me with generosity for the end of my Ph.D.

I cannot say goodbye to the LAMS without remembering with affection its administrative body. Thanks to Angela Devenoge, Danielle Alvarez, and more especially to France Faille and to Holly Cogliati for their friendship and hard work. You make the work at LAMS a delicious moment.

All my colleagues at EPFL during all these years of post-graduate studies and research were a wonderful source of ideas. In addition to them, I must show my gratitude to my ex-

colleagues in industry and especially to the Telematics Research Group of Universidad del Cauca, who convinced me to quit Colombia for pursuing my dreams. The endless support of my friends Daniel Ospina, Diego Acosta, Isabel Martínez, Ivan Hernandez, Ferney Rojas, Carlos Plaza, Marta Montaña, Carlos and Luis Perdomo, has been essential during these years.

The friendship of Brice Tsakam-Sotche, Paul and Martha Ostos-Briceño, Simon Keller, Etienne and Janine Bueche, Luis and Josefa González, Angelina and Pietro Cireddu made our life in Switzerland a nice experience. My gratitude to the members of the Association ACIS, who are also my friends, and of the Association ColombiaVive, *mis lanzas*; they are just too numerous to cite here; we worked together to make other people know the real Colombian style: folklore, hard work and a positive way of thinking. *Gracias muchachos*.

I want to thank my family, in particular to my parents, Nelly García and José De la Cruz (†), who that taught me to be relentless and also to respect His rules of the game, at the same time that they nurtured my intellect. I also want to thank my brother Carlos Andrés and my sisters – Angélica y Sandra –, and their beautiful families, as well as to my family-in-law — Isolina, Freddy, Fabián, Aidé— because they gave me the moral support required to endure during these years. *Los quiero a todos*.

Finally, I want to say to *mis duras*, Claudia and Diana, thank you for your love and comprehension. You accepted to live this adventure with me, to rebuild our lives, and to explore new worlds. Thank you for your courageous decision, and for your support. *Las amo*.

Gracias Señor por darme la vida, y esta vida en particular
Thank you, Lord, for my life

1 Introduction

Etre, c'est agir
Leibniz

Model-Driven Engineering (MDE) [MELLOR, S.J., *et al* 2003] harbors the promise of developing software based systems with little or no coding. Instead of coding, it is envisioned that software engineers build models that are automatically translated into code. The modeling notation of choice for model-driven methods is the Unified Modeling Language (UML) [OMG 2003].

In UML, modelers are forced to separately model structure (class diagrams), behavior (activity, sequence diagrams), state (statecharts), and integrity constraints (OCL). This separation of models makes it difficult to understand the overall behavior of the resulting system. In this thesis, we present a visual notation for specifying system behavior via the description of the resulting changes (of the structure and state) and the related integrity constraints. From a UML point of view, our visual notation is a combination of an activity diagram, a class diagram, a statechart, and OCL code.

In software engineering, the concept of contracts is useful for specifying operations done by a system [MEYER, B. 1987, WIRFS-BROCK, R., *et al* 1990, COLEMAN, D., *et al* 1994, D'SOUZA, D.F. and CAMERON WILLS, A. 1998, EVANS, A., *et al* 1998, HECKEL, R., *et al* 2001, BOTTONI, P., *et al* 2001, ATKINSON, C., *et al* 2002, SENDALL, S. and STROHMEIER, A. 2002]. Creating a contract requires determining what the initial conditions are, what the expected resulting state of the system is, and what the externally visible behavior is. Contracts are very useful as specification artifacts that condense the structure, the behavior, the state and the constraints. Nonetheless, contracts in the Information System field traditionally address the needs only at the programming level. We consider that for model-driven approaches to fulfill their promise, contracts must be specified in the models.

1.1 The Problem

The models made in the early phases of a system lifecycle are characterized by their lack of precision. These models are mainly diagrams complemented by some textual specification. They capture the information about the system, are technology-free, but are difficult to validate and verify¹.

The current model approaches require that *a)* the models are simple enough, *b)* the models are extensible and can be refined, in order to introduce more information or specialize the system later in the process, and *c)* the models should be verifiable, in order to identify the inconsistencies before going further in the modeling process. In summary, the models must be simple but meaningful, and informal but with a strong mathematical basis; this is a very complicated scenario.

Traditionally, the structure of the system is specified using one set of diagrams and constraints; the behavior uses another set of diagrams and constraints; and the states of the system are specified using yet another set. Constraints are expressed in sentential (textual) form, and contracts are traditionally specified in a textual form, too.

Proposing a unified, diagrammatic notation for contracts requires advances in the following dimensions:

¹ The verification is the process of checking whether the system is correct or consistent. A consistent system model is one whose properties respect the constraints of the system.

The visual dimension

The visual dimension (diagrammatic reasoning) enables the communication of additional information, which is neither present nor evident in the more standard, textual models [BARWISE, J., *et al* 2002, SHIN, S.-J. 1995]. Nevertheless, visual models are interpreted only as visual aids, not as practical, fundamental modeling artifacts [WARE, C. 2004]. Prior work has established the advantages of creating visual notations that have a formal basis [PEZZÈ, M., *et al* 2000].

The formal methods dimension

By adding an underlying formal semantics to visual notations, we can improve the comprehension of system behavior. The formal semantics guarantees that the system model can be analyzed using logic and mathematics. As a consequence, we have to build visual system models that have formal semantics.

Formalizing a visual notation, however, is not a simple task. Each element of the visual notation must then correspond to a mathematical construction that can be analyzed by a tool. In other words, a modeler can verify that the system model possesses a set of desired properties and avoids a set of non-desired ones. This activity is known as “reasoning” about the system [GLASGOW, J., *et al*].

System-centric dimension

In our approach, it is fundamental to establish the frontier of a system, and therefore to show how the system interacts with other systems. Moreover, it is also important to create models that make explicit that IT systems track the elements of the real world, in order to design the solutions that permit this tracking to take place. In most of approaches, it is normally implicit that the elements are at the same level of realization, as this simplifies the models. In addition, we also seek to make the hierarchy of systems explicit. Most approaches consider composition and refinement. We aim to complement these techniques with the fractal modeling both of systems and of viewpoints from the inside and the outside of a specific system.

1.1.1 Requirements for the Modeling Notation

From the above discussion, we conclude that in order to support the reasoning level required by model-driven approaches, the modeling notation used for the system specification should be:

- Graphical / Visual / Diagrammatic
- Compatible with formal methods
- System-centric

This notation might be used in order to create models of the system. Consequently, by creating these models and refining them, the modeler should be able to ascertain if the properties of the model correspond to the desired properties.

1.2 Visual Contracts

The definition of contract apparently fits the problem defined above. First, the contract is a concept that is understood naturally for expressing the constraints that should be respected by a correct interaction among actors. Second, it can express functional and non-functional aspects. Larman [LARMAN, C. 1997] proposed the use contracts as specification artifacts. RM-ODP [ISO/IEC 1996] also includes the concept of contract as a generic means to describe the role of each part in an interaction.

The value of contracts as specification artifacts is two-fold:

- It makes explicit many of the design decisions that took place for an element to be in the model. The statement of purpose for each piece of the model has as a

by-product “reasons” to include a certain module in the solution. Examples of unreliability caused by “additional, however harmless” software components include [BOWEN, J. 1996, JÉZÉQUEL, J.-M. and MEYER, B. 1997]. Furthermore, the specification artifact should be self-explainable.

- It can help reduce the cost of the development of IT systems. Economies of scale can be attained if the validation can be done as early as possible. This issue has been addressed by the model-driven approaches. Until now, however, it has not been possible to create models that are abstract enough and whose return on investment is high.

The specific goal of our work is to create a visual modeling notation for system specifications, which incorporates the four aspects (i.e., behavioral, structural, state and constraints) in a single diagram. As it is a complementary approach to the traditional, analytic one, we should be able to express features that are difficult to express using notations such as UML.

Creating a visual modeling notation with the use of contracts means that:

- Our main description unit is the action or service performed by a system. Consequently, our specification artifact is functional.
- We are able to describe the initial configuration of the system before the action is performed, as well as the final configuration of the system.
- We are capable of describing the constraints that must be applied for this action to take place.

Our work is partially based on the theories developed by our research group (SEAM – Systemic Enterprise Architecture Methodology) [WEGMANN, A. 2003], and its systemic ancestors, the General Systems Theory [WEINBERG, G. 2001] and the Living Systems Theory [MILLER, J.G. 1995]. SEAM has the necessary infrastructure for modeling behavior and structure. In this thesis, we expand on this and add the constraints.

1.3 Contributions

Visual Contracts condense the four components of the specification –i.e. structure, behavior, state, and constraints—in a compact form. As they contain all the elements required for a specification and are based on the set theory, Visual Contracts can be translated to formal methods notations in order to be verified and, eventually, be validated [LAPLANTE, P.A., *et al* 2001]. Visual Contracts can also be used as a complement to other modeling approaches such as UML or OPM.

Furthermore, modeling the behavior of a system as a transformation of its structure and state together –in the space delimited by the constraints affecting these transformations— results in the explicit modeling of change (see chapter 5). Change is a basic aspect of systems, but it is often difficult to understand and therefore to model. Modeling change in a compact, non-mathematical but nevertheless explicit manner is a contribution to systems engineering at large.

Finally, a fundamental issue is that modeling and specification techniques normally address single-instance problems. However, many types of systems –including Information Systems— are generally characterized by large data structures and multiple instances. Then, it would seem that it is essential to extend the capabilities of current modeling techniques in order to cope with multiple instances. One could argue that by mathematical induction, the modelers should be able to understand what happens in multiple cases. However, this has not been demonstrated. Furthermore, the techniques based on set theory cannot easily map to the

description techniques that deal with single instances. In this thesis, we propose a method for diagrammatically modeling and reasoning about single and multiple instances (see chapters 6 and 7).

The contributions of this doctoral dissertation regarding SEAM are:

- Better understanding of relationships as description elements. This complements the works of Genilloud [WEGMANN, A. and GENILLOUD, G. 2000] and Balabko [BALABKO, P. 2005].
- Use of relationships as dynamic entities, in supplement to Balabko [BALABKO, P. 2005] and Regev [REGEV, G. 2003].
- Comprehension of the quality-building attributes of relationships, continuing the works presented by Preiss [PREISS, O. 2004] and Regev [REGEV, G. 2003].
- Some contributions to the ontology of SEAM, in complement to the works by Naumenko [NAUMENKO, A. 2002] and Lê [LE, L.S., *et al* 2005].
- Finally, and more importantly, this work develops some insights into the behavioral semantics, a subject that has also been studied mainly by Balako [BALABKO, P. 2005] and partially by Naumenko [NAUMENKO, A. 2002]. Currently, the operational semantics is developed by the doctoral work of Rychkova [WEGMANN, A., *et al*].

1.4 Outline of this Thesis

The structure of this thesis is as follows: In Part I, we study the current state of contracts as specification artifacts and how to use them effectively in the analysis phase. In Part II, we describe the main design heuristics and introduce the reader to the Visual Contracts. In Part III, we validate our approach. In Part IV, we discuss the future work and present the conclusions of this thesis.

PART I – Contracts in Software Engineering, Visual Modeling and Systemic Modeling

In this part, we establish the foundations of our research work.

In chapter 2, we make an analysis of the problems present in current approaches to system modeling. We study the requirements for specification languages, we analyze how two system-modeling languages satisfy these requirements and we identify the aspects that could be enhanced by an alternative specification language.

In chapter 3, we study the various definitions of software contracts, discuss the practical consequences of the use of contracts by system modelers, and we synthesize a definition of contract that can be used as a specification artifact.

In chapter 4, we study the problems inherent to creating specification models that use visual or diagrammatic notations. As a result, we identify some elements that can be used to improve the expressive capacity of our notation for system specification.

2 System Modeling in Software Engineering

Describing a system is a complex task and therefore it requires tools and techniques to support the reasoning. In the case of Software Engineering, most modern approaches foster the use of concrete models instead of the use of implementations². These models are mostly diagrams that should be combined in order to form a complete description.

The combination of models is required because most notations support only very specific aspects of the description. This means that each model is a specific, specialized and concrete model thus easier to deal with. As a consequence, there is the need to create various, complementary models in order to have a more complete view of the system. In this way, modern approaches aim to fulfill the conventional software doctrine, where the software specifications must be [SHAW, M. 1996]:

- Sufficient and complete (say everything a user needs to know),
- Homogeneous (written in a single notation).
- Static (written once and frozen),
- Extensible.

In this chapter, we study UML, the one notation that has established itself as the reference in this domain. UML is considered as the best-breed of notations, and therefore its force is inescapable. Most works in the object-oriented domain, including those we study in chapter 3, can be considered as part of UML³.

Next, we study another notation and method that considers a different epistemological approach to modeling: OPM. It is based on systemic principles, and the models are holistic.

By studying models from these two different approaches, we can assess how the attitude towards system modeling has an effect on the quality of modeling of an IT system.

Finally, we characterize the weak zones of these approaches for creating a contractual specification artifact. We analyze in particular how the notations take into account the complexity of the final description⁴. From this analysis, the result is a series of points that we take into account for the design of our own visual notation.

2.1 Unified Modeling Language (UML)

UML [OMG 2003] is the *de facto* standard of the industry in object-oriented analysis and design. UML is one of the most popular graphical notations for modeling software (UML takes its roots in software development), systems [OMG 2005b] and businesses [ERIKSSON, H.-E., *et al* 2000]⁵.

UML consists of a set of diagrams that can be categorized into structure-related diagrams (part I of [OMG 2003]) and behavior-related diagrams (part II of [OMG 2003])⁶.

² Models are abstract but they “mimic” the implementation. Therefore, they are not concrete in the sense of materialization rather in the sense of conceptualization. As a corollary, they are abstract when compared to implementation, but very concrete when compared to some mathematical models.

³ For the sake of space, the use of UML is not illustrated here but extensive literature exists on the subject.

⁴ Nevertheless, we have to acknowledge that all modern notations take into account the complexity aspects and the integration needs, and such best practices are integrated at the method level. In this comparison we focus on the final, deep conceptual techniques for both of these approaches.

⁵ SysML, a UML version for system modeling is intended for generic system modeling. However, in this report we will discuss UML only because it is more mature and it is the basis for the SysML effort.

⁶ The use case diagram is an exception as it describes structure and behavior, however, the use case diagram has some limitations (e.g. diagram centered on one system) which limits its use as a general purpose diagram [WEGMANN, A. and GENILLOU, G. 2000].

UML was built by aggregating existing notations. However, harmonization among those notations is still an issue [DINH-TRONG, T.T., *et al* 2006], which has profound consequences as we explain next. The models are considered flexible because they are loosely coupled and UML notations are mostly informal and visual. Therefore, no automatic inter-model consistency checking can be made, even for simple systems.

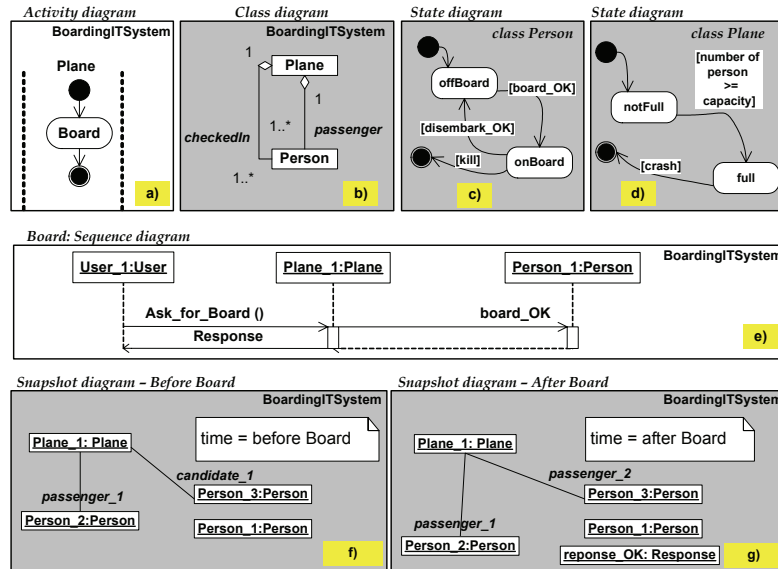


Figure 1. A partial UML specification for action `Board` for a system that controls the boarding of passengers to a plane

UML is a notation complemented by a rich set of methodological approaches that propose ways to model systems. The UML community affirms that a) there is a clear separation among notation and method, and b) that UML is only a notation and not a methodology. However, the isolation between the notation and the method has never been complete because many of the ontological and epistemological principles are implicit; as a result, the way their models can be created and what they express is dependent on the notation –as we discuss in chapter 4—.

Ad hoc methods for checking the consistency of the UML system specifications are very complex. Besides, they are limited to a given set of diagrams at a time. These methods require special techniques and tools, as well as the satisfaction of strict rules that guarantee that a full-fledged model (near implementation) exists prior to consistency checking [RICHTERS, M. and GOGOLLA, M. 1998, WARMER, J. and KLEPPE, A. 1999, LANO, K., *et al* 1999, BRUEL, J.-M., *et al* 2000, SENDALL, S. 2002]. Therefore, they cannot be used for early analysis or design, when knowledge about the system is still incomplete.

Table 1. Classification of UML notations

Description		Constraints	
Structure	Behavior	Structure	Behavior
class, (diagrammatic)	activity, (diagrammatic)	OCL (sentential)	OCL (sentential)
object, (diagrammatic)	communication, (diagrammatic)	OCL (sentential)	OCL (sentential)
component, (diagrammatic)	interaction overview, (diagrammatic)	OCL (sentential)	OCL (sentential)
composite structure (diagrammatic)	sequence, (diagrammatic)	OCL (sentential)	OCL (sentential)
package, (diagrammatic)	state machine, (diagrammatic)	OCL (sentential)	OCL (sentential)
Deployment (diagrammatic)	timing, (diagrammatic)	OCL (sentential)	OCL (sentential)
	use case (diagrammatic)		N.A.

As can be seen in table 1, OCL is essential for explaining the interdependencies among the various behavioral diagrams, among the various structural diagrams and also among the behavioral and the structural diagrams. Nevertheless, OCL is sentential whereas the other models are diagrammatic.

As discussed in chapter 3, several UML-based approaches support the notion of contract [WIRFS-BROCK, R., *et al* 1990, MEYER, B. 1992, COLEMAN, D., *et al* 1994, D'SOUZA, D.F. and CAMERON WILLS, A. 1998, ATKINSON, C., *et al* 2002]. Some researchers have developed schemas and contracts for UML using OCL [EVANS, A., *et al* 1998, HECKEL, R., *et al* 2001, BOTTONI, P., *et al* 2001, SENDALL, S. and STROHMEIER, A. 2002]. However, this notion of contract is still implementation-oriented, as it deals with the application of the principles of substitution for reuse of classes of objects, and assist with the composition problem of component-oriented design.

More recently, [LOHMANN, M., *et al* 2005] has proposed a graph-transformation technique that enables the automatic generation of code in order to reinforce the compliance to the system constraints (written in OCL). The resulting code ensures that the applications of the IT system obey the constraints and business rules in execution time.

2.1.1 Advantages

The UML system specification is made up of a set of seven diagrams and one OCL description. This guarantees the richness of the resulting specification, and leads us to suppose that it covers all the aspects of the system specification. As a consequence, the work can be divided among specialists of each kind of diagram.

- Imperative specification in the form of behavioral diagrams + OCL
- Declarative specification in the form of structural diagrams + OCL

Each diagram represents a different concern of the system. This makes the individual diagrams more legible than if they were merged together. The OCL description is used to add contextual information as discussed above.

Certain integrated approaches [BRUEL, J.-M., *et al*] combine the UML informal notations with formal notations; the informal notations allow for writing incomplete specifications. The integrated approaches consider not only the translation of each model but also the logic that ties the various model diagrams together [BRUEL, J.-M. 1998]. Integrated approaches and *ad hoc* methods continue, however, to be mostly manual processes [BRUEL, J.-M., *et al*]. Several initiatives have been established by the UML community in order to deal with the “separation of concerns” required among analysis, design, and implementation phases (and their corresponding models). This multimodal approach may provide more insights and lead to better specifications of the systems [ARGAWAL, R. and SINHA 2003].

- Time, ordering, and multiplicity constraints are all well specified in UML, although sometimes this is scattered in several diagrams, patterns, and other artifacts.

- In addition to the advantages of this notation, there is an extensive research on the integration to UML of standard software engineering practices, such as patterns, workflow systems, and interface design practices, among many others.

2.1.2 Limitations

As this modeling process builds the complete system specification in a single diagram (where all the objects, states and processes are combined), it does not scale and the diagram is unreadable.

OCL is used as a complement to the diagrams but is not fully integrated with them. OCL adds yet one more artifact to interpret.

As a consequence, the overall specification cannot be easily understood as a whole. There are no visible links between the elements represented in the different diagrams. In most cases, it is the modeler who “glues” the diagrams together within his or her mind [DORI, D. 2002b].

We could argue that this assumption is still valid when applied to UML nowadays: *“The semantics of this kind of functional description is dynamically non-committing in that it merely asserts that activities can be active, information can flow, and so on. It does not contain information about what will happen, when it will happen, or why it will happen”*[HAREL, D. 1992].

As we demonstrate in [DE LA CRUZ, J.D., *et al*], seven UML diagrams are required to model a single, partial scenario for a simple activity.

There is no consensus about the status and the use of OCL as the mechanism to specify actions and their effects. For instance, the use of OCL is not recommended nor encouraged by Kobra, given the difficulties they found: *“Describing the effects of operation invocations is not supported by the OCL, but should be describable using the Action Languages currently under development for UML”*[ATKINSON, C., *et al* 2002]. Among other works on Action Languages proposed by the UML community, we must mention QVT and the proposal for semantics of actions [MELLOR, S.J. 1999].

Even some researchers that work in the standardization committee of OCL propose to change its name because *“Object Constraint Language”* does not reflect the fact that it is used in a flexible form all over the diagrams⁷.

In what concerns the abstraction-level of the constructs, there are artifacts that are concept-oriented and others that are implementation-oriented⁸, but the latter are extremely strong in almost all specifications created with UML. For instance, a component is an implementation-oriented component, which somehow contradicts the efforts of Fusion, Catalysis and Kobra to use it as the basic building design element of models at all levels⁹.

⁷ [WARMER, J. AND KLEPPE, A. 1999] propose, for example, replacing “Object Constraint Language” with *“Expression Language”*.

⁸ As a matter of fact, UML provides five types of actions. These actions can be modeled by message passing, namely: *call*, *return*, *send*, *create*, *destroy*; unfortunately these primitives are **always** attached to the idea of a **method invocation**, which corresponds to a method of an object-oriented class but not to an action in business terms.

⁹ The complexity of the resulting models in model-based approaches is such that it gave birth to a counter-movement: the eXtreme Programming methods that aim to save time by dealing with the complexity directly at implementation level. Lately, a mixed approach has become very popular: the Agile Methods, where notations such as UML are used sparingly and partially, and a special project-management approach is required.

2.2 Object-Process Method (OPM)

The Object-Process Method (OPM) is a systemic approach to system modeling [DORI, D.]. This methodology of design is used to model systems and information systems. Unlike UML, it considers all aspects of a system in a single frame.

OPM considers a system as a hierarchical entity. It is thus able to cope with incomplete information on lower levels. This information can be added incrementally during the specification of the system. OPM provides a *graphical* and a *textual* version of the notation (**OPD** and **OPL**, respectively).

The notation elements of OPM are **entities** and **links**. An *entity* is a generalization of an **object**, a **process** or a **state**. The *objects* and *processes* are the basic elements of a system. Every *object* is in a *state*, and the *states* of an *object* change during the **execution of a process**.

OPM is mostly process-centric. It represents each *process* as an individual entity. This permits the creation of a unique model for the description of the structure and behavior of the system.

OPM explicitly adapts concepts from object-oriented modeling and focuses on describing the “**fundamental structural relations**”, where **exhibition/characterization** and **classification/instantiation** are taken into account, as well as the more standard relations (i.e., **aggregation/participation**, **generalization/specialization**).

The links connecting the entities can be built in a **structural** or **procedural** fashion. *Structural links* express static relations of couples of items: for example, the aggregation or the inheritance. The *procedural links* connect entities in order to describe the behavior of the system: for instance, how a process transforms, uses or is influenced by the other entities in the system. Then, the *procedural links* relate the *process* and the *objects* that are either: consumed by, obtained from, or affected (**effect**) by the *process*, and being input or output (**consumed** and **obtained**) as well as those required for a process to occur (**agent** when it is human, **instrument** when it is not human).

Boolean objects are an extension of the *procedural links* as they indicate what procedure is activated when a condition is true.

An object A that has attributes B and C is seen as an object A that “**exhibit features B and C**”. It is also possible to indicate what objects and states are modified by processes, as shown in figure 2. In this example the object ObjectA is an input, object ObjectC is an output of process ProcessP; the object ObjectC is affected by ProcessP, as its state changes from stateS1 to stateS2.

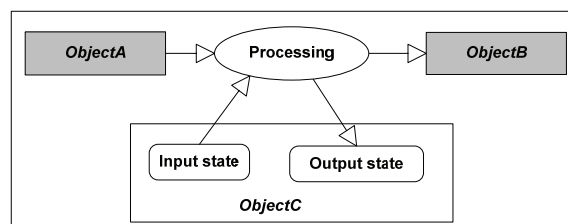


Figure 2. OPM diagram that represents a process that interacts with several objects

OPL is a textual language that covers completely the graphical counterpart, OPD. This language tries to be as compatible with natural language as possible. It could be classified as a form of *precise English*. This makes OPM multimodal, but not in the same way that UML conceives it.

As all the other forms of semi-formalized natural languages, OPM models tend to become voluminous for not-so-complex cases, whereas their precision is not guaranteed [EVANS, A. ET AL 1999]. But, the use of this form of language makes it user-friendly and enables direct

verification of the models. The modeler can then validate the translation of the graphical version (OPD) to a textual version (OPL).

2.2.1 Advantages

- OPM represents **physical objects** and **informational objects** as separate entities. He also represents *state inside* the *object/class declaration*.
- The complexity of OPM model is controlled via three techniques of refinement/abstraction: **zoom in/zoom out**, **unfolding/folding**, and **ellipsing/displaying** of state information. The *zoom in* shows how entities are composed, what elements are visible at each level of zoom. The *unfolding* shows the entities as the root of an oriented graph. The *ellipsing* occurs when a set of elements is not displayed based on *ad hoc* criteria. These techniques are supposed to enable the specifying and refining of the system under analysis in any level of detail without losing the comprehension of the whole system.
- The systemic side of OPM becomes evident when talking about **hard** and **soft attributes** and the *emergency of features*.
- The **final** diagrams include all the information about processes and objects that participate. They are a generic way to express everything that is required for the actions to happen, and what objects or features are affected. Figure 13 shows an example of final diagram.
- The *specialization* mechanism is powerful, as it is more explicit about the features that are taken into account or modified by the new sub-classes.

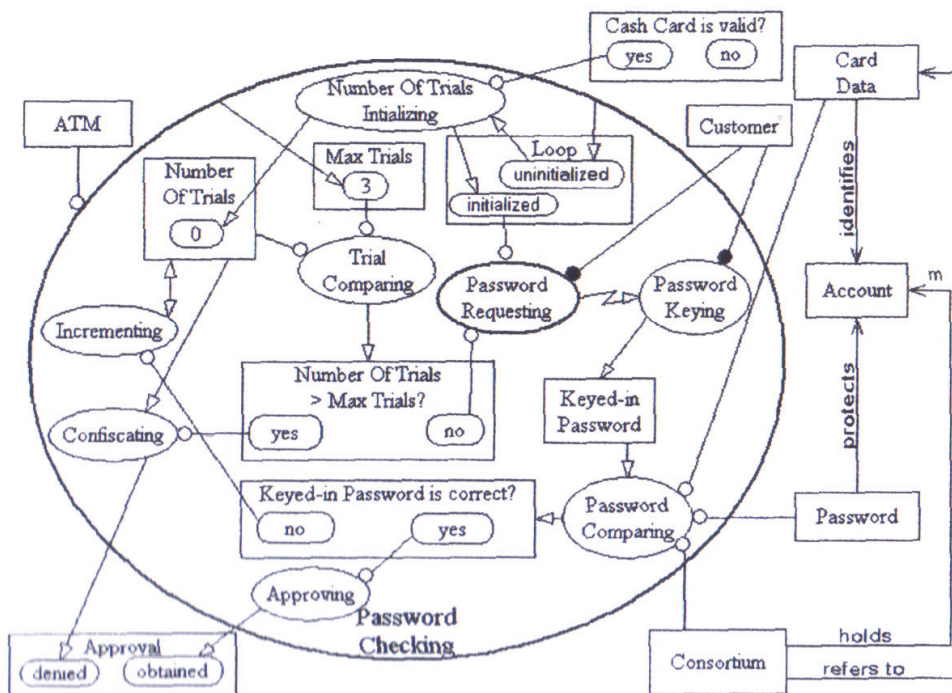


Figure 3. OPM final diagram with all information required for a complete system model. From [DORI, D. 2002a], page 430

2.2.2 Limitations

- As this modeling process represents elements from many different (UML) diagrams in a single diagram, this diagram contains all the objects, states and processes, and it does not scale as the diagram becomes unreadable.
- The quantity of information can be controlled via ellipsis.
- The instances are considered a sort of specialization of objects. This makes the qualification of instances very heavy (see figure 3).
- As a consequence, the constraints are minimal and no multiplicity can be put onto the diagram. Note that only single instances are modeled; there is no place where a number of instances (more than one) of any kind of object are explicitly processed. This can also be observed in figure 3, where a complete Automatic Teller Machine is described.
- OPM diagrams, especially high-level ones, do not indicate what actions are required in order to take place. The notion of causality is partially lost because of this missing aspect. Furthermore, the notion of time is mentioned in chapter 12.2 of [DORI, D. 2002a], but we could not find any further elaboration of the subject.
- There is no known support for formal verification. Even if the graphical system description can be translated onto OPDL, a textual, semi-structured English language, we do not know any formal analysis tool that can be applied directly to descriptions made in OPM. The descriptions in OPD can also be translated onto UML, but—as explained in section 2.1—UML cannot be analyzed.
- Reuse is not explicitly discussed.
- Finally, we may add that there is no clear integration of standard software engineering practices, such as patterns, workflow systems, interface design practices, among many others.

2.3 Limitations in Current System Modeling

In this section, we present the main points that we consider should be improved in order to create a modeling notation that are systemic.

2.3.1 Incompleteness & Uncertainty

System modeling is particularly difficult as knowledge about the universe of discourse is always incomplete. The current trend is to build total models, where everything is known, but this is impractical for most realistic scenarios. As [SWOBODA, N. and ALLWEIN, G. 2002] point out, there are objects with unknown states for certain attributes, and the modelers ignore objects that may be there but that do not participate in a given interaction¹⁰.

We could argue that completeness is not realistic in the context of information systems [RUSSELL, S. and NORVIG, P. 1995]. First, the models used in computer science are, in general, not as sophisticated as the models one may achieve using the basic model theory. Second, from the point of view of the formal specification of an information system, we must admit that the specifications are incomplete during the analysis phase and that the models should be able to deal with incomplete information [MONIN, J.-F. 2000].

In other words, the models cannot be complete because *a)* both complexity and methodological constraints, and *b)* the models must leave room for posterior changes in order to guarantee flexibility and adaptability. This means that the actions of the system must

¹⁰ Note that participation is a concept larger than an effective interaction. An object can participate in the form of a constraint for an action, even if it does not interact directly with other objects.

respect a certain number of constraints, expressed with logic formulae, but leaving room for a number of open options. These open options exist due to the incertitude and incomplete knowledge about the systems and their surrounding environments.

The concept of hierarchy can also be applied as a modeling heuristic. This permits us to isolate change and incompleteness. The notion of hierarchy is proposed by [PARNAS, D.L. 1972] and other researchers working on information hiding and modularization. [BROOKS, F. 1987, [WARD, P. 1985, YOURDON, E. 1988, [HATLEY, D., *et al* 1988] propose strict structural, top-down approaches to system modeling, as also—in the domain of visual notations— [HAREL, D. 1987] proposes a state-based hierarchical technique for modeling very complex behaviors.

2.3.2 Compartmentalization

From a systemic modeling point of view, current modeling techniques hide, unfortunately, the intimate relationship among behavior and state-structure:

- Behavior is defined in terms of **observable behavior**, in other words, the **events** that are communicated through the **interface** of the system, both **inputs** and **outputs**.
- *State* is defined in terms of the observable structural **changes**. The changes occur as the result of sequences of *events*. When the sequences stop, the changes stop. At that moment, the system is said to have a stable *structure* or to be in a particular *state*.
- *Structure* is considered as a “static” state: that which seems not to change to an observer on a given amount of time [WEINBERG, G. 2001].

As a consequence, behavior, state and structure are often modeled using separate models. In other words, this systemic symbiosis is not reflected in the way the models are created; behavior and state-structure are considered as isolated aspects of the specification, and therefore they require a separate set of both models and constraints.

We consider this to be a reductionist approach that denies the importance of interdependencies and constraints. Although this modeling principle minimizes the complexity on the construction of models and fosters specialization, it simultaneously limits the analysis of complexity of the model itself.

2.3.3 System and IT-System do not Interact

Current specifications methods often consider all the objects that are manipulated by the system and by the IT system are the same. Although the specification of the system and the specification of the IT system are models, this does not mean that whatever object is expressed in one of the models can be immediately used in the other.

More specifically, in most cases:

- The IT system is part of a larger system, and it performs a series of processes in interaction with the other parts of the same system.
- In order to achieve this interaction, the IT-system contains an interpretation of the world, and as such, it must track real-world entities and actions that are mapped in terms of its data structure and action structure, correspondingly.
- Therefore, the IT objects must be synchronized with the world objects.

We must find a solution to these three problems. This can be done by establishing a clear frontier—as well as the links— between the system and its environment, and also by applying the hierarchical approach to system modeling, as discussed in section 2.3.2.

2.3.4 Modeling of Change

In order to better understand what the system does, it should be clear how each system action changes the system itself and its environment. We argue that modelers do not consider modeling change because current methodologies do not provide mechanisms to combine explicitly the structural and behavioral components of the system. This was the intention of, for example, Use Cases [COCKBURN, A. 2001]; a Use Case is aimed to allow a quick understanding of the role of the system in its environment because it represents all the exchanges and changes due to their related actions; a Use Case is however sometimes so complex, that it does not really fulfill its objective.

By contrast, change modeling has been already addressed by formal approaches such as action semantics [MELLOR, S.J. ET AL, 1999, RUSSEL, S. AND NORVIG, P. 1995] and formal languages such as Z [SPIVEY, M. 1990]. It is therefore possible that modeling change would facilitate formalization.

2.3.5 Granularity¹¹

Current modeling techniques consider different models for each level of abstraction of the system. This means that different models are created for each phase and for each instantiation of the system. In general, the conceptual models are simple, whereas implementation models are extremely detailed and hard to follow. It is therefore difficult to align the models, and track features, specially because some terms mean different things for models in different levels of abstraction: a functional feature is understood as a use case at the conceptual level, but it may translate to a set of calls to methods of various classes at implementation level.

We should exploit the hierarchical nature of the system, instead of making large units of specification – as done in OPM—. In other words, new knowledge and contextual information should be introduced in each level of the hierarchy. This information can be encoded in the form of interactions and viewpoints. Interactions can be used to model how the systems interact, and viewpoints can be used to model how different observers perceive a system via the interactions.

2.3.6 Context Modeling

The choices made to classify the diagrams as either structure-related or behavior-related dramatically reduce the possibility to express the “interrelated conditions” in which model elements exist; more specifically, this choice in categorization reduces the capability to express context.

As explained in chapter 5, context modeling is essential. Context is defined in [MERRIAM-WEBSTER] as “*the set of interrelated conditions for one entity to exist*”, and this is what is pruned from the model when the information is scattered into many partial models.

The modeling of context has its limits on the system frontier. This means that this is more of an introspective view, and the interactions with other external systems is not taken into account.

¹¹ **Support for concurrency:** Complex systems are inherently multitasking and multi-user, and the notations should fully support this fact. Thread-safety is an issue in many domains, as well as the distributed nature of the system. Nonetheless, at the specification level these issues should somehow be addressed by the specification notation.

2.3.7 Analysis and Validation Required

It is essential to build models that allow for the system to satisfy the expected properties. This can only be done by verifying that the system is consistent, and by validating that the system does what it is entitled to do [LAPLANTE, P.A., *et al* 2001].

“Although the importance of testing and analyzing one-person algorithms has always been acknowledged, the world of complex systems has long suffered from something of an indifference to such needs... many past approaches to system development provided no means for capturing behavior, being centered instead on the functional aspects and dataflow. The approaches that did provide such means were informal, lacking the rigorous semantics necessary for even beginning to analyze the dynamics. Hence, it was impossible to predict in early states how the system would behave if constructed according to the model... As a consequence, most computerized tools that flourished around such methods...—CASE—...concentrated on providing mere graphic-edition capabilities, sometimes accompanied by... facilities. Their proponents heralded the ability of these tools to check model «consistency and completeness», which is really just a grand form of syntax checking”[HAREL, D.].

The actual trend for model verification is to validate the well-formedness of the instances (snapshots) of the system resulting from the execution of a scenario [ROYER, J-C 2004]. Lately there has been some work to create code that checks that the well-formedness constraints are respected by the code in execution time [LOHMANN, M. ET AL 2005]. The first approach is partial as not all models are taken into account and as generating snapshots is a cumbersome and work-intensive task that requires very complex tools for very simple systems; this makes this approach inappropriate for industrial practice. The last research line is more practical but requires full-fledged systems or a complex strategy to control what has to be checked during the development of the solution.

We want to work at the model level—as in the first approach—, but also have the possibility to introduce probes in order to **validate** and **verify** (V&V)—as in the second approach—. Given the level of abstraction, the flexibility for V&V and for making changes to the specification should be increased.

2.3.8 Cognitive Constraints

The capacity of the human brain to retain and process information is limited. This has been documented with studies such as [MILLER, G.A.]. This limitation raises several issues, especially for evaluating the adequacy of the use of many partial models vs. the use of less, more condensed models. As a matter of fact, this is a crucial cognitive frontier that constitutes the main criteria in the construction of our notation.

First, it is clear that diagrams express much more information than text [LARKIN, J. and SIMON, H.]. They are used in a number of domains to improve comprehension and reasoning.

Secondly, several studies demonstrate that practitioners do not really use notations presenting a high complexity. Instead of UML, practitioners use specific, *ad hoc* versions of it, which requires less modeling elements[CHERUBINI, M., *et al*]. Even in environments where people are relatively knowledgeable about UML, most practitioners specialize in only one or two kinds of UML diagrams [ARGAWAL, R. and SINHA, A.P.]. Some researchers criticize the use of rich notations such as those provided by UML, as this reduces its usability [DORI, D. 2002b]: its inherent complexity obfuscates human understanding. In other terms, this increases the amount of effort that the modeler invests in analyzing the information in the model.

In addition, some works demonstrate that UML notations are not necessarily the best notation to reason about each aspect of the system. For example, [IRANI, P.] demonstrates

that the Geo diagrams can replace class diagrams because they contribute to the memorization of structural components, and have a more appealing physical appearance that assists memorization. [SOWA, J.F. 1999] explains extensively several notations for knowledge representation, and how they support reasoning.

Only very recently, a number of works study the way UML is used by practitioners, and how the complexity of the language affects the way people use it.

2.3.9 Declarative Approach vs. Imperative Approach

[HAREL, D.] affirms that the special nature of reactive systems requires special specification artifacts that allows modelers to analyze systems early in the software life cycle (SLC). However, most information systems are as complex as the organizations they exist within, hence the economies of scale justify the existence of approaches that address the early validation.

In both approaches, one writes down a description of a problem or state of affairs, and then uses the definition of the language to derive new consequences. In the case of a program (imperative approach), the output is derived from the input and the program; in the case of a declarative approach, answers are derived from descriptions of problems and facts about the system.

What we need is a language that allows us to describe structure and behavior. As we see, in the case of UML notations, behavioral ones are imperative whereas structure-related ones are mostly declarative.

The first surprise was the so-called «predicate transformers» that I had chosen as my vehicle provided a means to directly defining a relation between initial and final state, without any reference to intermediate states as may occur during program execution.

I was very grateful for that, as it affords a clear separation between two of the programmer's major concerns: the mathematical correctness concerns (viz. whether the program defines the proper relation between initial and final state – and the predicate transformers give us a formal tool for that investigation without bringing computational processes into the picture) and the engineering concerns about efficiency (of which it is now clear that they are only defined in relation to an implementation)» [DIJKSTRA, E.W. 1976]

Because of the nature of the structure, it is rather difficult to think of an imperative language that describe the structure. However, it is possible to write declaratively the specification for behavior-related approaches. In summary, we should explore the declarative approach, in order to shed some new light on this domain¹².

“We might also be interested in reachability tests, which would determine whether—when started in some given initial situation—the system can ever reach a situation in which some specified condition becomes true. This condition can be made to reflect desired or undesired situations Moreover, we could imagine the test's being set up to report on the first scenario it finds that leads to the specified condition, or to report on all possible ones, producing the details of the scenarios themselves. We thus arrive at the idea of exhaustive executions” [Harel, D.]

¹² The imperative approach addresses calculability analysis, halting problems and other very interesting research issues. The SEAM methodology also addresses these issues. As the imperative approach is more related to operational semantics, it is partially covered by the work of Balabko [BALABKO, P. 2005], and currently complemented by Rychkova [WEGMANN, A., et al].

Nonetheless, in order to succeed in our effort, we must answer to the very basic practical question: *Is the required technology already there?* You can find the response to this in chapter 6 (for the representation) and 9 (for the demonstration).

2.4 Summary

Software specifications should satisfy the four criteria presented in the introduction. From the analysis briefly illustrated in this chapter, it is clear that the current state-of-the-art notations do not fully comply with those requirements.

The principle of separation of concerns [DE WIN, B., *et al* 2002] affirms that each aspect of the description must be treated individually. However, the interrelationships of the elements of the specification are partially lost in the process. For instance, the changes made by the system actions/services are made explicit by snapshots in UML; a class diagram, the state diagrams for the classes concerned, and an interaction diagram are strictly required for understanding the snapshots. But, to fully understand what happens, the modeler needs to write and read OCL in addition to all the diagrams. As a consequence, the task of understanding the goal the modeler wants to achieve with the action/service requires a large effort [DORI, D. 2002b]

The expressive capacity of OPM models strongly suggests that the systemic approach for system modeling is a promising alternative¹³. No additional sources of information are required. The semantics are not formal and the approach is not scalable but each action is described fully.

In table 2, we summarize some features of UML and OPM. In our systemic approach, we consider important three aspects shown in the table:

- The modeling of multiple instances and collections
- The formalization of the approach, and the corresponding support for validation and verification
- The explicit modeling of the system

It is clear from table 2 that these three aspects are not satisfied by these approaches. Hence, these aspects must be addressed by our specification artifact.

¹³ An interesting realization that resulted from this study is the systemic base of UML foundations even if UML itself is not systemic: Booch's presentation of system modeling is highly systemic[BOOCH, G. ET AL 1998]; Rumbaugh proposes to integrate aspects, such as state into structure [RUMBAUGH, J.R. ET AL 1991]; and Jacobson introduces Use Cases in order to have a better understanding of interactions, and a distribution of responsibilities [JACOBSON, I. ET AL 1993].

Table 2. Comparison between UML and OPM

	Notations / Diagrams	Representation of goals	Process/behavior of single instance	Process/behavior of multiple instances	Representation of State and Value	Representation of State Change	Local constraints	Global constraints	Formalization	Model Validation / Execution	Explicit System (Border)	Representation of Interactions
UML / Sys-ML	Multiple diagrams, written each in a different notation Some notations are hierarchical Complemented by OCL for pre-conditions and post conditions	Only in Use Case diagram Each Use Case diagram may be complemented by OCL	For all but class diagrams Each instance is drawn independently in: object, interaction, activity, state...	Static view on class diagrams Tacit support in activity diagrams, but often indicated with OCL expressions	State in state diagrams Tacit in interaction, activity and object diagrams Often indicated via OCL expressions Values depend on types and are supported, but no type checking is guaranteed	Shown only in state diagrams It can also be mentioned in other diagrams via OCL expressions or design comments It may apply to single instances only	Pre-conditions: can be added with OCL to action/activity diagrams Post conditions: can be added with OCL to action/activity diagrams Invariants: either visible in each notation, or can be added with OCL to action/activity diagrams No support for qualifiers except when text says something else	Pre-conditions: can be added with OCL to clusters of action/activity diagrams Post conditions: can be added with OCL to action/activity diagrams Invariants: visible in each notation No support for qualifiers	Consistency among models is not guaranteed at semantic level Many efforts, notation-by-notation Many efforts, notation + constraints in OCL	For several diagrams, because of work on each one of them For several clusters of diagrams OCL not very well integrated Via snapshots, mostly based on Graph Transformation (GT)	No Exchange barrier for message passing (with parameters) to/from actors (environment) Does not zoom out	In interaction diagrams Partially in state, class, and activity diagrams Can be indicated via OCL expressions
OPM	One diagram written in OPD One sentential representation written in OPL Graphical sentential representation are equivalent		Yes	Not explicitly supported	Both are supported Values are more abstract than states Support for value assignment and checking	Shown as consequences of processes It may apply to single instances only	Pre-conditions: symbols for required objects and required states or values Post conditions: symbol shows affected objects & operator shows the kind of change Invariants: mostly visual as structure in the diagram No support for qualifiers – single instances except when text says something else	Same as local constraints, because it is a hierarchical notation	Not known	Designed for correct design Animation is possible	No. System is implicit All participants are modeled as objects Everything makes part of system Does not zoom out explicitly	An operator to indicate participation An operator to indicate what object is affected by an action

3 Software Contracts as Specification Artifacts

This chapter introduces the notion of contract in the software engineering field. The goal is to understand the definitions, use and importance of software contracts

The contract-oriented approach splits the functionality into manageable chunks of behavior, which are described formally by using logics. Since its introduction in the 1990s, the contract has been a pervasive tool in the formalization effort that the software community launched. The term contract was coined by Bertrand Meyer¹⁴ in [MEYER, B. 1988] in the context of object-oriented programming, but the notion already existed several years before. Larman [LARMAN, C. 1997] and RM-ODP [ISO/IEC 1996] motivated us to do this research. Unlike the mainstream of contract-based approaches for computer programming, these two proposals use contracts as specification artifacts. Other proposals studied in this chapter address the use of contracts during the analysis and design phases of the software lifecycle (SLC).

In this chapter we proceed as follows: First, we build a terminology from the most popular kinds of contracts. More specifically, we adapt the original theory of contracts (mostly related to implementation) in order to deal with conceptual modeling. Next, we study the most well-known definitions of contracts, the problems they were intended to address, their differences, and how they deal with different theoretical issues that are essential for a good specification artifact. Finally, we synthesize a definition of contract that can be used effectively as a specification artifact in the context of our methodology, SEAM [WEGMANN, A. 2003].

3.1 Terminology

Contracts require the definition of a number of notions. These terms, explained below, are specific for the conceptualization or analysis phase.

3.1.1 System

In RM-ODP, a system is described as: *“Something of interest as a whole or comprised as parts. Therefore, a system may be referred to as an entity”* [ISO/IEC, et al 1998].

According to [WEINBERG, G. 1975] the **observer** is essential. The **frontier** of the system (i.e. the criteria defining the *“parts”* that are inside and those that are outside) is part of an interpretive view. As [REGEV, G. 2004] explains, a *“set of parts”* or system is not an absolute property in the world because the same set of parts can mean something totally different for another observer:

“We call observer the person making the judgment about which set is a system and what elements belong to this set. In this view, the set itself is an interpretation of the observer ... Hence a system is a set of interrelated elements representing an entity in the observed reality as defined by an observer.

The set that an observer defines as a system establishes the frontier that the observer identifies between system and environment. The set of elements and their

¹⁴ The decision of Mr. Meyer to trademark the expression “Design-by-Contract” was not accepted by many researchers. This is apparently one of the main reasons for the software development community to avoid using the term *contract* for many years.

relationships constitute the system. All other aspects of the reality of the observer she considers as being the environment of the system.” [REGEV, G. 2004]

In other words, an object is a system. Furthermore, we assume here that the system is decomposed hierarchically, in a fractal fashion¹⁵.

In accordance with SEAM (please refer to section 5.1), we define working objects (i.e. systems) and information objects (i.e. properties).

3.1.2 Service

In systems theory, a system is built or exists in order to satisfy a number of goals. These goals are described in the context of the system and its environment. Accordingly, we can affirm that a system can be described either by the actions it executes or by the services it provides to its environment.

Thus, we define a service as the functionality that can be delivered by a system. It is equivalent to an action, from the point of view of an external observer¹⁶.

3.1.3 Contract

The meaning of contract, after the Merriam-Webster Dictionary [MERRIAM-WEBSTER. 2005] is:

Main Entry: ¹**con·tract**

Function: *noun*

1 a : a binding agreement between two or more persons or parties; *especially* : one legally enforceable **b** : a business arrangement for the supply of goods or services at a fixed price <make parts on *contract*> **c** : the act of marriage or an agreement to marry

2 : a document describing the terms of a contract

As our purpose is to find a contract notion that is compatible with conceptual models, the contract concept in definitions *1.a* and *2* fulfill our needs.

The definition *1.b* deals more with Quality-of-Service (QoS) issues (e.g. performance, availability); This does not make part of our study. Neither do marriages, thus definition *1.c* is also discarded.

In summary, we consider a contract as a specification artifact that describes –definition *2*— an even enforces –definition *1.a*— the responsibilities of parties participating in an interaction or collaboration.

3.1.4 Interpretation of a Contract

The interpretation of the specification of a contract in programming is borrowed from the *theory of formal program validation*.

Meyer explains this thoroughly in [MEYER, B. 1992]. The main principle is the use of a correctness formula:

¹⁵ As a consequence, the terms **object**, **system** and **sub-system** will be used interchangeably in this report.

¹⁶ The original, implementation-oriented contracts refer to routines, operations, component operations and methods. In this document, service is a general term to refer to these (late) terms. The current discussion on “software as a service” (SaaS) is taking place nowadays and the use of the term **service** may be misleading to readers.

$$\{P\} A \{Q\} \tag{1}$$

that means

“Any execution of A , starting in a state where P holds, will terminate in a state where Q holds”,

being A one action or service, and P and Q two conditions or assertions. Each assertion (i.e. P and Q) may be composed of various assertions bounded together by logical connectors¹⁷.

Correctness formulae (also called *Hoare triples* [HOARE, C.A.R. 1969]) are a mathematical notation, not a programming construct; however, by the use of predicates (evaluation of properties, which are logically equivalent to assertions or conditions¹⁸), a meaning may be assigned to any piece of software, and its value of truth consequently computed.

Moreover, conditions are named according to their order of evaluation in the logical expression: P is called a **precondition**, Q is a **postcondition**; the subset of conditions that are common in P and Q are named **invariants**.

3.1.5 Assertions: Preconditions and Postconditions

Assertions are expressions involving some objects (the subjects), and stating properties that these entities may satisfy at certain stages of system evolution –via the execution of actions—. Mathematically, the closest notion is that of predicate and, syntactically, they are simply Boolean expressions with a few extensions. Two main types can be identified:

- **Precondition:** a predicate that characterizes certain conditions under which an action may be executed. It corresponds to P in equation 1.
- **Postcondition:** a predicate that must be true immediately following the execution of action. It corresponds to Q in equation 1.

The motivation for including pre- and postconditions was to enable reasoning about the correctness of the whole system action¹⁹. As we consider the specification as compositional, then the correctness of the whole action specification can be interpreted as the combination of the correctness of its parts.

It is also an abstraction mechanism that adds flexibility to the model process: **“To this end, pre- and postconditions offer a means to constrain the required behavior of an operation. Furthermore, if the assertions are strong enough, they express everything that the caller and the designer needs to know about the operation without disclosing how the operation is or should be designed/implemented.”** [SENDALL, S. 2002].

3.1.6 Assertions: Invariants

The invariants are predicates that must be true during the entire lifecycle of the object.

[MEYER, B.] explains the invariants in terms of social contracts as follows:

“Invariants have a clear interpretation in the contract metaphor. Human contracts often contain references to general clauses or regulations that apply to all contracts within a certain category; think of a city’s zoning regulations, which apply to all

¹⁷ The $\{P\} A \{Q\}$ notation as used here denotes *total correctness*, which includes termination as well as conformance to specification. (The property that a program will satisfy its specification if it terminates is known as *partial correctness*).

¹⁸ In this text, predicates, assertions and constraints are considered equivalent.

¹⁹ The precondition **False** is the strongest possible assertion, since it is never satisfied in any state. Any request to execute A will be incorrect, and the fault lies not with the system responsible for A but with the requester — the “client” of the service — since it did not observe the required precondition, for the good reason that it is impossible to observe it. In the same way, the postcondition **True** is the strongest possible one.

house-building contracts. Invariants play a similar role for software contracts: the invariant of a class affects all the contracts between a routine of the class and a client.”

The invariants have a larger scope than preconditions and postconditions. They are valid for the whole system and not only for a specific system action.

Invariants may be implemented by enriching the preconditions and postconditions of all actions in the system. The correctness expression of equation 1 can be modified for including invariants. Thus, it becomes:

$$\{INV \text{ and } Pre\} A \{INV \text{ and } Post\} \quad (2)$$

This means: **“any execution of action A, started in any state in which INV and Pre both hold, will terminate in a state in which both INV and Post hold.”**

Adding INV makes stronger both the precondition and the postcondition:

- In addition to the official precondition *Pre*, you may assume that the initial state satisfies *INV*, restricting even further the set of cases that you must handle.
- In addition to your official postcondition *Post*, you must ensure that the final state satisfies *INV*.

In fact, when the invariants are not isolated, repeated predicates appear in both extremes of the specification (pre- and post-) making both the routine’s purpose fuzzy and the specification clumsy. By isolating invariants, two portions of the specification are differentiated:

- Individual action specification (pre- and postconditions)
- Life-time system specification (invariant).

In software engineering, most invariants used in actual programs could be seen as the **safety conditions** (“*nothing bad can happen*”) while the other assertions are analogous to the **liveness conditions** (“*something good will eventually happen*”) [LAPLANTE, P.A. and (ED.) 2001]. And, as [LAMPOR, L., *et al* 1989] demonstrate, any specification can be written using safety and liveness conditions only.

In summary, the use of invariants is practical because they are conditions that must hold for the system, become a default part of the specification of each and every action²⁰.

3.1.7 Configuration

It is the expression of the global state of the system. A **configuration** is a snapshot: a global description of the objects that exist in the system at a given point in time.

As each object can be in a state, and has relationships with other objects, the configuration includes this information, too.

We define a correct system as one that behaves correctly, or more specifically, as *a*) a system whose configurations in time correspond to configurations that comply with the constraints of the system and with the description of service of the system towards its environment, and *b*) a system whose actions are able to transform initial configurations (pre-conditions) to final configurations (postconditions).

²⁰ [CRNKOVIC, I. 2002] define invariants as the states which must remain valid during and after the execution of the operation. However, this is not a precise definition given that –in spite of its name– the invariant does not need to be satisfied at all times. Indeed, at some intermediate stages the invariant will not hold; this is fine as long as the procedure reestablishes the invariant before terminating its execution. Some interesting research problems have their origin on this dichotomy. Further elucidation is given in section 3.2.6.1.

3.1.8 Interface Contract

An interface of a component can be defined as a specification of its access point
Szyperki – “Component Software”

Objects are structural modeling entities of the specification of a system. An object is a system. Each system (set of objects) provides a set of services. In addition, each one of the objects within this system provides a set of services. The services of an object are accessed via the interface of the object (a method in programming terms). The object that provides the service is known as the **server**, whereas the calling object is known as the **client**.

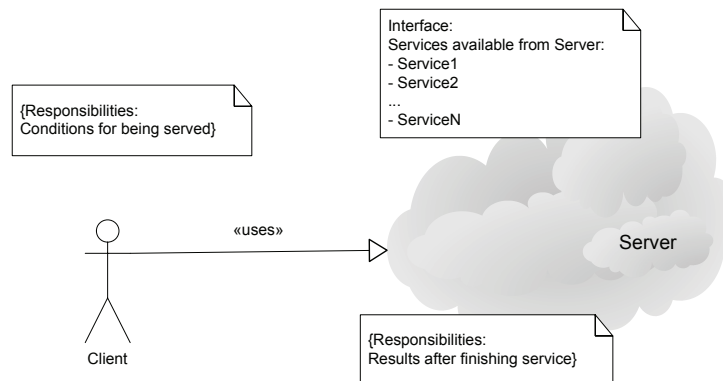


Figure 4. The contract establishes the responsibilities of the elements participating in each interaction. There is always a client and a server for a given service; the service must be declared as available in the interface

If the modeler wants to compose more complex services, she can specify interactions of the objects that orchestrate the use of services of other objects. In this case, each object plays a well-defined role and has a set of responsibilities.

Under these assumptions, the behavior of a system can be understood as the composition of the services provided by those objects (that result in a change of the state of the objects, and in consequence, of the global state of the system or *configuration*). Therefore, the resulting state (the **goal** of the system) is achieved through the interactions of the objects via their interfaces.

Therefore, in order to specify the behavior of the system, it is essential to describe how to interact with an object. In other words, the modeler should be able to specify how the objects interface to each other. We call this the **interface contract**. As we can see, interface contracts are implicit in object-oriented programming languages and methods.

3.1.9 Usefulness of a Contract

*If one implements, for instance, a programming language,
one will not prove that the implementation executes any correct program correctly;
one should be happy and content with the assertion that
no correct program will be processed incorrectly without warning.*
Dijkstra—“A discipline of programming”

The goal of design-by-contract is to improve the reliability of the software: a system’s ability to perform its job according to the specification (**correctness**) and to handle abnormal situations (**robustness**) [ISE, MEYER, B. 1992]. This correctness is a necessary condition to deploy reusable system/object specifications.

Therefore, from a neat and simple proposal, contracts grow into a systematic approach for specifying and implementing system elements and their relations within a system. More specifically, in software engineering, contracts constitute a framework for debugging, testing and even for quality assurance [MITCHELL, R., *et al* 2002].

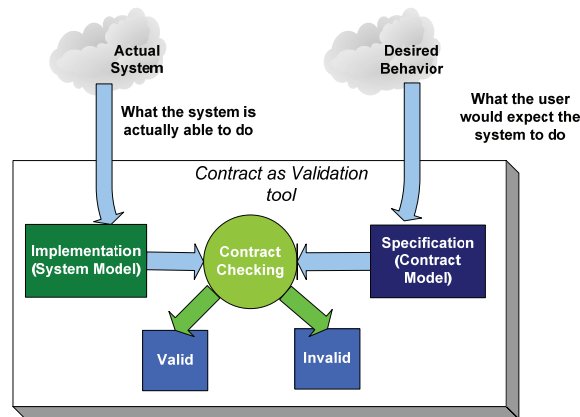


Figure 5. Use of contracts as a validation tool of the system model

Figure 5 shows the way to use the Contracts as part of a validation workbench. The **system model** corresponds to the implementation and is generally created using specification languages (usually *imperative*-style) whereas the **contract model** corresponds to the specification (generally of *declarative*-type).

3.2 What is in a contract?

Contracts are artifacts and, therefore, have a physical representation whose name and composition varies throughout the suite of possibilities sampled in this chapter. We present now a perspective of the historical development of the concept of contract in the software engineering field. Our goal is to understand the different approximations to the concept, the elements required to build a contract, and the way to create a contract from the composition of those elements. This is by no means an exhaustive study.

3.2.1 Historical Remarks

It is important to note that the general ideas exposed in verification and design through contracts were not originated by the Meyer's proposal, but during the early phase of research on program correctness, as noted by several authors such as [BLACK, P.E. 1998, GRIES, D. 1981].

Assigning meanings/predicates to programs. In 1967 Floyd created a method to assign meaning to programs initially through the introduction of assertions to a directed graph representation of a program [FLOYD, R.W. 1967]; Hoare developed Floyd's idea of a programming language for specification (Hoare's logic), showing that the axiomatic approach to language definition –in terms of how to prove a program correct, instead of how to execute it– might lead to a simpler design [HOARE, C.A.R. 1969]. This is the core concept in contracts, as seen in section 2.1.5.

This pioneering work was followed by the development of the concept of abstract data types (ADTs) [HOARE, C.A.R. 1972, LISKOV, B., *et al* 1974, GUTTAG, J.V., *et al* 1978] and the further development of proof techniques like [IGARASHI, S., *et al* 1975, GRIES, D., *et al* 1980]. Besides, various research groups worked heavily on programming methodologies, but found it very difficult to prove correctness even on the simplest programs.

Weakest predicate. Dijkstra was the one to open the way to a more generic approach to the formalization of programs. He proposed that it would be more productive to develop a program and its proof together (**correct by construction** philosophy), and the use of the termination condition for verifying [DIJKSTRA, E.W. 1976]. This replaced the “verification afterward” orientation of Floyd and Hoare. The weakest predicate is a given postcondition, which in this case is the strongest or more specific one. “*Suppose we have a predicate Q*

and a command C, the expression $wp(C, Q)$ denotes a predicate that describes the set of all initial states from which executing C will lead to the termination in a state that satisfies the postcondition Q [GRUNDY, J. 1993]. Dijkstra also created a calculus (the *guarded command language*) for the derivation of programs (*program calculation*) based on weakest preconditions. His approach is a mix of formal and semiformal reasoning, and is categorized as rigorous; the degree of informality makes tool support difficult. Dijkstra justified the partial informality of the approach because of the *“freedom offered by pen and paper”*[GRUNDY, J. 1993].

3.2.2 Operation Schemas

Parnas’ work on modularization and information hiding [PARNAS, D.L. 1972] allows modelers to build a set of descriptions. The first one depicts the functional part of the system (the operations), and the second one represents the control of those operations. The operations are specified by schemas, artifacts that describe what constraints the action was supposed to satisfy. This reduced the difficulties found during the design and implementation of correctness proof. Nonetheless, applying Parnas’ approach to any nontrivial program remained a highly complex task.

Hence, the next step was the step-wise refinement²¹, a bottom-up approach that reduced the complexity of dealing with the formality and abstractions and that maximized the product of the effort. As an example, VDM [JONES, C.B. 1990] established a strategy where after specification, the ADTs are replaced by concrete-data types (data reification) and then the step-wise derivation of an implementation takes place through the operational decomposition. The most interesting aspect is that any a given moment the target system is partly implemented and partly specified. This is possible because VDM uses a three-valued logic that allows for the notion of partial functions (because programs are rarely total). *“However, with the introduction of «undefined», the approach is more complicated than the traditional «total function» model”* [SENDALL, S. 2002].

During the same period, the formal specification language Z [SPIVEY, M. 1990] –based on set theory and first-order logic– was also widely adopted. In this case the unit of specification is the schema, which can be used to model both the static and dynamic properties of systems. [SENDALL, S. 2002] states its limitations:

“A schema has only two parts: a declaration and a predicate part, which means that preconditions and postconditions are intermingled in the predicate part of a schema for an operation... Unfortunately, the schema notation does not distinguish the role played by different schemas, for example, whether a schema represents an invariant or an operation”.

On the other hand, the approach is powerful as it supports composition (of various schemas using schema-level operators such as hiding, disjunction, conjunction, negation and composition) and incremental specification.

The wrap-up technique is an alternative approach: a formal language under the hood of a semiformal, more manageable representation. The first to implement a solution of this type was an axiomatic wrap-up for the programming language Pascal [HOARE, C.A.R., *et al* 1973]. Afterwards, the use of wrap-ups is popular because it is possible to guarantee the coherence and semantics of different modeling languages in more or less general cases, see, for example [PETERSON, J., UNION, I.T., pUML, RICHTERS, M., *et al* 1998]. By contrast, [GRIES, D. 1981] demonstrates that axiomatizing a language that is not created with axiomatization in mind requires a huge effort.

²¹ Decomposing the design process into a number of steps.

At this point in time, Meyer identifies the need for a technique that introduces reliability mechanisms naturally into the system [MEYER, B. 1988]:

“Surprisingly, few programming languages have included syntactical provision for assertions; ... The connection with object-oriented development introduced by the [contract] was foreshadowed by the assertions of CLU²² which, however, are not executable.”

3.2.3 Meyer’s “Design by Contract”, 1988

Meyer considers the routines as the starting point for building correct software and, therefore, for his method “Design by Contract” (DBC) [MEYER, B. 1988]. His method can be classified as a white-box approach for specification because all the details are observable. Meyer even created a programming language –Eiffel–as a mechanism for dealing with the declarative/imperative nature of the method. For Meyer the Eiffel language is the expression of DBC and vice versa.

Eiffel is an object-oriented language where the classes are the base element of design, and they contain the different routines or services for a given class; moreover, in DBC the services are the reason for a class to exist. Each routine can be implemented as in any other programming language (*imperative* or prescriptive part). In general, communications are synchronous because the routines obey the call-type protocol: the sender waits for the event to be received and an answer to be emitted. Events that are incoming to the system can cause the system to both change its state and output events to its environment.

Note that in the analysis phase, Design by Contract (and all other types of contracts in this chapter, except when indicated) assumes that all system operations occur instantaneously, i.e., the event is communicated from the sender to the system and the corresponding operation is executed all in zero-time.

Eiffel also provides primitives for the inclusion of assertions that permit establishing the conditions (scenarios) under which the execution of routines are valid, and to depict the obtained final conditions (scenarios) that hold after the execution of the routine. These are, correspondingly, the pre- and postconditions and constitute the *declarative* part of the Eiffel program. Finally, the third kind of assertions –known as invariants– can also be included.

A typical class specification is the following:

```
class ClassName feature
  attribute declarations...
  routine_name (argument declarations) is
  require
  Assertions - Preconditions
  Do
  ... code goes here ...
  ensure
  Assertions - Postconditions
  rescue
  ... rescue code goes here ...
  [ retry ]
end - routine_name
...Other routine declarations...
invariant
  Assertions
end - class ClassName
```

3.2.3.1 Exceptions

Through the chapter, we have explained how to use the contracts to specify the conditions

²² CLU is a programming language created by Barbara Liskov. Please refer to the substitution principle [LISKOV, B., et al].

that hold for the normal, well-behaved cases. But what happens when an error is detected? Meyer also proposes a robust approach to *error-handling* [MEYER, B. 1992]:

First of all, it's not useful to test for errors for which there is no planned handling procedure.

By default, contracts do not guarantee any behavior when a violation occurs (either a precondition is not true or the precondition cannot be reached)²³.

DBC proponents suggest using exceptions in order to recover from an error. They cause control to be passed to some exception handler, which is a portion of code able to attempt to fix the error or to admit that it cannot do so. Raising the exception notifies the caller that the method has failed. The most common error (a failure) occurs when the preconditions cannot be satisfied, for example. There should be provisions on the client side to cope with this kind of situation.

Eiffel provides, consequently, such an exception-handling mechanism as can be seen in the section **rescue** of the preceding template.

If an exception occurs in a method, then the rescue code is executed. The rescue code should try and fix the problem and retry execution of the entire method (when the **retry** instruction is included as part of the rescue code).

If the rescue code does not retry the method, the **Organized Panic strategy** takes place: the exception is passed up to the invoker of the currently executed routine, and so on, until either a handler can successfully fix the problem or the first level of the calling sequence chain is reached and the program dies (the "fix" being to abandon the program).

This constitutes a simple and robust exception handling mechanism that is able to deal with run-time, dynamic, real-world situations. This approach is adopted by the mainstream of object-oriented languages.

3.2.4 Wirfs-Brock et al, 1990

In their method the **operation** is the base concept for creating contracts. Operations materialize the services provided by a system, that is to say, the set of its responsibilities facing the users and the environment in general.

"Grouping responsibilities into contracts help us understand our design. We can use contracts to reason about the services provided by a class" [WIRFS-BROCK, R., et al 1990]; we may add that this responsibility-centered reasoning process is expanded to other parts of the methodology in order to understand the goals of each design decision and, finally, of the system as a whole²⁴. Responsibilities are mapped into classes directly.

A class can support one or more distinct contracts, and a contract corresponds to a cohesive set of responsibilities such as performing some action or handing out some information. A group of classes may actually provide a service (the notion of service is not clearly defined). A specification for each class includes: *a*) its overall purpose, and *b*) its contracts and responsibilities as well as all associated signatures.

First the responsibilities are established, then the contracts are created, and finally the collaborations (implicit in the contract template below, clauses *Server* and *Client*). The collaborations are designed in order to fulfill the contracts.

This form of contract takes into account the fact that every class may have several interfaces to provide different sets of services to different client classes; this is beneficial and is clearly grasped by Objectory and, hence, the UML language [BOOCH, G., et al 1998] and the Rational Unified Process [KRUCHTEN, P. 2000, SCOTT, K. 2002].

²³ In good design, this situation is not acceptable. Doing something not specified, or doing nothing cannot be considered robust behaviors.

²⁴ This argument will be retaken in the chapter 5, for establishing a new paradigm for contracts.

Nonetheless, this aspect happens to be extremely tight in the way it is described on the Wirfs-Brock contract. As can be seen, the clients are explicitly indicated and this binds them statically from the beginning. This is an early decision that may affect the outcome of the development process.

Natural language is the specification language of choice. The contracts follow this schema in the first phase (named **assigning collaborations to contracts**):

```

Contract      NumberOfContract: Text-Contract-Short-Description
Server:      Name
Client:      Name
Description: Text-Longer-Description

...Other contract declarations...

```

Once the collaborations are mapped, the specification of each class is made, including the contracts though in a different manner:

```

Class: ClassName
Superclasses: Class
...
Contracts
Text-Contract-Short-Description
Text-Longer-Description
Routine1 (params) returns result
Uses ListOfClasses
Text-DescriptionOfRoutine1
... more routines here...

Text-Contract-Short-Description

... routines and more contracts...

```

This mapping to classes is also used in the other approaches explained below.

It is important to note that in this proposal, there is no mechanism for handling exceptions explicitly and no communications scheme is mentioned.

3.2.5 Fusion, 1994

In Fusion the **collaborations** are the basis for establishing the contracts. As objects collaborate in order to accomplish each of their responsibilities, these ones become contracts between the objects that are clients on the collaboration and those who act as servers [COLEMAN, D., *et al* 1994]. Like Wirfs-Brock, a Fusion class can also support one or more distinct contracts.

Two differentiated phases, analysis and design, take place. In the first one, the system is treated as a black-box; neither the classes inside the object model nor those ones inside the system object model have methods (i.e., responsibilities) assigned to them, because “analysis” classes describe concepts of the problem domain rather than software components. Analysis places a particular emphasis on defining the system interface and the information that is relevant to the system for the purposes of fulfilling requests from its environment. These two aspects of the system are described by the object model and the **interface model**.

Contracts are defined in the context of the interface model, which consists of the **Operation model** and **Life-cycle model**. It defines the inputs and outputs of the system. The communication is asynchronous, which means the sender does not wait for the event to be received. Events that are incoming to the system can cause the system to both change its state and output events to its environment. A pair consisting of an input event and the corresponding effect is called a **system operation**.

The contracts are declarative and written in the form of **Operation Model Schemas** accompanied by Life-cycle models; a schema is written in (structured) natural language, and it defines the pre- and postconditions of a system operation, and the events that are output.

By including event sending as part of the postcondition, it is possible to clearly state under which circumstances events are output.

The general format for an Operation Schema is the following [COLEMAN, D., *et al* 1994]:

Operation:	Name
Description:	Text
Reads:	supplied Item1, Item2,...
Changes:	new Items
Sends:	ListOfEvents
Assumes:	Assertions - Preconditions
Result:	Assertions – Postconditions
<i>...Other operation declarations...</i>	

The **Assumes** and **Result** clauses are analogous to the pre- and postcondition clauses, respectively. The word **supplied** implies that the **Item** in question is passed as a parameter while **new** reveals the **Item** is created during the operation.

This is the first of the methods studied in this chapter where the state becomes important in terms of delivering a service. As a consequence part of “functionality” of the contract is displaced to other models and thereby enriched by these other models resulting in a stronger set of conditions (pre- and postconditions).

There’s no exception handling mechanism but the authors consider it necessary. As a matter of fact, the authors mention the use of Eiffel and the **assert** macro or the **try/catch** mechanisms of C++ in order to implement it.

3.2.6 Catalysis, 1999

The Catalysis approach is a component-oriented development method. It offers a very rich set of features for modeling component-based and object-oriented systems. Catalysis integrates many of the favorable features of both Fusion [COLEMAN, D., *et al* 1994] and Syntropy [COOK, S. and DANIELS, J. 1994], particularly Syntropy’s use of OCL for describing pre- and postconditions on operations [SENDALL, S.]

Catalysis defines three levels of modeling²⁵:

- *problem domain or business* – captures the relevant concepts for the environment of the system and for the stakeholders of the system;
- *component specification* – captures the external behavior that the component should exhibit;
- *Component implementation* – captures the internal workings of the component.

Actions are the basic modeling elements in Catalysis. However, collaboration as the minimum relevant modeling unit. Collaborations consist of combinations of actions that accomplishes some goal; by contrast, Use Cases are sets of sequences of actions whose descriptions transport also the corresponding sequencing information [CATALYSIS 2002].

Contracts appear in the context of Behavior Modeling and of Interaction Modeling, and describe Actions of the system or one of its components. Actions can be decomposed into subordinate actions, or composed to form a super ordinate action. An action represents work performed by one or more entities, which is described by an action specification, generally using pre- and postconditions written in OCL²⁶. Catalysis defines two main kinds of actions: **localized** or **joint actions**, according to whether it has one or many participants. More precisely, Catalysis prefers to call joint actions to the multi-party collaborations, and

²⁵ In general, Business modelling is not considered specifiable but loosely describable for the most part of authors. This will be discussed in chapter 5.

²⁶ OCL is a specification language. Refer to chapter 2 for further information.

localized actions to the interface declaration of the services provided by a particular component.

It is also possible to state as part of a postcondition that another action is invoked either synchronously or asynchronously, and the invocation takes the form of the sending of a message.

Unlike the previous approaches, an action may take up some undetermined period of time. Because of that, some actions are quite interesting not only for what they have achieved after they have finished but **for what they do while they are in operation**. This is discussed further in section 3.2.6.1.

In order to create their *behavior models* for object types, Catalysis takes all the elements from previous approaches and provides a form of contract that is capable of dealing with concurrent actions:

Action:	name :: (parameter1: Type1, parameter2: Type2) : ResultType
Pre:	Assertions - Preconditions
Post:	Assertions - Postconditions
Rely:	Assertions (for concurrent or interleaved actions)
Guarantee:	Assertions (to be maintained as true during actions executed concurrently with others)
Inv:	Condition (applies for every action in the model)
Inv effect:	Assertions (applies to all actions conforming to signature, pre and rely conditions)

3.2.6.1 Rely and Guarantee-Conditions

Taking a more focused view of the effect of operations on the state of a system, it is possible for a system to have operations that change the state of the system concurrently and even at the same time²⁷. Furthermore, it is possible that operations access common resources of the system. If such operations were to execute in parallel, then interference problems could arise and the classical sequential program theory does not apply anymore; in these cases, pre- and postconditions cannot ensure correctness.

Rely- and **guarantee-**conditions allow one to cope with the specification of concurrent operations that share resources and may have overlapping executions. In two logical parts: The assumptions consist of the *precondition* and *rely-condition*, and the commitments consist of the *guarantee-condition* and *postcondition*.

This point was first examined in section 3.1.6, but now some features are added: ***“The meaning of the assumption/commitment specification is the following: if all activities other than the one executing the corresponding operation observe the rely-condition during the execution of the operation and the precondition holds initially (the assumptions), then the operation will terminate in a state that satisfies the postcondition and the guarantee-condition will have been held by the operation throughout its execution (the commitments).”*** [SENDALL, S. 2002]

Any change to the system state by any other operation over the period of execution is supposed to satisfy the *rely-condition*, while any change to the system state by the operation must satisfy the *guarantee-condition*. If this is not the case, nothing can be stated about the results of the operation.

3.2.7 Kobra, 2002

This is a component-oriented development method, specifically of “Kobra components” or **Komponents**. In this case the contracts are considered in the phase of the **Komponent specification**.

The basic goal of *Komponent specification* is to create a set of models that collectively

²⁷ This concurrency phenomenon may also happen because the actions have a long lifetime.

describe the externally visible properties of a Komponent. In the UML an interface represents a set of operations, but a Komponent specification also includes information about the behavior of the Komponent, the logical effects of its offered operations and its expectations about the Komponenten that surround it (that are in its environment) [ATKINSON, C., *et al* 2002].

The specification therefore defines the *requirements* that the realization of the Komponent must satisfy, like in Catalysis. A Komponent's specification represents the *contract* between the *Komponent* and its clients and servers²⁸. A Komponent specification may contain up to six distinct artifacts where only the **Behavioral Model** and the **Functional Model** are in the scope of our interest. The two are considered as primary artifacts within Kobra and correspond roughly to the models that specify the properties of a system in OMT [RUMBAUGH, J.R., *et al* 1991] and Fusion [COLEMAN, D., *et al* 1994].

The *Functional Model* describes the externally visible effects of the operations supplied by the Komponent. It consists of a set of operations specifications, one for each operation, constructed using the template shown below.

Name :	OperationName
Constraints:	Text-properties
Receives :	Information input
Returns:	Information output
Sends:	Events or operation invocations
Reads:	Externally visible information accessed by the operation
Changes:	Externally visible information changed by the operation
Rules:	rules
Assumes:	weakest precondition
Result:	strongest post-condition

Several remarks about this description:

- The **Constraints** field contains an informal description of the operation effects, both for normal and exceptional executions.
- The basic goal of the **Result** clause is to provide a declarative description of the operation in terms of its effects.
- The *precondition* and *post-condition* (in the last two clauses) are to be written in some undefined language, depending on the criticality of the application, i.e. from free-style text to fully formal languages.
- The *assumes* item introduces weakest precondition, introduced in section 3.2.1.

Finally, notice that the *rules* clause may hide some of the functionality of the contract, which in the other schemas would certainly appear exclusively in the sections for preconditions, postconditions and invariants.

3.2.8 ANZAC, 2002

Anzac [SENDALL, S. 2002] deals with contracts and defines a methodology for specifying reactive systems in terms of these contracts. The basic modeling element is the Use Case, which is made up of interactions; contracts are created to fulfill them.

The Anzac methodology defines both a **Stakeholders Contract** and a **Design Contract**. We are interested only on this second kind of contract.

A Design contract is an **Operation model** which is an operation schema complemented by a **Protocol model**. It clearly defines the interface between the system and its environment, and would provide a basis for validating the system's role in that environment by making sure that each constraint that it places on the design of the system is aligned with –and traceable

²⁸ Actually, this is true for all kinds of specifications. The contract mentioned here is more of a conceptual one.

to— the (system-related) goals of the various parties that have a vested interest in the software application²⁹.

The Operation model describes each operation via a collection of **Operation Schemas**. This is a clear heritage from Fusion [COLEMAN, D., *et al* 1994]. The Protocol Model for a system is analogous to the Life-cycle model of Fusion, too, and serves as a filter for the sequences and scenarios in which a given message can be processed.

Operation:	SystemTypeName::OperationName (ParameterList)
	: Return Type;
Description:	Text-Description
Notes:	Text-Notes
Use Cases:	UseCaseName1:: {Step1, Step2...} + UseCaseName2:: {Step1, Step2...} + ... ;
Scope:	ClassOrAssociationName;
Messages:	Message1, Message2, ... {MessageN Throws ExceptionMessagesN}, ... ;
New:	ListOfObjects
Aliases:	ExpressionSubstitutionDeclarations
Pre:	Assertions - Preconditions
Post:	Assertions - Postconditions

The several new items found in this schema are explained briefly next:

- The **Use Cases** clause declares all use cases (1 or more) that have a (traceability) relationship with this operation.
- The **Scope** clause declares all those classes and associations from the Concept Model that define the name space of the operation.
- **ExceptionMessages** defines the exceptions that are thrown by the call denoted by the message preceding the *Throws* keyword.
- The **New** clause provides a declaration of all those names in an Operation Schema that refer to new objects or (new) messages. All names declared in this clause are local to the schema.
- The **Aliases** clause provides a declaration of all those names in an Operation Schema that refer to expression-substitutions. Everything declared in this clause is local to the schema. An expression-substitution is similar to a macro.

Additionally, Anzac proposes the use of the *Rely expression*, which has some similarities to *rely-conditions*, because they both define “during” invariants. In contrast to *rely-conditions*, *rely expressions* are used within the **Post** clause and they have scope over only a subset of the operation’s effects. Another difference is that a *rely-expression* has a fail predicate that is asserted if the condition that is relied upon can not be held during the period the service is being carried out.

3.2.9 Diagrammatic Approaches to Contracts

The precise UML group proposes the **3-D Box** as an extension of Constraint diagrams for specifying actions [KENT, S. and GIL, J.]. Constraint diagrams –presented in chapter 4— permit modeling propositions by using diagrams that are based on the Venn-Euler-Pierce basic diagrams for sets.

A 3-D contract is shown in figure 6. It is composed of a pre-condition (top diagram) and a postcondition (bottom diagram). The notation for the constraints (on the left) is based on the basic diagrams for representing sets (please refer to section 4.2.2.3). Some additional information, using a UML-like notation is given on the right side of the diagram.

²⁹ These parties are commonly referred to as the *stakeholders* of the system.

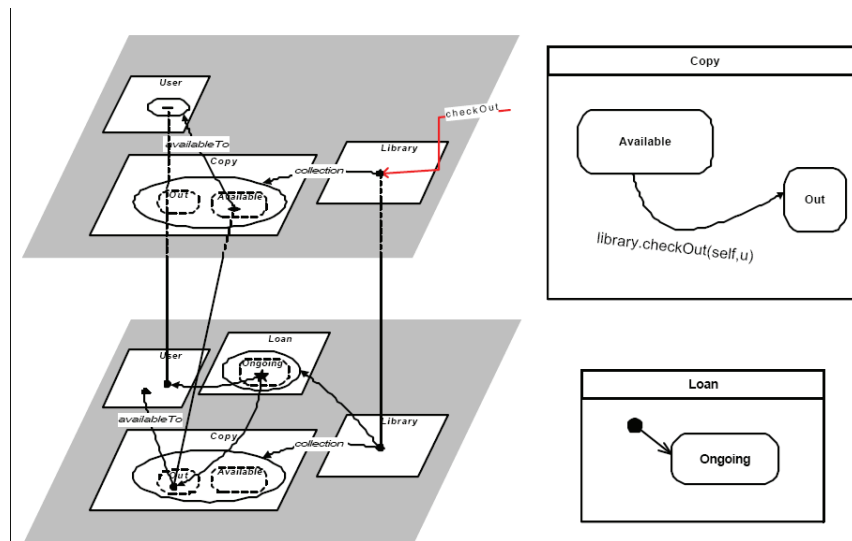


Figure 6. 3-D box that specifies the way a reservation can be done using a library IT system. It represents how users of the library can loan copies of books. From [Kent, S. and Gil, J.]

Figure 6 shows the example of a library reservation system. The figure describes the rules for making a new loan, and what happens to pending loan requests. It is clear in this figure that the notation presents scalability problems (see section 4.2.3) and uses *ad hoc* notations that are not compatible with UML.

3.3 Summary

Contracts are used in many methodologies as a powerful tool for specification during the analysis and design phases, even if almost always this is not recognized explicitly³⁰. Their importance is synthesized in the **Law of Excluded Miracles**: “*if we don’t state what a module should do, there is little likelihood that it will do it*” [GRIES, D. 1981]. As it happens in social contracts, “*the specification of the obligations and benefits afforded to both, the client and supplier in an interaction*” [ILLINGWORTH, V., *et al* 1996].

In this chapter we showed that contracts are useful to state clearly what the system can do, without looking at the performance, the optimality of the distribution, and other issues. Even if a contract can serve to describe structural and behavioral aspects [MITCHELL, R., *et al*], we focused on it as a mechanism for producing “systems without bugs” [ISE 2001, MEYER, B. 1997]. Similar to contracts in object-oriented programming, software contracts can help avoid errors that are process-related and conceptual [IEEE 1993] [BEIZER, B. 1990].

A contract describes the net effect of an action or service. Contracts are written in the form of predicates (declarative style) that allow for thinking more clearly in terms of the problem domain. As a consequence, by not stating the precise solution, the modeler can defer the decision of how to implement the actual system.

Contracts let the modeler focus on the problem. Contracts are, however, not straightforward; the declarative style represents a formidable intellectual problem: by nature, people tend to pick up a concrete solution and optimize it, instead of constraining a space of possible design solutions [GLASER, H. ET AL 2001, JACKSON, D., ET AL 2000] as is required by the declarative style of specification.

³⁰ One reason for this is the fact the term “contract” was coined and later trademarked by Bertrand Meyer.

In theory, contracts have shown their capacity for specifying the interfaces in a complete manner. *“In practice, it is amazing to see how far just stating what a module should do goes towards helping to ensure that it does it”* [MEYER, B. 1997]. An interface specifies the semantics of the access points to the services provided by a software object/component. In this chapter we demonstrate that, in practice, contracts also describe part of the internal behavior of the system. This makes them more meaningful as specification artifacts. It is difficult to study the different forms of contracts without regarding thoroughly the details of each methodology. A summary of their common elements is shown in table 3.

Table 3. Elements for description of contracts

Elements	Reserved word	Interpretation
The <i>Operation</i> clause declares the signature of the operation.	N.A.	Action
The <i>Description</i> clause provides a concise natural language description of the purpose and effects of the operation.	N.A.	N.A.
The <i>Precondition</i> clause presents the <u>required</u> scenario that enables the execution of the service.	Pre Rules Inv	Constraints
The <i>Postcondition</i> clause presents the <u>obtained</u> scenario after the execution of the service.	Post Result Changes New Inv	Constraints
The <i>Messages</i> clause declares the possible messages that can be output with the execution of the operation. This clause declares the type of messages that can be sent by the operation together with their destinations, i.e. the receiving actor classes.	Receives Sends Returns Messages	Communication – Input/Output
The <i>ExceptionHandling</i> mechanism indicates what should happen in the event of the non-compliance with invariants	Rules Returns Messages	N.A.

We demonstrated that the notion of contract in the software engineering field is pervasive. It has evolved recently, but the roots can be traced to the beginning of the domain, under different forms, degrees and notations. As a taxonomy for contracts in the software engineering field has not yet been made, we propose a historical categorization.

Table 4 is a synoptic comparison of the expressive capabilities of the different types of contracts studied in this chapter³¹. It is clear from Table 4 that the techniques studied in this chapter use contracts that do not integrate the different aspects of change, and formalization is not supported. Hence, these aspects must be addressed by our specification notation and artifact.

Moreover, we have identified several specific challenges that our notation should also address:

- Contracts are textual. A visual approach is required.
- Contracts not only describe the behavior on the interface of the system, but also describe part of the internal behavior of such system. The signature-based approach must be complemented by primitives that enable the description of internal behavior.
- Contracts should be used in a hierarchical fashion, not making differences among services at the system level and at the level of its composite systems.

³¹ The elements cited in the *operation prerequisites* and *operation effects* are the labels that appear in each of the contract templates.

Table 4. Comparison of the different approaches for contractual specification

Design by contract (1988)	Wirfs-Brock (1990)	Fusion (1994)	Catalysis (1999)	Kobra (2002)	Anzac (2002)
Object-oriented programming	Object-oriented Analysis and Design	Object-oriented Analysis and Design	UML Component-oriented	UML Component-oriented	UML Reactive systems
Routines in programs	System service Collaboration	Collaboration	Localized actions: Joint actions	Component interface	Use Case
Class methods, functions	Analysis-phase: a group of classes can provide a service Class support one or more contracts	Interface model = operation schema	Services provided by a single component Services provided by multiparty collaborations	Component Functional model – behavior:	Design contract (operation schema + protocol model)
Pre Inv	Pre Inv Client-server static binding	Assumes (input) Reads	Pre Inv Rely Guarantee	Assumes Receives Reads Rules	Pre Messages
Post Inv	Post Inv	Result Changes Sends	Post Inv Inv effect	Result Changes Sends Returns	Post Messages New
Synchronous, call-type protocol	No	Asynchronous	Synchronous & Asynchronous	Synchronous & Asynchronous	Synchronous & Asynchronous
Yes	No	No	No	No	No
No	No	No	Yes	May be with rules	Yes
No	Client-server static binding	Lifecycle model complements	May be, via assertions	Expectations about Components that surround it	No
Eiffel	Natural language	Structured natural language	OCL	Informal Declarative language (may be OCL)	OCL
No	No	No	No	No	No

4 Creating Visual Models

There are no correct or incorrect models.

Models are more or less useful.

Martin Fowler, Analysis Patterns: Reusable Object Models

As seen in Chapter 3, software contracts are mostly textual specification artifacts. Our goal is to allow modelers to create visual representations of systems, in order to deal with complexity. Therefore, we need to create a visual model that corresponds to a software contract.

Our main source of inspiration is the work of Shin [SHIN, S.-J.], who demonstrated recently that formal visual notations can be built and used effectively to support reasoning.

In this chapter, we study the main aspects that should be considered in order to create adequate visual representations. As this is a broad area of research, we limit ourselves to the related works made in the computer science field. Moreover, we constraint the scope of our research to the specific needs of SEAM: a visual specification artifact that can be used to reason about systems, from a systemic perspective. At the end of the chapter we identify the requirements of our notation for describing systems in terms of systemic actions.

4.1 Problems for the Creation of a Notation for Modeling

In order to create an adequate visual notation we need to address three issues:

- *How do we create models that enable reasoning?* In other words, what is the language required to express, in order to support the reasoning process.
- *What are the expected results from analysis/reasoning?* In other words, what is the modeler supposed to obtain from this reasoning process?
- *How do we make the analysis of these models automatic?* Making the models compatible with formal languages allows us to use tools, as well as diagrammatic reasoning. Therefore, the resulting diagrammatic notation should not only support direct reasoning but also automatic or semi-automatic analysis.

This chapter addresses specifically the question *a* and lays a foundation for question *b* ³². Figure 3 illustrates the whole process of modeling: from conceptualization to analysis. This figure also makes evident the reasons for creating a new modeling notation. [AGRAWAL, A. 2003] affirms that the steps to develop a language include the definition of its syntax, visualization, semantics and algorithms for execution. In order to satisfy these needs, first we study what the model theory offers us, and how reasoning can be done using these models. Later, we study the state-of-the-art of visual notations for system modeling. Finally, we identify some limitations and patterns from these notations that are points to improve with our specification artifact.

4.1.1 Choosing What to Represent

The problem of how to choose what to represent is studied by the domain known as **knowledge representation** [RUSSELL, S., *et al* 1995]. The two fundamental questions to answer concern the representation itself:

- What entities should the modeler describe?
- What can the modeler say about these entities?

³² Question *b* will be complemented in chapters 7 and 9. Question *c* will be answered in chapter 8.

In the case of SEAM, previous works already attempted to answer these issues. The first question was already covered by the works of Naumenko [NAUMENKO, A. 2002], Lê [LÊ, L.S., *et al* 2005] and Wegmann [WEGMANN, A., *et al* 2005]. They defined the kinds of model that should be built in SEAM, the entities that should be modeled in each one of those models, and how to go through the different levels of the hierarchy. This definition of the ontology takes into account all the different aspects (namely, categories, measures, composite objects; time, space and change; event and processes; physical objects). The second question was addressed by the work of Preiss [PREISS, O. 2004] and Regev [REGEV, G., *et al* 2003]. Preiss defined what the primary and secondary properties of a system are, how to define the quality of such a system (in terms of quality attributes), and complemented the ontology in what concerns measures. Regev defined a method that permits the modeler to describe how and whether the quality attributes really enable the satisfaction of system goals.

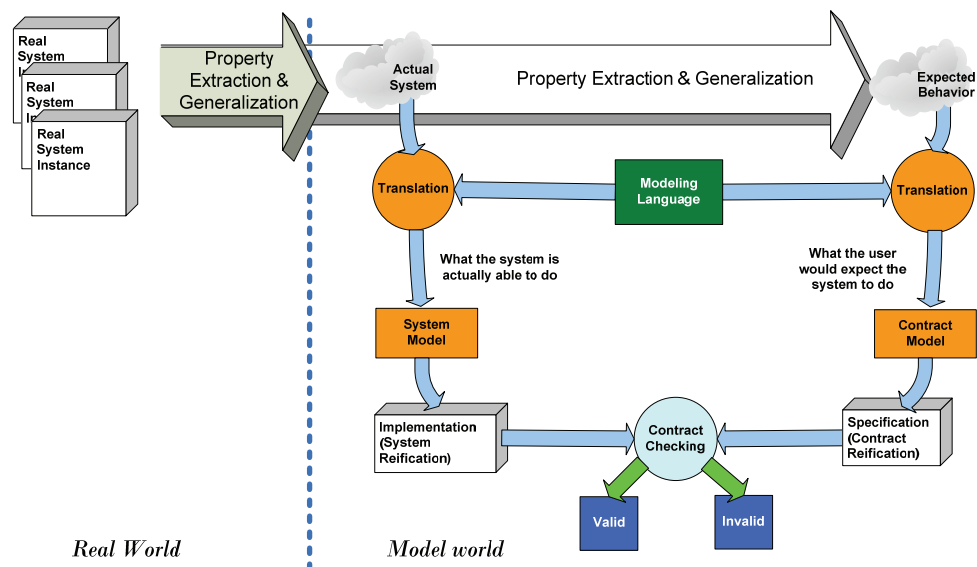


Figure 7. Model vs. reality. Expanded version of figure 5, that makes explicit the impact of modeling language in the validation process

4.1.1.1 Representation and Reasoning

In its original form, the **model theory** enable reasoning about system properties [HODGES, W. 1993]. The model theory explains that the models allow the designer to understand how the system is interpreted. There are essentially two manners to specify a system that can be used complementarily [MONIN, J.-F. 2000]:

- representing a system by describing its properties;
- Creating a model of the system by using a set of pre-built definitions.

In both cases it is evident that models are only representations of reality (actually, of the knowledge an observer has of that reality) that we build in order to be able to reason about the reality itself. Concretely, **elements** of the model have **properties**. These properties are expressed via logic axioms, and the models are built using operations on sets of elements. What we assert about the elements –what properties are true— is expressed via **facts**. This is illustrated in figure 3.

Facts are part of the world but their representations are not. As shown in figure 3, the representation of facts must be encoded in some way that can be represented in our specification of the system. We cannot put the world inside a real specification, so all reasoning mechanisms must operate on representations of facts, rather than on the facts themselves.

Once again, we must be clear about the real nature of this reasoning process: a model is not the reality. Therefore, it is important to understand the relationship among **logical consequence** and **semantic consequence**. The logical consequence is a result of the reasoning/transforming process only – that may remain completely syntactic—, and may be inadequate for reality. This means that an observer does not necessarily see (i.e. interpret) the result of some logical consequence, as it does not relate to the properties that she is capable of perceiving. On the other hand, the semantic consequence deals with the connection to the reality, and the fact that what should be interpreted from a set of predicates should also be true in the real world.

Therefore, in a certain situation –described also with predicates—such properties can be examined. This description of a specific situation is called a *configuration* (please see definition in section 3.1.7). Proper reasoning should ensure that the new configurations represent facts that actually follow from the facts that the old configurations represent. As we already said before, it is important to distinguish between facts and their representations. Because sentences are *configurations* of parts of the system, reasoning must be a process of constructing new physical configurations from old ones.

4.1.2 Choosing How to Represent

As shown in figure 3, there is a difference among the real system and its model. At least two processes appear before the one commits to the other: a feature-extraction process and a translation process. The first process serves to filter out the features that are not taken into account for the reasoning process. The latter process depends on the expressiveness capacity of the modeling language; this capacity restraints the way the solution is developed, as demonstrate by the quality problems [DIJKSTRA, E.W. 1976, WIRTH, N. 1995].

A **logic** or *knowledge representation language* is defined by [RUSSELL, S. and NORVIG, P. 1995]:

- The *syntax* of a language describes the possible configuration that can constitute sentences.
- The *semantics* determines the facts in the world to which the sentences refer. And with semantics, we can say that when a particular configuration (syntactical construction) exists within a system, the system believes the corresponding sentence.
- The *proof theory*—a set of rules for deducing the entailments of a set of sentences. The right side of figure 4 illustrates this deductive process, where a sentence can be entailed or deduced from a set of sentences that map a set of facts.

The goal of such a representation is to express knowledge in computer-tractable form in order to be analyzable [SOWA, J.F. 1999] by computer tools or to build systems that can perform well.

SEAM is built using first-order logic (FOL). First-order logic makes a stronger set of ontological commitments. The main one is that the world consists of **objects**, that is, things with individual identities and **properties** that distinguish them from other objects. Among these objects, various **relations** hold. Some of these relations are **functions**—relations in which there is only one value for a given input.

In order to deal with many problems in the “naïve” version of the set theory, Russell introduced the notion of types in order to avoid that the predicates and the elements could be used in a free way, leading to inconsistent models. According to our definition above of semantic consequence, the inconsistent models include configurations that cannot exist because they do not comply with the expected behavior of the actual system.

4.1.2.1 Cognitive Reasoning: Soundness vs. Intelligibility

A second problem with representation is the expression of the expected behavior. The modeling language also determines this aspect, as explained in figure 3. The reasoning process as such, can be separated into a series of complex propositions.

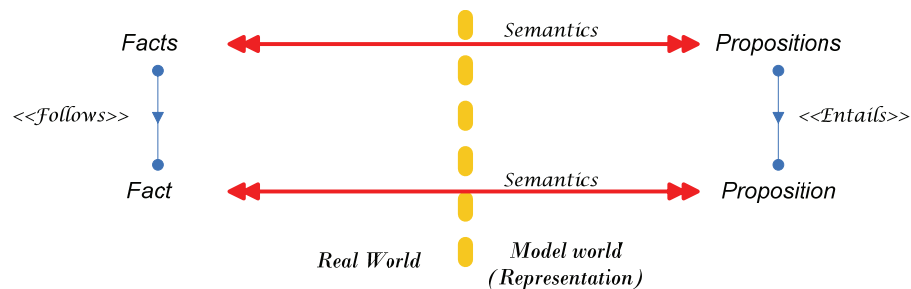


Figure 8. The connection between sentences and facts is provided by the semantics of the language. Adapted from [RUSSELL, S. and NORVIG, P. 1995], page 158.

According to [BUTCHER, K.R., *et al* 2004], there are three levels of reasoning: understanding each proposition (or *paraphrases*), the connection among propositions (i.e. *elaborations*) and from these to the context (i.e. *monitoring statements*), and making inferences (both from the propositions and from the connections of this propositions). Inferences are classified as *path* inferences, *nonpath* inferences and *integration* inferences. Because of the focus on the soundness of logical notations, only the first aspect is strongly integrated into formal notations. Sometimes the third aspect is also assimilated.

It seems that a more integrated approach is required. Current formalized diagrammatic notations are difficult to grasp by the practitioners and students, and are often replaced by notations whose format and expressive capacity allow putting together larger amounts of information [CHERUBINI, M., *et al* 2007]. Unfortunately, these notations are *ad hoc*. We can say that such forms of representation support the second type of reasoning and the corresponding inferring process; these notations permit creating shared knowledge quickly and efficiently.

As a result, our notation should improve intelligibility, not by reducing the complexity of what is represented, but by integrating more information in order to support direct diagrammatic (visual) reasoning.

4.1.2.2 Support for Perceptual (Visual) Reasoning

“Les diagrammes sont essentiels pour la modélisation systémique”
(Diagrams are essential to systemic modeling)

Durand – La systémique

The preceding numeral raised an interesting issue: strict attachment to formal translations is not successful on creating notations that are easy to handle. However, *ad hoc* notations, created for rapid exchange of information, are able to encode complete sets of knowledge. This happens because –and despite– of the lack of a formalized approach. If we reason on the contrary, we should value the visual dimension of the notation as important as the formal dimension.

[LARKIN, J., *et al* 1987] showed first that visual notations add a local indexing advantage over the sentential and tabular notations. [CHENG, P.C.-H. 2004] found that the real advantage is the recognition of paths, and of applicable rules leading to solution planning. This is apparently due to the structure of the representations: *“the design of effective representational systems to support problem solving and learning... a good representation should use location indexing as a means to coordinate information that will be needed to problem solving”*.

This phenomenon is adopted in diagrammatical approaches in the form of the principle of **immediacy**, defined in the works of [AKKOK, N. 2004] and [SWOBODA, N. and ALLWEIN, G. 2002]. In Swoboda’s version, this principle establishes that there is a logics of observation, how immediate the information of a property related to it is represented in a diagram. On the other hand, Akko’s version is more related to the distance among different sources of information, about the amount of information that can be indexed locally because of the nature of the diagrammatic representation. In both cases, the works address the description/design of complex, homogeneous systems.

On the other hand, the modeling of heterogeneous systems (considering more than one type of properties: state, communication, actions, etc) is very complex. There is always a tradeoff among localization problem and information stacking problem. *“Reasoning practices and decision making often require information from many different sources... there are many advantages to reasoning with the diagrams themselves, as opposed to re-expressing the information content of the diagram in sentential form and reasoning in an abstract sentential language”* [SWOBODA, N. and ALLWEIN, G. 2002].

In the domain of information visualization, several principles and laws guide the design of notations and interfaces used for representing information visually. One of the most important frameworks is known as the **“Gestalt Laws”**, created after the works of the Gestalt school of psychology [AUDI, R. 1999]. The main laws proposed in this framework are: *proximity, similarity, connectedness, continuity, symmetry, closure, relative size and figure and ground* [GLASER, H., *et al*]. Further discussion of these laws and how the notation satisfies them appears in chapter 6.

Another aspect that is of interest to us is the representation of change that we also study. It is used mainly for the representation of flows and also in the case of integral and differential calculus [WARE, C.]

4.2 Techniques for Reasoning with Diagrams

“It is common to consider them (the diagrammatic approaches) as an auxiliary means valuable for heuristic consideration whereas to be precise and implementable schemas should be converted into ordinary linear (string) specifications based on the well-established notions of term and formula.”

Diskin -- [DISKIN, Z. 1995]

In this section we introduce a series of works of Information Systems field that explore the utilization of visual specifications against the more traditional ones, namely, sentential and tabular.

The actual trends in software development show that graphical notations are more popular than sentential ones (i.e., textual or mathematical). However, diagrams are in general considered as good intuitive, informal aids to increase the understanding of the system rather than as formal models of the same system [BRUEL, J.-M., *et al* 2000, DISKIN, Z. 1995, SHIN, S.-J. 1995]

4.2.1 Visual Formalisms

As we said in section 3.1.1.1, the techniques of formal specification are sometimes called specification by properties or specification by models. Most formal methods or languages are **sentential** or **textual**. They associate a strict semantics to a language, in order to make it mathematically sound. A detailed study on the different families of notations can be found in [GERVAIS, F. 2004]. As they are the most popular and more developed type of representation, their influence is paramount for the creation process of our graphical language.

A second, intermediate form of specification is the **tabular form**, where the elements of the specification (behaviors, structural elements, logical rules) are arranged into tables [PARNAS, D.L., *et al* 1998, SHIMOJIMA, A. 2002]. Being a two-dimensional form of representation, tables exhibit already several of the advantages of diagrammatic notations (e.g. local indexing, contextual indexing) [LARKIN, J., *et al* 1987]. Therefore, the tabular form is a nice compromise among a very formal approach and a diagrammatic one.

On the other hand, diagrams are to understand, describe, document, and other human-related processes, but not necessarily related to some formal method. “Node-link diagrams” or “Boxes plus arrows” is the standard approach [WARE, C. 2004] with differences in semantics. This was demonstrated by empirical studies such as [CHERUBINI, M., *et al* 2007]. Shin [SHIN, S.-J. 1995] and Hammer [HAMMER, E. 1995] demonstrated that diagrammatic approaches can be formalized. They laid out the foundations of a new domain. They demonstrated that diagrams can have the same logical status as sentential systems. This gave birth to a broad field of research known as Diagrammatic Reasoning (DR).

4.2.2 Expressing Constraints Visually

There is a rich literature on the construction of diagrammatic formalized languages (i.e., notations whose semantics are formal³³, at least in part). The motivation for the formalization effort is explained in [EVANS, A., *et al* 1998, PUMML 2002]. We can identify two main trends in the development of formalized diagrammatic notations: graph-based, and non-graph-based. This categorization takes into account only the nature of the underlying theory.

Table 5. A classification of visual formalisms based on the aspect they illustrate

	Structure	Behavior
Reactive	SA/RT – structural part	<u>Observability</u> CCS, CSP, Pi-Calculus Causality: Petri nets, Statecharts, SDL
Information Systems	Entity-Relationship UML: Class + object + deployment diagrams	UML: Activity + sequence + interaction diagrams BPMN, Structured Analysis, IDEF0/SADT,

[BARESI, L., *et al* 2005] presents an approach to create automatic interpretation for diagrammatic notations, and includes a rigorous and more generic state-of-the-art on this domain. As [WINTERSTEIN, D., *et al* 2004] shows, the use of diagrammatic logic can even support reasoning processes about very complex large mathematical problems. [JARRATT, T., *et al* 2004] studies the use of visual notations for product modeling in a complex design, multi-variable, multi-actor process.

4.2.2.1 Reasoning with Graph-based Notations for Behavior Modeling

The graph-based notations are all those based on the use of the mathematical notion of graph. A graph is defined as a structure consisting of nodes and links [WARE, C. 2004]. It

³³ The formal methods and notations are introduced in section **Error! Reference source not found.**

can be considered as the graphical expression of a process algebra automaton. The diagrams are generally drawn in the form of a directed graph, made of a set of nodes and of a set of transitions, which form the vertices and arcs respectively. Graphs show status/actions and transitions that form a sequence. There is one **starting node** and one or more **terminating nodes** (one per branch of operation). Because of the explosion of connections and nodes for complex systems, a large graph is not easy to tackle without sophisticated tools.

We can affirm that graph-based representations are the most popular and natural approach. The most important example of graph-based formalized notations is the Petri net and its derivatives (e.g. Colored Petri Nets—CPN, High-level Timed Petri nets—HLTPN, among many others). Petri nets are a pervasive tool in many domains, particularly for modeling/simulating problems that can be translated to sets automaton, agents, protocols, industrial processes and business processes, among others. Other examples of graph-based formalized notations are Flowcharts, Structured Analysis (SA) [WARD, P. 1985, YOURDON, E. 1988], [HATLEY, D., *et al* 1988], and SADT/IDEF0 [MARCA, D.A., *et al* 1988], Diskin's Sketches [DISKIN, Z., *et al* 1998],.

- In the case of flowcharts, the nodes are the actions of the system, and the arcs denote the sequence of actions. This is the less complex and less formalized approach in this series.
- Petri Net are a more mathematical or formal approach. In Petri Nets the nodes are called places, and the links are called triggered transitions. In statecharts, the nodes are the either the states the system can be on, or the branching operators (condition, fork, join), and the arcs are the state transitions, which include not only triggers but enabling conditions.
- The workflow notations, notably YAWL and BPMN, inherit from Petri Net and from Structured Analysis.
- More formal notations are related to process algebra. In other words, they are not graphical. These notations are used to create models of extended communicating finite state machines such as CCS[MILNER, R. 1980, 1989], CSP[HOARE, C.A.R. 1985] and π -calculus [MILNER, R. 1999].
- Lately, in a special, new generation of workflow systems, we find BPMN [BMIDTF 2007].

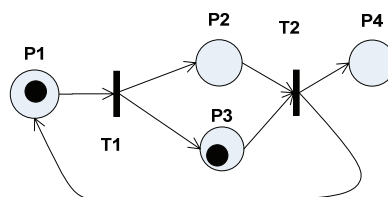


Figure 9. Example of a simple Petri Net

Types of Analysis

In this case, the constraints are: ordering of nodes, starting and terminating nodes, enabled transitions, events that fire transitions, capacity of each node (to store tokens).

As we can see, the works on this area are used mostly as notations for describing both process execution, and system state evolution. Because of the control-oriented nature of these notations, the most important types of analysis done here is the:

- Navigation of the sequence of activities in order to validate its consistency and the lack of dead branches such as it is done for [AALST, W.M.P.V.D., *et al* 2003, PETERSON, J. 1981].

- navigating the definitions of the hierarchy of inheritance of a single statechart [JIN, Y., *et al* 2004] or scalechart [BOSWORTH, R. 2004] in order to verify the consistency of such hierarchy.
- simulating the behavior of multiple instances, in order to verify the synchronization mechanisms for accessing shared resources [DAVID, R., *et al* 1997].

In all these cases, the internal actions, and states, are not actually covered but the reasoning is done in terms of the coverage of the set of nodes, and the convenience on the arrangement of transitions in order to satisfy this goal. No transition should carry the system to a state where it is not able to make any further transitions but when in a terminating node. On the other hand, no state should remain uncovered (not executed) if it is actually useful. As most practical systems contain very complex automata, an automatic analyzer is normally required.

Domains of Interest

Most of graph notations address the needs of modeling either reactive/concurrent or workflow/network-type systems. In some cases concurrency is allowed, and the analysis may therefore also include issues about resource usage: safety and liveness conditions. This happens because when automata communicate and interact, complex behaviors appear. The most common issues of analysis are, thus, mutual exclusion (no two instances of a process may access the same resource instance at the same time), serial access for all instances (for a given resource instance, we should guarantee that the instances of a process are serviced sequentially), and fairness (no instance can starve for a resource, it should be given access at a certain moment).

4.2.2.2 Reasoning with Graph-based Notations for Structural Modeling

We selected FOL as the expression mechanism. However, one of the problems of the pure formal methods is that they are type-less in practice [MONIN, J.-F. 2000], which means that a large number of constraints cannot be actually enforced. Although this is not critical for behavior-oriented notations, it is for structural modeling. We consider that there is still the need to invest some effort on the domain of structural modeling in order to integrate the use of the class and object types.

On the information systems (data) track, there are many works on modeling the structure of the information processed by a system, also known as domain models. As Diskin says: *“there is a long-time tradition of using two dimensional graphical images (we will call them schemas) for presentation of data. In artificial intelligence (AI) it is connected with the concept of semantic network. In the database area (DB) the tradition is usually formulated under the name of entity-relationship (ER) data model...”*

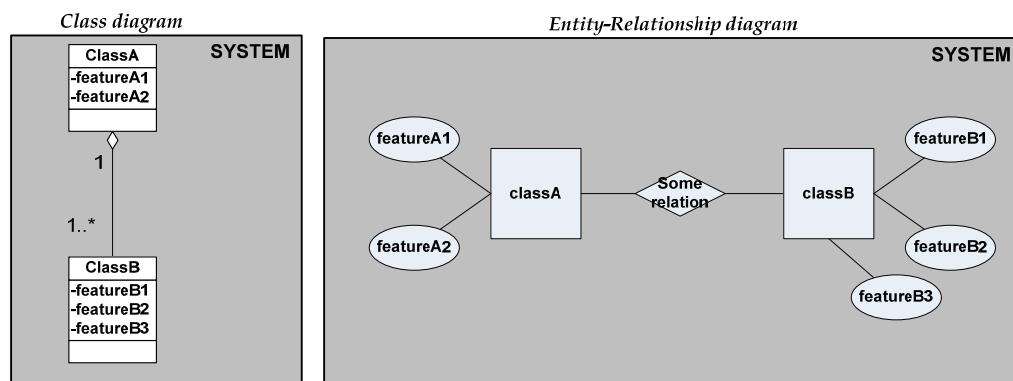


Figure 10. Static structure diagrams: UML class, E-R (Entity-Relationship)

We can identify two big groups of notations, those that represent sets and those who do not. On the set-oriented track, the most important of these works are those related with Entity-Relationship Diagrams [CHEN, P. 1976], Class Diagrams [OMG 2005c], and dataflow specifications [YOURDON, E. 1988]. These notations are normally supported by a relational algebra or some other approach that permits working on the entities that make up the data.

Association Represented by the Static Structure

Structural diagrams such as E-R (Entity-Relationship) diagrams and UML class diagrams represent the elements that remain constant in the evolution of the structure³⁴.

E-R and UML structural diagrams as defined as showing the “static structure of the system”. On the other hand, the behavioral diagrams show the dynamic part of the system. The constraints are used to declaring the semantics of an element of the UML specification; in practice, constraints are used to tie together structure and behavior and specify in a detailed manner what a well-formed system is.

UML Class diagrams consist of classes, relationships and associations. A UML relationship is defined as “*an abstract concept that specifies some kind of relationship between two elements*” [OMG]. Relationships and associations are very complex primitives that allow declaring all sorts of properties and interrelationships among classes such as inheritance, specialization, abstraction, and dependency, among others. An association “*declares that there can be links between instances of the associated types*” [OMG]. Classes correspond to object-oriented types.

An association is a named link connected to two or more classes. It transports much of information about the instances of the intervening classes: **roles** of each, **limit on cardinalities** for each, **direction/ navigability**, **aggregation/composition**, **constraints** for each, and many others.

However, the elements just mentioned above are only invariants or constraints that should be respected. Therefore, even if they are true for all actions done by the system, they are not useful for specifying an action performs concretely.

Types of Analysis

We identify three lines of work:

- navigating the structure of the classes that constitute the information system specification (either for domain, analysis or design), in order to verify properties on the relationships among data entities [BRUEL, J.-M., *et al* 2000, LISKOV, B., *et al*, WARMER, J., *et al* 1999]
- combining/separating entities in a flow [WARD, P. 1985, YOURDON, E. 1988, Hatley, 1988 #240].
- navigating the definitions of the hierarchy of inheritance of a single class in order to verify the consistency of such hierarchy [LISKOV, B. and WING, J.M. 1993, ROYER, J.-C. 2004].

There are a few works on this area that try to overcome the difficulties enumerated by [SHIMOJIMA, A. 2004]³⁵. There is notably the work of Diskin on arrow-based notations [DISKIN, Z. 1995] to support more precise conceptual modeling.

³⁴ In this discussion, we will only address UML structural diagrams. The reader is invited to do the analogy for E-R diagrams

³⁵ Shimojima makes a long presentation of different problems. For the sake of space, we do not include them here.

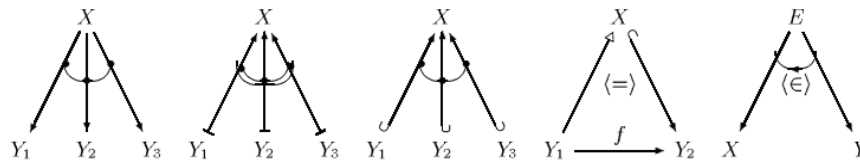


Figure 11. Arrow diagrams integrate additional semantic information. The decorations add some constraints over the functions and relationships (the arrows). From [DISKIN, Z.] p. 6.

Diskin was the first to demonstrate that E-R diagrams are perfectible by adding more graphical information. His approach uses arrows and applies to conceptual (static) modeling and is based on Category Theory. The solution is to provide schemas with an underlying graph-based logic so that graphical images themselves can be regarded as formal specifications.

The Diskin approach is called **arrow thinking** [DISKIN, Z. and KADISH, B. 1998] that contrasts to usual thinking in terms of sets and elements, and actually it constitutes the essence of category theory. The sketch logic is a logics of assertions about sets and functions. It deals well with the problem of heterogeneous view integration.

There exist other efforts on more complex and concrete problems such as software architecture. For example, in [OMMERING, R.C.V., *et al* 2001], they present a graph language to visualize design structure. They consider that an architecture is the specification of a design, where a design is an abstraction (or structure) of an implementation. This is an order of magnitude more complex than the approaches mentioned above. We can say analysis of architecture of systems is the final goal of this domain and, for the moment, it is very restricted because of the limitations on the underlying formalisms.

4.2.2.3 Reasoning with non-graph-based Notations for Structural Modeling

In the non-graph-based we can distinguish two main lines of approach: a) those oriented to support directly the set theory, and b) those based on the use of ideograms. The first one has a solid mathematical base whereas the latter is based on a more non-structured picture/drawing based approach that is near the domain of the modeler. Examples of formalized notations for set theory are the Venn diagrams, the Pierce diagrams, and the Euler diagrams. Most of ideogram-based notations are *ad hoc*, unstructured by definition but reusing a set of elements and techniques from the established approaches. However they are considered to be useful for interacting and reasoning in real, industrial contexts [CHERUBINI, M., *et al* 2007].

Diagrams in this family are normally used for navigating the structure of the classes that constitute the information system specification (either for domain, analysis or design), in order to verify properties on the relationships among data entities.

There are three basic diagrams for representing set theory: Venn, Euler and Venn Pierce diagrams [BARWISE, J. and ETCHEMENDY, J. 2002, PUMML 2002, SHIN, S.-J. 1995]. **Euler diagram** is a set-oriented topological representation where the modeler can represent subsets, disjoint set and set intersection. The crossing or the lack of it is the basic visual operator for doing this. **Venn diagram** adds shading to express the empty set. This avoids the ambiguity present when only the crossing of boundary sets is used. **Venn-Pierce diagram** adds existential operators in order to indicate the non-emptiness, disjointness, and other properties of the sets and among sets.

These three notations were severely limited and their use as formal logical tools was limited to the very introduction of set-theory at school level. The fundamental problem with notations is that their syntax and semantics are intertwined deeply; this issue was solved in 1994 by Shin [SHIN, S.-J. 1995]. In order to solve this limitation, Shin found that a correct diagrammatic representation should have two levels of syntax: concrete (or token, the physical representation of a diagram) and abstract (or type syntax, the semantically important

spatial relations between syntactic elements). She demonstrated that diagrams can have the same logical status as sentential systems. This gave birth to a broad field of research known as Diagrammatic Reasoning (DR).

[SHIN, S.-J.] built two logical notations –the Venn-Pierce diagrams I and II—that corrected the problem in notations from Euler, Venn and Pierce. Venn-II is composed of an underlying Venn diagram, and of a monadic language Lo used to write the constraints or **x-sequences**. These x-sequences can be used to visually specify properties about the quantification of the sets. She demonstrated that the expressive power of her language Lo is equivalent to a textual FOL.

After Shin’s breakthrough, a number of proposals of enhanced diagrams started to emerge; namely, Euler-Venn, Spider, Constraints, and others. Spider diagrams [HOWSE, J., *et al* 2001, STAPLETON, G., *et al* 2004], Constraint diagrams [FISH, A., *et al* 2005, GIL, J., *et al* 2001] and Contract boxes[GIL, J., *et al* 1998, KENT, S., *et al* 1998]. They all adapt the Venn-Pierce diagrams of Shin for representing constraints of object-oriented analysis models of IT systems (mainly class diagrams and object diagrams [BOOCH, G., *et al* 1998, OMG 2005c]), and by correcting the two main limitations of Shin’s approach: the pure monadic nature of the language and the lack of constants and function symbols.

The Euler/Venn diagrams are also based on Euler diagrams but are similar to Venn-II. These diagrams introduce the use of constant sequences [STAPLETON, G., *et al* 2004] that add specific-instance information.

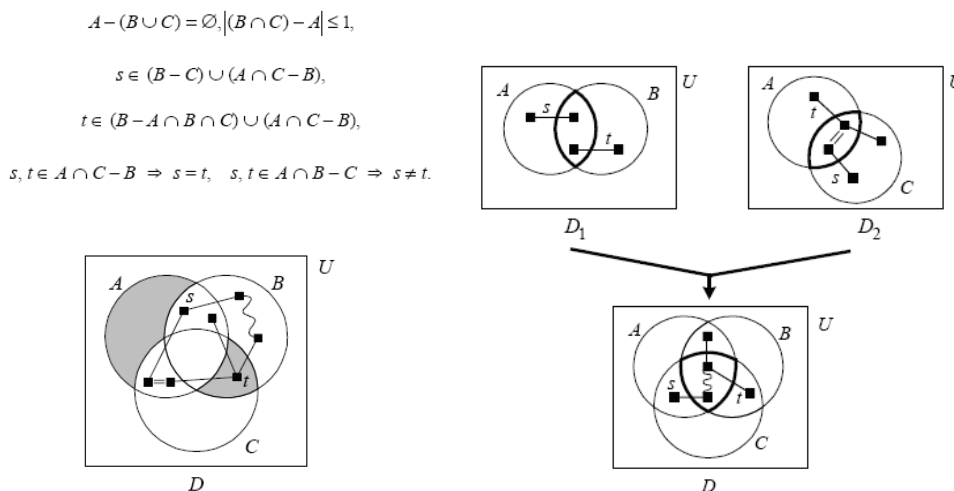


Figure 12. Spider Diagrams, from [HOWSE, J., *et al*, STAPLETON, G., *et al* 2004]

The Spider Diagrams permit describing the existence of elements[HOWSE, J., *et al*, STAPLETON, G., *et al* 2004]. Spider diagrams are also based on Euler diagrams but use graphical symbols to denote the existence of elements. By allowing lower and upper bounds on the cardinality of sets, spider diagrams are more expressive than Venn-II diagrams. The language ESD is a monadic predicate language that extends Lo . An example is shown in figure 12.

The definition of the elements of a spider diagram is the following:

“A contour is a simple closed plane curve. Each contour is labeled. A boundary rectangle properly contains all contours. The boundary rectangle is not a contour and is not labeled. A basic region is the bounded area of the plane enclosed by a contour or the boundary rectangle. A region is defined recursively as follows: any basic region is a region; if $r1$ and $r2$ are regions then the union, intersection and difference of $r1$ and $r2$ are regions provided these are non-empty. A zone is a region having no other region contained within it. A region

is shaded if each of its component zones is shaded. A spider is a tree whose nodes (called legs) are straight lines. A spider touches a zone if one of its feet appears in that region. A spider is said inhabit the region which is the union of the zones it touches. This union is called the habitat of the spider.”[STAPLETON, G., et al 2004]

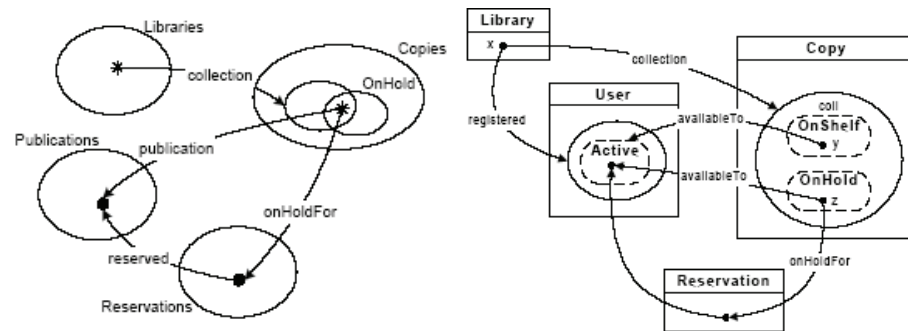


Figure 13. Constraint Diagrams [GIL, J., et al 2001]

Constraint diagrams [GIL, J., et al 2001] are an extension of spider diagrams. The objects in spiders diagrams are oval contours (representing sets that overlap or intersect), points (representing members of sets) and spiders (linked points representing relationships) and quantifiers. Only points and spiders are quantified. Quantifier type is handled by drawing points in different ways: “point” for “there is at least one point”, “star” = “for all”. Quantifier ordering problems are dealt with by labeling quantifiers with numbers. No generalization/specialization is supported as all conditions must be made explicit but for set membership.

4.2.3 Limitations

4.2.3.1 The Common Critical Problem: Scalability

In the case of notations based on the set theory, although they are popular because they are used as a didactic tool to introduce mathematical notions to children, their use is restricted mostly to this domain. This is because the use of these notations consume large amounts of space both for *a*) more than a few instances and *b*) a small number of sets (showing more than five overlapping sets in an Euler diagram is a tough topological problem, as shown in figure 14). Other examples can be found in [COMBINATORICS 2007]



Figure 14. a) Simple and symmetrical Venn diagram with four contours. b) The simple symmetrical Venn diagram of five contours. c) Adelaide, a symmetrical Venn diagram of seven contours [GIL, J.Y., et al 2000]

This problem was partially addressed by the spider diagrams and the constraint diagrams by showing the relations between sets and their elements/objects [GIL, J., et al]. This can be seen from the concept of region; this notion simplifies the notation for overlapping sets, as shown in figure 15.

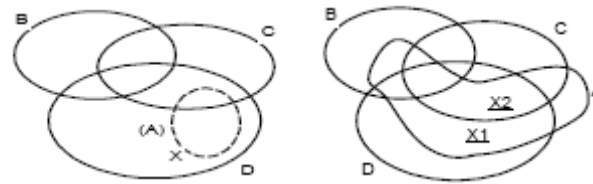


Figure 15. Use of regions in order to map complex configurations. The syntax of regions (left) is simpler than semantics in Euler-Venn diagrams (right) [GIL, J.Y., *et al*], Page 126.

A concrete example is shown in figure 16. It expresses that “*for all x in $(B-A)$ ”, the relational image of x under g is A , and there is a y in $(A-B)$ whose relational image under f is an element of C . The dashed contour labeled x denotes the set obtained by “projecting” the set A onto the context $(D-B)$.*

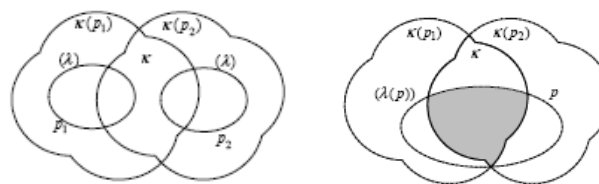


Figure 16. Intersecting contexts in Venn-Euler Diagrams using regions [GIL, J.Y., *et al* 2000], Page 126.

The use of color in spider diagrams is found to be cumbersome; at least one author found that when using color for the quantifier type this severely limits the way in which color can be used elsewhere in the diagram [WINTERSTEIN, D., *et al* 2004]. Thus, instead of solving a problem for expressing constraints, it becomes a highly complex topological problem.

On the other hand, As [PRICE, S. 2004] suggests that “*if attention focus is facilitated and guided then the consequences of the amount of perceptually available information may be counterbalanced in terms of detrimental effects on learning.*”.

4.2.3.2 Node-link as stereotype

Any customer can have a car painted any color that he wants as long as it is black
Henry Ford

Containment is a fundamental contextual relationship among entities. It can define structural but also behavioral binding among the contained entities and the container one. Most diagrammatic and visual notations use it. Nevertheless, the use is so common that the default semantics is not analyzed extensively, at least not in a way we are able to find in the bibliographical research.

The default technique for containment is based on the use of closed curves (especially for set theory and for automata) and closed rectangles (for structural notations, including E-R and most UML structural diagrams). Closure is essential for representation as it enables symbolizing contours and regions in diagrams, as shown in [TUFTE, E.R. 1997, WARE, C. 2004].

Figure 17 illustrates the problem. On the left side, figure 17.a, we can see how the set theory uses closed curves around dot elements. Next to that representation, the UML class diagram represents the mapping of the function via an association. Finally, the resulting representation in terms of pairs (relations in set theory) is shown as a table.

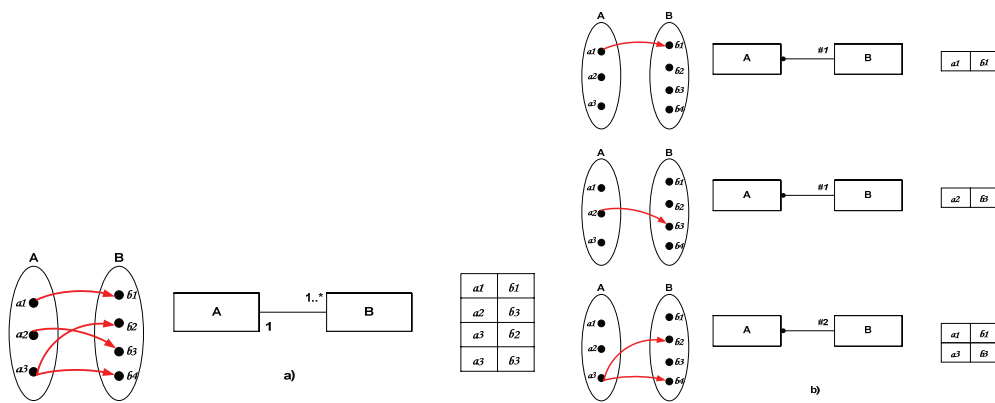


Figure 17. Node-link interpreted as a table. On the left, the relational approach, adopted by E-R and class diagrams, where the observer has the whole knowledge. On the right, the relativist, systemic approach we use.

The arrow-based notation and the tabular representation make both evident that an omniscient observer/modeler is present. However, in our systemic approach we need to model the world in a relativistic way, one where we model the object’s viewpoint. This is illustrated in figure 17.b.

4.2.3.3 Snapshot as the Way to Represent Structural Changes due to Behavior

The languages that support operational semantics like CCS, CSP and Pi-calculus allow you to model stackable data structures, storing and retrieving data in a sequential or random way. This kind of representation has a major inconvenient: it focuses on the data-structure representation, and on the basic algorithmic problems. However, for a generalized approach, the chosen data-structure should not weigh more than the actual domain modeling.

To give an intuition of structural change, let us model a system that has one instance of **Person** at the beginning (**@pre**) of an action **op1**; at the end (**@post**) of the same action, the system has two instances of **Person**. Figure 18 shows a UML simplified model to represent this scenario using a class diagram, a use case diagram, an activity diagram, and a statechart. The figure 18.a of corresponds to the pre-condition (**@pre**) whereas figure 18.b corresponds to the post-condition (**@post**).

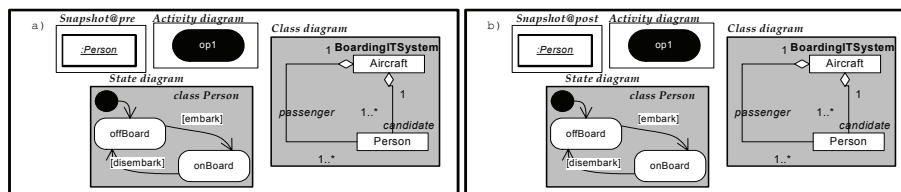


Figure 18. UML object, activity, state, and class diagrams for (a) before and (b) after action op1. The effect of change performed by operation op1 is not evident

Note that no change is noticeable among both sides of figure 18. This is because most of the information transported by the UML associations is static. This means that the class diagram remains unchanged. Because behavioral diagrams are not structural, they do not reflect the changes in information content. However, from the description of the scenario, some information should change in time. UML object diagrams are useful to express static snapshots. Nevertheless, as illustrated in sections 6.2 and 6.3, it is up to the modeler to add the causal information among snapshots, making them not very useful as specification artifacts.

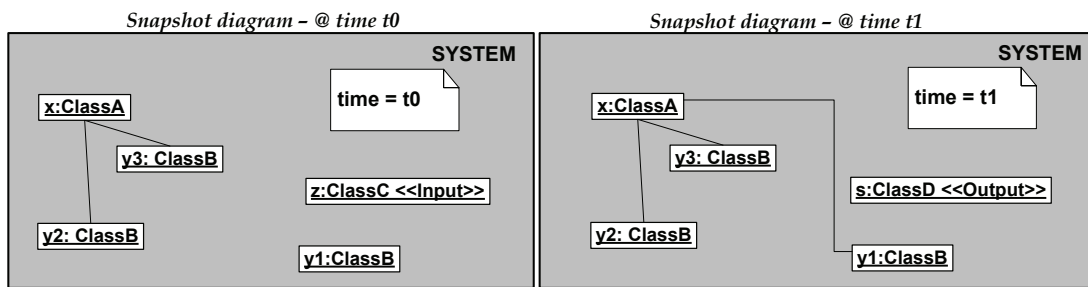


Figure 19. Static structure diagrams: UML object diagrams (2 snapshots)

Figure 19 shows the object diagrams that represent the expected structural changes. However, object diagrams themselves do not make explicit neither what has changed nor why the change takes place. It is necessary to relate figure 19 to the diagrams in figure 18 in order to understand the latter. Isolated diagrams do not express everything that is required.

4.2.3.4 Representing Change Diagrammatically

As [WINTERSTEIN, D., *et al*] puts it: *“Diagrams in textbooks are necessarily static... diagrammatic reasoning on computers need not be just a straight conversion of diagrammatic reasoning on paper”*. However, he proposes to work on animation of diagrams for representing and reasoning about temporal relations. We choose to deal with the same problem in a different way.

Several works, such as [BIENVENIDO, J.F., *et al*, NARAYANAN, N.H., *et al*] deal with the problem of representing behavior by creating a mental image of the execution of actions over elements. [BOGACZ, S., *et al*] even suggests that forecasting in complex systems such as the weather changes is done by animating static spatial information, however this requires a great deal of effort.

As [PRICE, S. 2004] demonstrates, the animation of (static) diagrams does not always fulfill the expected results. The additive way of displaying information, where an additional component of information is exposed at the same time as the previous, is less confusing than the substitute one, where a currently displayed section becomes masked again when a new component is displayed. This means that the snapshots are not necessarily the best way to show evolution of a system as it is done in UML.

As for problems with animation, [WINTERSTEIN, D., *et al* 2004] asserts that *“its main drawback is that it is not suited to being printed,... except as cumbersome comic strips where the simplicity of the representation is lost”*, a logical consequence of what we said concerning the additive nature of this way of representation, and of losing the local indexing mechanisms.

The levels of complexity of a temporal representation like the one just presented also has a negative effect on observers, if we interpret the Price’s work correctly. This work suggests that the amount of perceptually available information is problematic for comprehension, due to resulting reduction in focus of attention.

Understanding and making appropriate information links is often more problematic with the substitute animation than with the additive animation [PRICE, S. 2004]. The substitute animation resulted in clearly segregated sections.

“Animation may show dynamic changes more explicitly but without specific reference to linked aspects this makes integration difficult and adds to the memory load needed to remember information across representations”

After [THOMAS, L., *et al*] the human being that is successful in diagrammatical reasoning uses as strategy that goes from the static model to the dynamic one. It analyses the static model of a system, and the connections among elements; then, she integrates the

information, hypothesize lines of action, **“and finally construct a dynamic mental model by mental animation”**. Our notation should then support and facilitate part of this information integration process.

4.2.4 Discussion

Visual representation is a rich, complex and broad domain of study. Nevertheless, research in visual representation of formal approaches (including contracts) is an area yet to develop [GLASGOW, J., *et al*]. For example, [COX, R., *et al* 2004] shows that very popular visual approaches such as set diagrams, and networks are particularly poorly understood by computer science students, with very high rates of misclassification and misnaming: a totally unexpected result. This is confirmed directly by [CHERUBINI, M., *et al* 2007]. Other works such as [ARGAWAL, R. and SINHA, A.P. 2003] strongly suggest that there is a misunderstanding about the real mechanisms used by modelers to integrate information from system models.

One part of the problem is the lack of formalism in current notations: Having a better logical support may resolve the problem. Even formalized notations such as Petri Nets allow the modeler to freely link predicates among diagrams. This suggests that contextual-aware models may avoid this kind of problem

However, as noted by [SHIMOJIMA, A. 2004], the creation of a visual notation that supports reasoning about a logical domain has several limitations:

- Similarity on the *problem domain vs. problem-solving power*, which requires expressive generality;
- *Expressive richness vs. expressive flexibility*, which means that the graphics are good at expressing conjunctive information but not at expressing disjunctive information.
- Finally, *graphics limit abstraction*, which enables inference processing.

[ARGAWAL, R. and SINHA, A.P. 2003] and [COX, R., *et al* 2004] show that the class diagrams are particularly well understood and used in the arsenal of diagrammatic tools. The simple predicates, even in the form of tables are also very easily understood. This, lead us to create a non-pure diagrammatic system, known as a heterogeneous system in the Diagrammatic Reasoning community. The work on pure diagrammatic approaches, such as Hammer’s and Shin’s prove that sound and complete notations can be built, but also that the resulting notations are too difficult to use. Regarding usability, [WINTERSTEIN, D., *et al* 2004] states: **“By extending diagram systems on purely logical criterion without considering ease-of-use issues, the final systems lost the very qualities that make diagrams attractive”**.

Some notations, such as Petri Nets, resolved the problem of typing by exploiting visual dimensions such as the color. Other notations, such as E-R diagrams and Arrow diagrams are extremely specialized, enabling the creation of first-order logic (FOL) specifications via concise graphical schemas of equal logical rigor yet more evident and observable, thus taking advantage of the power of representation. Nonetheless, there are also tradeoffs: the Petri Nets are not intended to show instances of data, and the structural diagrams cannot easily cope with values.

One interesting point is that visual representations for relationship dynamics, or behavior, are almost non-existent. Pi-Calculus partially solves the problem, but because of its operational nature, it cannot be considered as a possible solution for our declarative specification artifact. Table 6 resumes very succinctly certain features of each diagrammatic family and ranks their support for reasoning.

Table 6. Comparison of different notations and their relative adequacy to representing behavior and/or structure

	Graph-based	Set theory	Ideogram
Goal	simulate the behavior of the automata	Finding inconsistencies among types, hierarchy of types or combination of dataflows	used to reason about a system or situation
Logics	a form of operational semantics	Set theory, Relational algebra/tables	Ad hoc
Formatting	very structured, sequential descriptions	Very structured, relational description	Substratum
Expressive capacity	properties can be described as automata or as typed automata ³⁶	Properties can be put on types and on relations among them	make evident the relationships among the elements
Usability	Easier to understand than process algebra	Easy to understand	some notation features (e.g., elements, colors, caps) that make more relevant one or several of the items on the diagram
Typical use scenario	Addresses specialists	Addresses non-specialists	Use & throw-away
Format	Node-link	Node-link	Mainly node-link

Our research work does not analyze the cognitive dimension, but we identified the elements that are essential in order to define the design criteria for our notation.

Table 7 summarizes the expressive capacities of the notations as specification artifacts. The graphical approaches studied in this chapter present some limitations to represent the three aspects in the shadowed columns, namely:

- The integration of static and dynamic aspects, in order to represent change
- The expression of constraints, including pre-conditions and post-conditions
- The capacity to represent multiple instances and collections

Our visual notation should then be designed to avoid these limitations, because these aspects are fundamental for representing the elements necessary to build contracts.

³⁶ The Colored Petri nets (CPN) represent different types with different colors. For more information, please refer to [DAVID, R. and ALLA, H.H. 1997]

Table 7. Comparison of the different approaches for contractual specification

	Underlying theory (supports machine reasoning)	Easy to use	Static	Dynamic	Integration Static/Dynamic	Pre- & Post-condition	Supports Multiple Instances	Data Mapping	When to use	Comprehensible (supports human reasoning)	Analyzable (supports algorithmic reasoning)
Graph based	Process algebra	Yes, when each instance of the process presents same behavior Yes when not required much data information	Only for Colored Petri Nets	Causality Observability	No	No	Yes, but only in the form of tokens. Tokens can be typed when color is used.	No	No complex data dependencies When model does not require complex structural rules, not even for integrity	Yes for dataflow and workflow representations	Yes for creating operational traces that can be used to validate operational properties.
	Entity-Relationship Diagram	Yes, even for complex data types and data dependencies	Yes	No. Only invariants are represented	No	No	Yes, but only as invariants or typing information. Rich information	Yes. This is its main purpose	When complex data types are required When complex data dependencies are required, including integrity rules	Yes	Yes, for hierarchy of types and static validation of integrity rules (as it does not include dynamics aspects)
Set-theory-based	Set theory	For reduced number of instances For data types with low complexity	Yes	No	No	No	Yes, but only with basic existential operators within the set. No typing is possible	May be	For problems with low levels of complexity For illustrating simple relationships	Yes for small examples	Not for systems modeling. Yes for highly mathematical, abstract problems
	Spider Constraint diagrams	/			No	Yes, in the form of a 3-D contract	Yes, but only with basic existential operators. Typing is possible				
Ideogram-based	Ad hoc	Yes. It is ad hoc	Yes. It is ad hoc	Yes. It is ad hoc	No	Ad hoc	Ad hoc	Yes. It is ad hoc	When a shared understanding is required	Yes. It is ad hoc	Not at all

PART II - Visual Contracts for System Modeling

In this part, we introduce the Visual Contracts and illustrate their use as specification artifacts.

In Chapter 5 we get acquainted with SEAM and discuss the modeling of context. We identify several heuristics that we use to build the notation for Visual Contracts.

In Chapter 6, we introduce Visual Contracts and illustrate their basic syntax and semantics. We introduce the elements of the notation and explain how to interpret a Visual Contract, and how to compose a specification.

In Chapter 7, we illustrate the semantics and pragmatics of Visual Contracts. We explain set-associations in detail, including the operators, algebra of collections, and how to interpret the context of existence.

In Chapter 8 we explain in detail the notation for Visual Contracts. We present the primitives of the notation, the well-formedness rules of Visual Contracts, and the metamodel of Visual Contracts.

In Chapter 9 we explain how to translate Visual Contracts to Alloy, and how to perform the verification and validation of our specifications.

It is necessary to study not only parts and processes in isolation, but also to solve the decisive problems found in organization and order unifying them, resulting from dynamic interaction of parts, and making the behavior of the parts different when studied in isolation or within the whole...

[Ludwig von Bertalanffy –
General System Theory: Foundations, Development, Applications](#)

5 Systemic Modeling of Systems

Current approaches to system modeling do not allow us to create specifications that are systemic in the way SEAM conceives it. It is then required that we develop a better comprehension of the systemic approach of SEAM, in order to establish clearly how to develop systemic contracts.

This chapter elaborates on a number of heuristics that we discovered and used during this research work. We first address the need for modeling the context in each of the system models. Then we introduce SEAM, the methodology of our group. Once the scope is set up, we present the heuristics and describe how they satisfy our needs, as well as their implications.

5.1 SEAM as a System Modeling Approach

SEAM is based on both the Reference Model of Open Distributed Processing (RM-ODP) and the theory of living systems [MILLER, J.G. 1995]. Our modeling ontology is inspired by the ISO/ITU standard RM-ODP [ISO/IEC 1995-1996] that defines how to model systems. A formal description of RM-ODP our group developed is available in [WEGMANN, A., *et al* 2001]; a longer version is [NAUMENKO, A. 2002].

We define the model elements system and environment³⁷. The “system” can be defined as a whole (i.e. only its externally visible functionality is described) or as a composite (i.e. its composition is described). One cannot design a **system of interest (SoI)** without taking into consideration the immediate “environment” that interacts with it. The “**supra-system**” of the SoI is “the next higher system in which the SoI is a component. The immediate environment is the supra-system minus the SoI itself” [MILLER, J.G. 1995].

A **system** is described as a hierarchy of **organizational** and **functional levels**. If a system is represented as a whole at a given organizational level, then it is represented, at the next organizational level, as a composite (i.e. showing its sub-systems). The “**organizational level hierarchy**” is useful to capture system components (e.g. an IT system made of software components). The “**functional level hierarchy**” captures different levels of detail in the functionality of the systems. For example, an action might be described as a whole in one level of functionality and as a composite in the next. All these concepts are informally defined in [WEGMANN, A., *et al* 2005] and formally in [LE, L.S. and WEGMANN, A. 2005].

At each level, the system can be described either **as whole**, an extended finite state machine (EFSM) that communicates with the environment of the system, or **as composite**, a set of extended finite state machines that communicate with each other (the initial EFSM is decomposed) and with environment of the system.

5.1.1 Main Modeling Concepts

Our ontology defines two kinds of objects: **Working Objects (WO)** and **Information Objects (IO)**. **Information Objects** are the internal representation of real-world objects (**working objects**) and constitute the information viewpoint. *Information objects* describe information that the system has about itself and about its environment. The system’s information is modeled with the “**state**” of information objects. The set of properties of a

³⁷ In RM-ODP, the term “system” designates an entity in the universe of discourse and not a model element in the model. For the sake of simplicity, in this paper, we consider that the term “system” designates the representation of a “system” in the model. So “system” is a model element. This paper does not address any issues related to the universe of discourse.

system and their interrelationships make up the information specification (IS) of the system. System functionality is described with “*information objects*” and “*actions*”.

Our work addresses specifically the need to create functional specifications for the information viewpoint.

For a given functional level in the hierarchy, the state machines work in parallel, and several occurrences of the same state machine may concurrently exist. These state machines are Extended Finite State Machines (EFSM). Each EFSM work either on information it knows (*information objects* or *properties*), or in collaboration with concrete real subsystem (*working objects*).

The finite machine is extended in the sense that local variables for each machine may hold details about the history of the machine; each state may require inputs (events, data) as well as produce outputs (events, states). Also branching is allowed on state transitions for splitting into several sequences of actions. The modeling is based on the use of the process concept that is mapped onto systemic actions, which can be local or joint.

5.2 Modeling Heuristics

5.2.1 Definition of Context

We base our work on the Merriam-Webster [2005] definition of context

Main Entry: **¹con·text**

Function: *noun*

Etymology: Middle English, weaving together of words, from Latin *contextus* connection of words, coherence, from *contexere* to weave together, from *com-* + *texere* to weave

1 : the parts of a discourse that surround a word or passage and can throw light on its meaning

2 : the interrelated conditions in which something exists or occurs : [ENVIRONMENT](#), [SETTING](#) <the historical *context* of the war>

We take the second definition, and we apply it to system and system models. Our goal is the representation of contextual information in graphical system specifications. As a consequence, in this chapter we elaborate an interpretation of both: what “*something*” is and what the “*interrelated conditions*” are when this definition is applied to graphical system specifications.

The term “*something*” can be replaced by the concepts defined in our modeling ontology [LE, L.S., *et al* 2005, WEGMANN, A., *et al* 2005]. We therefore need to show the “*interrelated conditions*” (in terms of system, environment, information object, action, state) in which the system, environment, information object, action, state exist. Our main focus in this research work is system specification. Hence, we mainly analyze the relationships between the concepts: system, information objects, states and actions. We briefly mention the relation between the system and its environment but this is the topic of future work.

The most obvious “*interrelated condition*”, as stated in the Merriam-Webster definition, is between the system and its information objects and actions. All actions and information objects are within a system. In the notation, the information objects and the actions should be

surrounded by a rectangle that represents the system to which they belong (as shown in Figure 20 for the Information Object x of s).

The previous contextual definition is what most methods and notations support and focus on, as we have already discussed in previous chapters.

In this section, we present some of these modeling principles and their impact on the notation.

5.2.2 System / Environment Complementarity

Specifying a system fundamentally involves describing two distinct domains: the real world and the world built in the model.

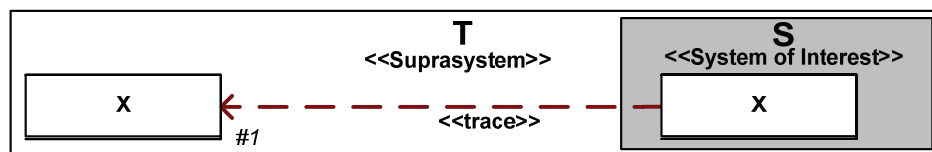


Figure 20. The objects in the model track the real models in reality, and trace one part of their properties and behavior

Hence, one “interrelated condition” is the relation between the system and its environment. A system has information about itself and about its environment. For example, in the video store example, the **PORT** has some information about the physical video that exists in its environment. So, one of the “interrelated conditions” in which an information object exists is the relation between the information object itself (belonging to a system of interest) and what it represents (in the environment of the system of interest).

In Figure 20, τ represents the supra-system of s and x shown in the left part of τ is in the immediate environment of the system s . The information object x in s represents the knowledge of s about x in T . This is represented by the `<<trace>>` relationship. A value of 1 as cardinality of the relationship shows that there is one instance of x in the real world τ that is mirrored by an instance of x in the system s . A more systematic analysis of the relationship between a system and its environment is part of our future research.

We capture the necessity to relate actions, information objects and states to a system in which they exist and the necessity to relate a system to its environment as the “**system / environment complementarity principle**”.

5.2.3 Behavior / State-Structure Complementarity

An action changes the state of one or more information objects. Quite often the action’s identifier makes implicitly references to the information objects that change state. For example, the modeler can guess that an action **rent** in a **video store** refers to a *video* and a *renter* because she knows the meaning of the word *rent*. With this knowledge, the modeler can guess the relationship between the elements in the diagrams (e.g. in UML, the action **rent** is shown in an activity diagram whereas the video concept is shown in a class diagram). This is not sufficient as the concepts used can be defined in multiple ways. For example, it is unclear if the **rent** is related to some payment or not. We can eliminate this ambiguity by making explicit, in one diagram, the “interrelated conditions” between the actions and the information objects involved in the actions. So, one of the “*interrelated conditions*” in which an action exists is the set of information objects that are modified by the action.

This relation between actions and information objects can be further developed. When a modeler represents an action, implicitly she is referring to a change of state of information objects. Vice-versa, when a modeler represents the change of state of information objects, she is referring to an action. This is known as the **state/behavior duality**. This duality leads

the modeler to often consider that modeling either the action or the state change is sufficient to specify the system. We claim that, if we want to make the context explicit, we need to model both. So, the “interrelated conditions” in which an action exists include the states (before and after the action) of the information objects involved in the action. Vice-versa, the state is related to the actions that consumes them or modifies them.

Figure 21 represents an action **A** that modifies the state of the information object **x** and created the object **y**. The arrow between the states **s1** and **s2** illustrates the state transition resulting from the execution of **A**. To show the relation between the actions and their effects, we need a way to relate them. This is done via the change of cardinalities, and the change of state of such instances: there are conditions that trigger some processing, and processing changes the state of information objects. In this case, the presence of one instance of **ParA** in the context of action **A** is interpreted as the **eventA**; as a consequence, one instance of **x** is processed: when the condition **eventA** is fired up, all the clauses that include it are triggered; one instance of **x** goes from state **s1** to state **s2**, changing the corresponding cardinalities. A more detailed explanation is made in section 5.2.3.3.

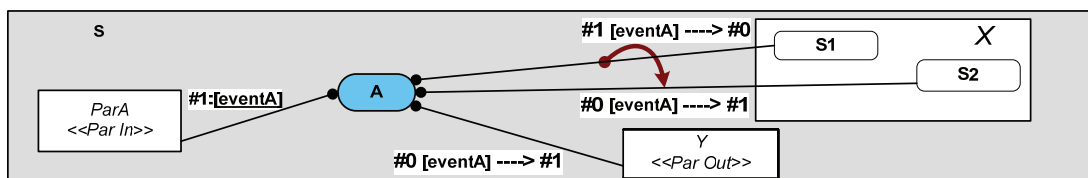


Figure 21. The action changes the state. This change can be seen as changes in the structure (cardinalities) and state of the objects themselves.

5.2.3.1 Contexts of Existence

Everything exists in a context, and is relation to that context. That context is normally described in terms of an omniscient **observer**. Nevertheless, in systems thinking [LE MOIGNE, J.-L. 1993] there are two principles that are incorporated into our approach: the **teleological principle** and the **interaction-based view of the world**. The *teleological principle* asserts that there is always a purpose for entities we model; the *interaction-based view of the world* indicates that entities should interact. By splicing those two principles, we can say that it is essential to model entities in a relativistic way, because an entity can only fulfill its goal when it is related to or “known by” other entities (in the case of data entities) or “affects” other entities (in the case of action entities). It is also essential to model the lifecycle and scope of the objects, such as it is already implicit in our hierarchical approach. Therefore, we adopt a modeling view from the perspective of each object in the context of the interactions where it participates.

The context of existence of an entity defines the context where the entity “lives”. In other words, the spatial context defines:

For a data item:

- Where it exists
- Where the actions can create/ delete it
- More importantly, where the actions can read it/modify it.

For an action item:

- Where its impact is observable
- Its inputs
- Its outputs
- What conditions trigger it (state, inputs)

It is clear that there is, once again, a dichotomy data/action that cannot be resolved using standard specification techniques.

5.2.3.2 Behavior as the Context of Existence of Objects

Some information objects live in the context of actions. We call this the context of existence of an information object. In figure 21 we see that action **A** is the context of existence of all the information objects shown. This means that none of the represented information objects survives after the action **A**. In Chapter 6, we see how we can represent that an information object can potentially exist during the whole lifecycle of a system.

As can be seen in figure 21, the local changes and the creation of subsets of objects all take place in actions. If the action is aborted, such changes do not take place.

5.2.3.3 Communication as the Context of Existence of Objects

We also define a special kind of information object: the **parameters**. *Parameters* are necessary to represent the communication through the boundary of the system or through action boundaries. The stereotype can be «**Par In**» or «**Par Out**» depending on whether it is an input or an output parameter.

This is done by an identifier (**eventA** in Figure 21), that designates all changes. It is also necessary to represent what triggers the action's execution (and how the action enables system exchanges with its environment). This is done by special information objects that act as parameters. The stereotype <<**par in**>> indicates that the information object is an input parameter for the system (<<**par out**>> would indicate an output parameter). In Figure 21, the fact that **parA** enters the system via **Param** (an input parameter) triggers the state change of **x** (rounded arrow from **S1** to **S2**) and the creation of **Y** (shown by the multiplicity change #0 → #1). All the changes are marked by **eventA**. One of the labels is underlined and this highlights what triggers the changes. We can see in Figure 21 that a UML class diagram, a UML activity diagram and a UML state diagram can be merged together.

5.2.4 Whole / Composite Complementarity

In systemic modeling, the modeling elements (e.g. system, action, information object) can be interpreted *as whole* or *as composite*. A modeling element as whole appears as monolithic and its internal structure is hidden for the modeler. A model element as composite exposes to the modeler the component elements and the way they are related. The whole is defined as the result of the composition of its components. The composition of the components can be understood as we know what the whole is. So, as system theory shows [LE MOIGNE, J.-L. 1993, WEINBERG, G. and WEINBERG, D. 2001], whole and composite are both necessary as they define the context of each other: the whole is part of the “interrelated conditions” of existence of the composite (and vice-versa).

In other words, it is because we can recognize the whole that we can see the components and vice-versa. For example, in a video store IT System, the action **Rent** can be understood because we implicitly know that such action includes getting information about the **renter** and getting information about the **videos** to be rented. Vice-versa, it is because we know the component actions **GetRenter** and **GetVideos** that we can imagine the existence of the composite action **Rent**. However, our goal is to make explicit the implicit information that is hidden in the diagrams.

In figure 22, we apply this principle to the action **A**. Action **A** as whole (the bubble on the top) is equivalent to **A** as composite, composed of the actions **A1**, **A2**, **A3** and of the constraints of execution between them. The special association symbol –made up of three lines– between **A** as whole and **A** as composite makes this equivalence relation explicit. In other words, **A** as whole can be substituted by **A** as composite. Some standard approaches in

process algebra like Petri Nets[PETERSON, J. 1981], CSP[HOARE, C.A.R. 1985], and CCS[MILNER, R. 1989] also support this way of modeling.

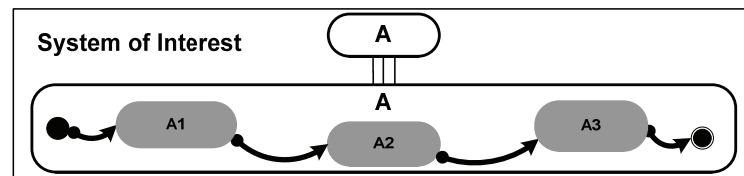


Figure 22. Action **A** is an action as a whole, Actions **A1**, **A2** and **A3**, plus the ordering constraints constitute the action “as a composite”

5.2.5 System Identity = Myself

We can reason about systems because we can see how actions change the instances. Therefore, it is fundamental to model the lifecycles of the instances (to understand which instances exist and when). A context of existence of an instance is the temporal frame within which an instance or a set of instances exist.

For instance, in figure 22, the action **A** as whole and the action **A** as composite define two functional levels in the functional level hierarchy. One interesting question is: what action is at the top of the functional hierarchy of a system? In other words, what is the action in a system which includes all the actions the system does? This is the action that corresponds to the **system lifecycle**. This lifecycle action captures the behavioral context in which all other actions exist. The lifecycle action starts when the system is created and ends when the system is dismissed (i.e. the action lasts from system “cradle to grave”). The system’s lifecycle action is at the top of the functional hierarchy. The added-value to model the lifecycle is, first, to force the modeler to think on how the system is created and how it is phased out. This can highlight critical issues in terms of system initialization or information retrieval at system phase-out. A second added value is the creation of a hierarchical approach by default; this explained as the *whole/composite complementarity*).

In figure 23 we can see that the lifecycle of the system **S** is equivalent to the set of all the actions that the system can execute (**A**, **B**, **C** and their execution constraints). Each action can be further refined. For example, action **B** is decomposed into actions **B1**, **B2**, **B3** together with their execution constraints. Nevertheless, in figures 22 and 23 we use the simplest execution constraint among the actions. This is important to note as real cases include more complex constraints such as loops, partial order, etc.

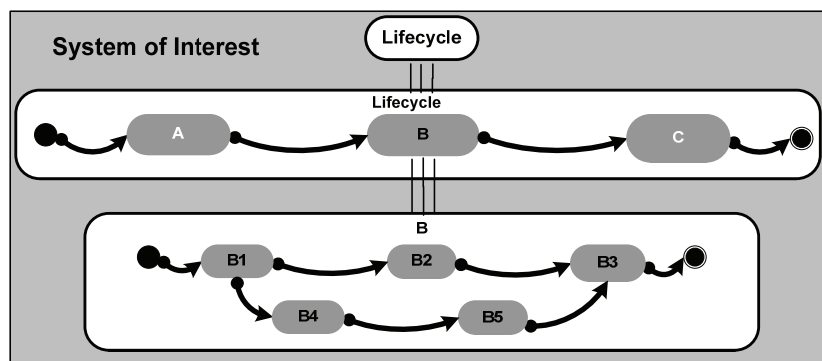


Figure 23. Lifecycle composition with actions

The same complementarity whole / composite of actions shown in figures 22 and 23 can be found on information objects. However, presenting this is out of the scope of this discussion.

We capture the necessity to relate whole and composite as the “whole / composite complementarity principle”.

5.2.6 Discussion

These principles permit the inclusion of contextual information in a graphical system specification. Contextual information has to do with the “interrelated conditions” in which model elements exist. Through our systemic approach, we showed that any system has a lifecycle, made up of actions that change the state of the information objects that exist in the system. These information objects represent information about a system of interest and about its environment. The relations between all these model elements need to be explicit if we want to model explicitly the context

We showed in our discussion that the actions and the information objects are bound to the lifecycle of the object in which they exist. So, instead of a universal ontology principle, we rely on the “lifecycle epistemological principle” that we define as “all actions and information objects exist in specific lifecycles (of systems and actions/information objects)”. To accept the contextual relations between the model elements requires being less strict in applying the Occam’s razor principle [SPADE, P.V. 2006]. We still aim to limit the number of concepts we use in the theory (for example in our method, we have 6 main concepts: system, environment, action, information object and state). However, we should not limit the analysis of the relationships between them to only the relations between information objects or relations between actions (as, for example, normally done in UML). We should keep the possibility to represent all relationships between actions, information objects and state. In other words: we claim that the Occam’s razor should not be applied because it eliminates the contextual information.

5.3 Summary

This chapter addressed the graphical representation of contexts in visual system specification. We showed that the two epistemological principles (universal ontology and Occam’s razor principle) can explain the way we currently structure graphical specifications. We believe that if we adopt the lifecycle epistemological principle (instead of the universal ontology principle) and if we do not eliminate the contextual relationships, we can obtain system specifications that exhibit the “interrelated conditions in which the model elements exist”.

In other words, we can have system specifications that make the contextual information explicit. It is worth highlighting that the Merriam-Webster definition of context can be easily interpreted from a systemic standpoint.

In practice, we identified four modeling principles that need to be adopted to model the contexts:

- System / environment complementarity
- Action / information object & state complementarity
- Whole / composite complementarity
- System identity and lifecycle

Unlike the analytical approach, the assumptions are that *a*) the modeler is not an omniscient observer, and *b*) systems are not always synchronized with the environment. This means not only that we should model different perspectives for each observer, but also that the exchange on the frontier of the system is a multimodal (real object vs. information object) active (not default and automatic) process.

The principles mentioned above allows us to integrate relativistic views of the system to the description of the action, establish the frontier between the system and its environment –in

order to map the system's knowledge of its environment—, and make explicit the lifecycle of all things that exist in the system.

With these principles, we showed how system states (configurations) can be graphically specified in the context of actions. This requires the modeler to make explicit the relations of the given action to the information objects it modifies and its relations to the parameters that enter and leave the system.

6 Visual Contracts

Successful systems development in the future will revolve around visual representations
David Harel – “Biting the Silver Bullet”

As seen in Chapter 2, current system modeling techniques carefully avoid the dichotomy among behavior and structure, defining different primitives for the one or the other and obtaining completely independent models.

In this chapter, we deal specifically with how to introduce elements of Diagrammatic Reasoning [GLASGOW, J., *et al* 1995] in the form of contracts that serve to describe actions and that also support reasoning (about single actions and about compositions of actions). In this chapter we introduce the notion of **visual contract (VC)**. First, we address our need to specify systemic actions via the use of contracts. Second, we present our diagrammatic language. Finally, we explain briefly the composition and use of VCs.

By using these models the designer can establish the state of the system after a series of actions are executed, she can also synthesize new, equivalent models from the originals. This requires the creation of a logical background (rules, primitives and semantics) for the visual notation that guarantees that the models are correct individually, as well as consistent with each other. The modeler should then be able to create a functional model.

6.1 Semantics of Visual Contracts

In SEAM, we model the behavior together with the state of the systems.

The SEAM language can be defined as a triple:

$$\mathcal{L} = \langle \mathcal{A}, \text{Ord}, \text{Obj} \rangle \quad (3)$$

such that \mathcal{A} is the set of processes/actions, Ord is the set of sequence/ordering constraints and Obj is the set of objects. Actions can be classified as local (internal) or as joint (interactive). In order to create a contract for an action, we must see an action as a predicate transformer:

$$\{ \mathcal{P} \} \mathcal{A} \{ \mathcal{Q} \} \quad (4)$$

This equation indicates that there is a set of predicates for the initial conditions (\mathcal{P}) that will guarantee that the final state (\mathcal{Q}) is reached [HOARE, C.A.R. 1969].

Now, in order to represent the state of the system, normally we should represent the state of all objects. This is what makes UML object diagrams cumbersome. Instead, we can represent the sets of objects –as it is done in formal approaches like Z— and reduce the size of the representation for most cases; this is shown in section 6.2. This is done through the concept of **set-association**; this represents the instance information either by the cardinality (intensional form) or by explicit identification of the actual instances (extensional form).

Subsequently, the **global system state** S_{SX} is the sum of the state of all set associations ($S_{\mathcal{A}_{all}}$), as expressed by equations 5 and 6.

$$S_{SX} \equiv S_{\mathcal{A}_{all}} \quad (5)$$

$$S_{\mathcal{A}_{all}} = (S_{\mathcal{A}_1}, S_{\mathcal{A}_2}, \dots, S_{\mathcal{A}_m}) \quad (6)$$

Then the actions operate on objects, whose values identify a state in the system.

As an action is a predicate transformer, the notion of change is implicit. We introduce the unary and binary operators for «change» (Δ and \rightarrow , respectively). They symbolize the change of state due to an action. Hence, from equation 4:

$$\Delta_{\mathcal{A}S_{SX}} \equiv \{\mathcal{A}P\} \rightarrow \{\mathcal{A}Q\} \quad (7)$$

If we introduce labels **@pre** and **@post** to distinguish the state before the action starts from the state after the action ends, we obtain:

$$P \equiv S_{SX} @pre \quad (8)$$

$$Q \equiv S_{SX} @post \quad (9)$$

$$\Delta_{\mathcal{A}S_{SX}} \equiv \{\mathcal{A}S_{SX} @pre\} \rightarrow \{\mathcal{A}S_{SX} @post\} \quad (10)$$

This is seemingly equivalent to a diagrammatic form of Dijkstra's notion of **predicate transformer** [DIJKSTRA, E.W. 1976]. As SAs capture instance information, actions change the state of SAs.

As here we represent sets, we can also describe: operations/relationships on sets, and contexts of existence. As an example of an operation of sets, we may transfer one or more IO instances from one SA to another. C_{SA} is the cardinality information of the set-association, and S_{IO} is the state information of the instances in the set-association. Hence, we can express the change for specific instances of IOs and SAs as:

$$\Delta_{\mathcal{A}C_{SA}^m} = \{\mathcal{A}C_{SA}^m @pre\} \rightarrow \{\mathcal{A}C_{SA}^m @post\} \quad (11)$$

$$\Delta_{\mathcal{A}S_{IO}^m} = \{\mathcal{A}S_{IO}^m @pre\} \rightarrow \{\mathcal{A}S_{IO}^m @post\} \quad (12)$$

As we are able to express diagrammatically the static state of a set-association, we can now integrate the notion of change of state. The modifications to the state of SAs can also be represented through the binary operator «**change**» (\rightarrow). This operator introduces the notion of time onto the diagrams because there is a “before the action” (**@pre**) and an “after the action” (**@post**), as in equation 10.

A set as a context of existence means that an instance that belongs to it can be referenced but cannot exist in two different contexts. This is a consistency rule that is not normally reinforced by UML.

In summary, we are able to represent the changes of cardinality ($\Delta_{action\mathcal{A}C_{SA}}$) and of state ($\Delta_{action\mathcal{A}S_{IO}}$). Then, the SEAM language in equation 6 can be redefined as a duple:

$$\mathcal{L} = \langle \mathcal{A}vc, Ord \rangle \quad (13)$$

such that $\mathcal{A}vc$ is the set of actions described as visual contracts, and Ord is the set of sequence/ordering constraints.

Below we will discuss the basic elements of the notation we created to reflect this semantics. In Table 12, in Chapter 8, we present a synoptic table of the syntax for visual contracts.

6.2 Primitives for Visual Contracts

In our approach, we model the behavior together with the state of the systems. Our modeling ontology is based on RM-ODP [ISO/IEC and ITU-T 1998] and on our formalization of it [LE, L.S. and WEGMANN, A. 2005, NAUMENKO, A. 2002].

A system is modeled as a RM-ODP object that we call working object. Working objects can be specified as wholes or as composites. The difference between view “**as whole**” and view “**as composite**” is essential. A working object “*as whole*” is described atomically. Only the externally visible behavior is described using a model-based description [SCHÄTZ, B., *et al* 2002]. A working object “*as composite*” is described as a set of component working objects that collaborate together. It represents the construction of the working object. In visual contracts, we consider only working objects “as wholes”.

Working objects as wholes are described in terms of information objects, set associations and localized actions. The information objects define the possible states of the concepts necessary to describe the working object. The set associations define instances of these concepts (together with their state). The localized actions change the state of the concept instances.

6.2.1 Basic Elements

Myself

As all instances exist in a context, there is a “first context” for each system. It is the model element **Myself**. This element represents the “root” of all behavioral and conceptual information describing the system. The double nature of this model element is represented by a symbol that combines an IO and an action (overlapping of the rectangle and the oval).



Figure 24. Symbol representing the identity of the system in SEAM

Properties = Information Objects

An **Information Object** (*IO* for short) captures the type (\mathcal{N}) of each concept necessary to describe the observed system. For example, Figure 25 represents the **IO Person**. The **IO Person** captures the information of a person in the real world. The attribute **Id** is necessary to distinguish instances from each other. Note that an IO is similar to a UML class without methods.

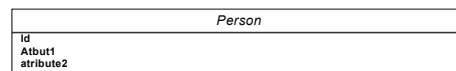


Figure 25. Representation of an Information Object or Property in SEAM

State

In addition to the type (\mathcal{N}), an IO also captures the possible states of the IO (*SSIO*). For example, Figure 26 represents the **IO Person** using the notation I propose. The attribute **Boarded** captures the state of a **Person** in relation with the **Plane** she wants to board. Thus, \mathcal{N} is **Person**, *SSIO* is **Boarded**={**offBoard**, **onBoard**}. We also include the attribute **Id** necessary to distinguish instances from each other.

Multiple orthogonal states are also possible. If we wanted to describe a second set of states, it is necessary only to add an attribute to the description of the IO in figure 26. For instance, if the *SSIO* is **civilStatus**={**single**, **married**, **divorced**, **widow**}, it is required only to add a **civilStatus** attribute with the 4 possible states. Afterwards, we might be able to describe instances of **Person** that are **divorced** and **onboard**.

Moreover, hierarchical statecharts can also be used when describing more complex configurations.

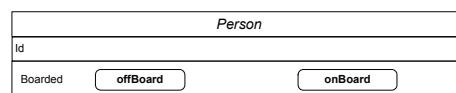


Figure 26. Representation of a stateful Information Object. State information appears in each of the attributes.

Set-Associations

Relationships between information objects are represented as **set-associations (SA)**. The purpose of set-associations is to capture information about instances. The instances of the IOs exist only in the context of SAs³⁸.

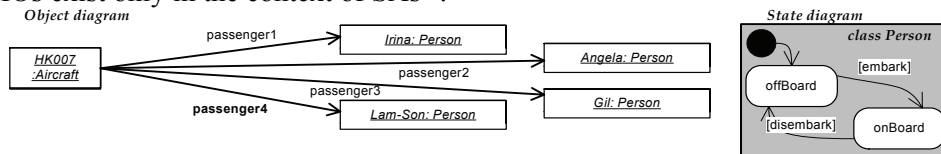


Figure 27. UML object diagram for 4 passengers and the state diagram for the class *Person*

We can give an intuitive feeling of our approach, with a UML [OMG] example (Figure 27) representing 4 people (**Irina**, **Angela**, **Gil** and **Lam-Son**) who are on the list of passengers of an aircraft. As the state diagram shows, the **Person** can be either **onBoard** or **offBoard**, hence two lists are required. This is an implicit requirement of the problem that is not visible in the UML specification (figure 25).

In UML, it is not clear how to instantiate the state information of objects. However, we just saw that we must include state information in order to differentiate the lists of **passengers**—one for **offBoard** (at the dock), and another for **onBoard** (already on the plane)—. Two of the options to represent the instance (object + state) information in one diagram are shown in Figure 28.

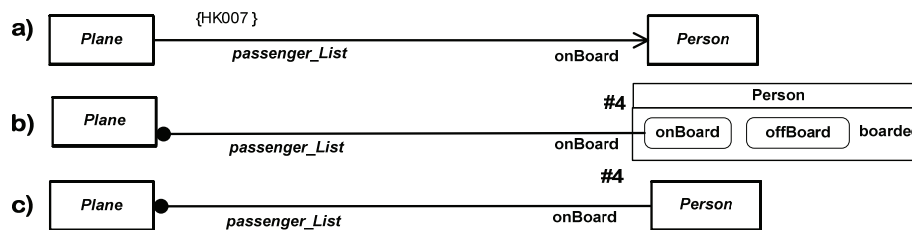


Figure 28. Representation of 4 passengers (a) UML-like model with instance identifiers, (b) SEAM-like representation using instance cardinalities + explicit IO state information, (c) SEAM-like representation with state information implicit on the set-association

Set-associations (*SA*) capture information about these instances. As set-associations relate IOs, it means that these instances exist within a context. For example a set-association between a **Plane** and a **Person** can represent a person that is either **offBoard** or **onBoard**. This is illustrated in Figure 28. For practical reasons, we avoid identifying the single instances (extensional form, like in the object diagrams of UML), and we use instead **cardinality (CSA)** and **state information (SIO)** of the set of instances. We call this the **intensional form**.

As illustrated in Figure, 28 a *SA* requires a *name* (**passenger_List**), a *referring instance* (**#1**), a set of *referred instances* (**#4**) and, optionally, a *state* (**offBoard**).

We claim that, when reasoning with graphical models, our minds use instances implicitly. This is supported by several approaches such as [LIEBERMAN, H. 1986].

Action/Service

³⁸ As explained in section 5.2, this approach is based on a relative observer, one whose reference system is the type itself and not an omniscient observer, that sees all the instances at the same time. This means that we can only model the instances of the type the object may know.

The formal definition can be found in section 3.1, and a mathematical description in section 6.2.

The most important aspect of actions in terms of specifications, is the fact that they are required in order for the system to change its state.

Parameter

A special Information Object that is normally exchanged among the system and its environment. A parameter permits the communication of the system, as explained in sections 5.2.2.3 and 6.1.

Binding strength

The binding strength permits specifying whether a set-association “contains” real instances of objects or only references to them. In other words, by using the binding strength, the model can determine some characteristics of the lifecycle of the objects in that set-association:

- When the set-association contains real instances, then their lifecycle is dependent on the lifecycle of the set-association. This is known as *tight binding*.
- On the contrary, when the set-association contains only references to actual object instances, there is no lifecycle dependency. This is known as *loose binding*.

We use a filled- and a hollow-diamond, respectively, on the side of the containing IO.

Intersection

It represents the intersection set created by two set-associations.

An intersection can be total or partial, depending whether the intersection includes all the elements of any of the intersection set-associations. Total intersections are represented by a full (black) squares, whereas partial intersections are drawn as hollow (white) squares.

6.2.2 Behavioral Description

Instantaneous cardinality

The cardinality of a configuration (snapshot) of the system is temporary. It is represented as a common cardinality, preceded by a “#” sign.

It is different from the (standard) cardinality as it is a time-dependent attribute. In contrast, the (standard) cardinality is time-independent and constitutes an invariant for the system.

Cardinality or State Change

Our set-associations evolve over time by changing their states. We distinguish three main classes of SAs:

- **IO-IO**: as described until now, it makes explicit relationships between IOs.
- **Action-IO**: makes explicit pre- and post-conditions. It allows for describing instances that exist only during an action.
- **SA-SA**: equivalent to a simple relation algebra among SAs (sets).

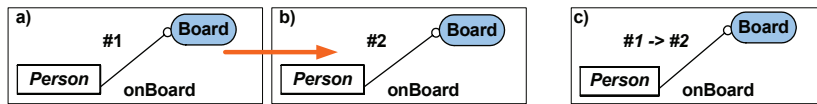


Figure 29. Action changes cardinality of set-associations. On the left side the initial (a) and final (b) conditions. In (c), SEAM notation for representing changes in cardinalities

As now we consider a single action as a unit of specification of the system, state transitions become possible. Initially, the SA selected includes one instance of IO **Person** that is **offBoard**. At the end, this same person is **onBoard**. Figure 29 summarizes this state change using the «change» operator (\rightarrow) for cardinalities and figure 31 adds a «transfer» operator (curved, wide arrow) between lists.

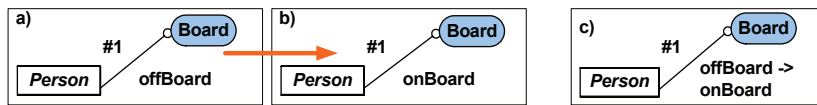


Figure 30. Action changes state of instances in set-associations

As a corollary, what is shown as not changing in a visual contract of an action is, then, considered to remain unchanged for that specific action.

Guard

A predicate associated to some set-association or operation on a set-association that makes part of a larger expression.

The guards trigger or fire whenever the corresponding predicate holds true. When a guard is fired up, the truth value of the expression must be recomputed. This is explained in detailed in section 6.4.1.

Select

Most operations on objects require a way of expressing that:

- an element belongs to a collection,
- an element of the collection can be retrieved by using one or different ways
- the kind of collection is still open to discussion

This is partially satisfied by model-based languages, like Z, where you can say simply that p belongs to a set P . Nevertheless, the fact that an element p belongs to a set P does not mean that the element p is in a collection P . The definition of set is very mathematical, and then p can be in several sets P , Q , R that are not collections but elements that from the strict point of view of logics, share a property. For example, p is in collection P , p is of type Q , but p is also in state $s1$, which means it belongs to the set R (the set of all elements of type R that are in state $s1$). The collection P is a mechanical construction, whereas Q and R are logical sets. Of course, for p to become part of collection P , there are some logical criteria that are to be taken into account.

It is clear then that the current approaches do not take into account the type of the element as we should when thinking from an information viewpoint: several attributes of the element p might be used to retrieve it, even if they are not the criteria that were used for it to become part of collection P .

Transfer

The transfer is an operator that expresses the change of state of instances, which has as a consequence the change on the membership of the instances. Therefore, more than an actual transfer of instances, it is a form of interpretation.

The transfer is shown in the form of cardinalities for the set-associations, and as a causality arrow in the diagram. This causality arrow explains the immediate cardinalities of the linked SAs change: the direction of change, the nature of the transfer *–partial or total*, i.e. hollow circle or full circle, correspondingly—.

For instance, if we retake the example shown in the definition of Set Associations above, we obtain:

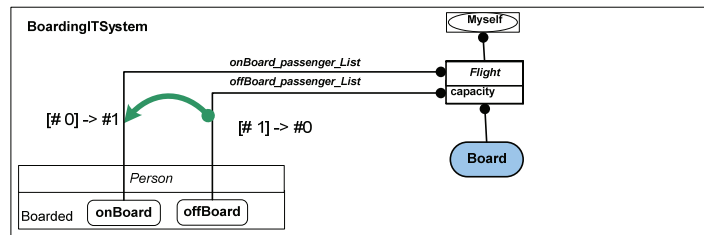


Figure 31. Action changes cardinality of both SAs for passenger lists. A transfer has been made, meaning also a change of state of the respective instances of IO Person.

Instance Creation

Instance creation represents the appearance of a new instance within the system. This is normally represented by a “new” symbol. By using a special set-association, we can represent the creation of several objects of the same type. Besides, the set-association also indicates which is the context (action) responsible for the creation of those objects.

Computation

The computation of values is particularly problematic. It is represented by the **computation** primitive. David Harel affirms [HAREL, D.] that the algorithmic operations on variables and data structures cannot be completely translated in an economic form to a visual counterpart. From our experience using this notation, we fully agree with him. In this case, the **computation** symbol should be linked to an algorithmic, sentential description of the computation or transformation that should be performed.

6.3 Visual Contract

A **visual contract (VC)** is a specification artifact that shows, in a visual fashion:

- the net effect of an action: what changes take place
- what conditions are required in order for an action to be executed

More formally, we define Visual Contract (VC) as the visual model that represents both the pre- and the post-conditions for an action, and that makes explicit the changes from the pre to the post state [DE LA CRUZ, J.D., *et al* 2006b].

A VC is created using the notation and semantics presented in the preceding sections of this chapter. The visual contract defines the required parameters that fire different postconditions; this will be explained in the section related to the execution.

A Visual Contract can also be decomposed onto pre- and post-condition diagrams. However, the causality links are lost in the process. This will be better explained in chapter 8.

Figure 32 shows an annotated version of a visual contract; it is the equivalent of the UML version in figure 33.

The main advantage is that only the elements required to intervene, either as pre- or as postconditions, are modeled. All other elements -- those that have not been affected by the action being modeled-- do not have to be included in the diagram.

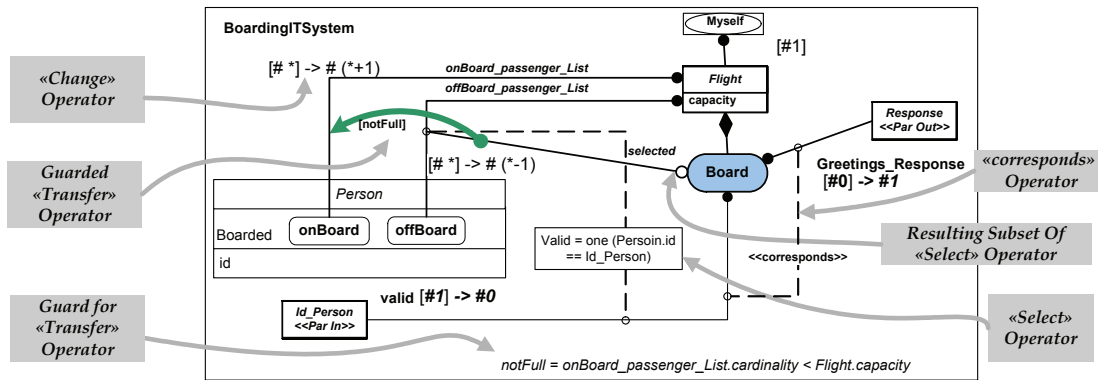


Figure 32. Annotated version of Visual Contract for action Board. Please refer to chapter 9 for a complete description of this illustration

Note that by defining Visual Contract as the specification of a specific action, this implies that some mechanism must be found to indicate:

- The constraints that should always be respected (the state and structural features that should never change). This can be done using a VC for invariants that we call the **VC domain model**.
- The situations that should be avoided. This can be done with **Visual Anti-Contracts** that are equivalent to misuse cases [REGEV, G., ET AL 2004].

Therefore, a complete system specification is made up of:

- A **Visual Contract domain model** that explains the general scenario for the system (the static structure)
- A set of **Visual Contracts** for the operations that explains how actions will be achieved, what their prerequisites and their effects on the system and on the environment are.
- Tentatively, a set of **Visual Anti-Contracts** that describe the scenarios to avoid

6.3.1 Execution of Visual Contracts

Actions take place in order to change the system state.

The core of model execution is the ability to carry out a single step of the system's dynamic operation, with all consequences taken into account

David Harel – Biting the Silver Bullet, 1992

A system or working object can perform a series of actions. Actions modify the state of information objects³⁹. The system is sensitive to events that happen in the environment and to internal events. When an **event** is **raised**, the virtual machine should perform some pattern matching on the conditions associated to input and output parameters. The default strategy is

³⁹ Note that the information objects and the actions can be further considered as whole or as composite.

greedy, which means that the longest chain of events that could be associated to a visual contract is the one that should be consumed.

A condition is said to be **fired** when the associated predicate holds true. Instead of invoking actions, we consider that the actions are automatically triggered when the preconditions of the Visual Contract have been fired. This all depends on the transactional nature of the action under scrutiny (more on this on section 7.4). Nonetheless, the default mode is transactional, which means that all preconditions of a VC should be fired in order for it to perform the action.

As in Anzac [SENDALL, S. 2002], this theory can be explained mainly for reactive software systems, i.e. systems that interact with their environments over time in an organized manner, where stimuli arrive in an endless and perhaps unexpected order. The correct treatment of these stimuli means that a reactive system must “know” whether it is in an appropriate situation in order to serve the requests and notifications that are implied by the stimuli.

In this particular case, the basic execution form is a non-deterministic automaton, with the implicit idea of a virtual machine. Most specifically, time is divided onto discrete steps. During a step of the execution, the environment can generate external events, change the truth values of conditions (this is known as **firing**), and update variables and other data elements. Consequently, the status of the system is changed. This can be expressed either declaratively via a single VC, or via a composition of multiple VCs. *“Given the current status and the changes made by the environment, calculating the effect of a step usually involves complicated algorithmic procedures, which are derived from, and reflect, those semantics”* [HAREL, D. 1992]

In this way, we can compose complex automata to model systems. However, it is important to understand the differences between internal and external operation of a visual contract and how to bind VCs together.

6.4 Discussion

The originality of the approach is to capture instance information exclusively on the set associations.

The general trend is to draw them in separate diagrams and consider only static information. In our approach, we focus on change dynamics.

The system’s behavior we just described actually corresponds to the changes in the number and state of its internal objects. The results of those changes are communicated to the environment (i.e., result objects). This visible part is what we interpret as behavior.

We focus on interactions among objects of systemic actions as the specification units of description. They contain enough business information to illustrate the purpose of the modeling process at each level during the modeling and its refinement. Catalysis [D’SOUZA, D.F. and CAMERON WILLS, A. 1998] inspired us on this regard. The refinement of actions in Catalysis is seamless throughout the whole development process. Unfortunately, Catalysis does not go far enough in elaborating how joint actions should be described at the problem domain or business level. Apparently, they considered that pre- and post-conditions are inadequate for capturing the complexities and subtleties of a group of business interactions. We demonstrate that by the use of a modified syntax, we can create Visual Contracts that fulfill this requirement.

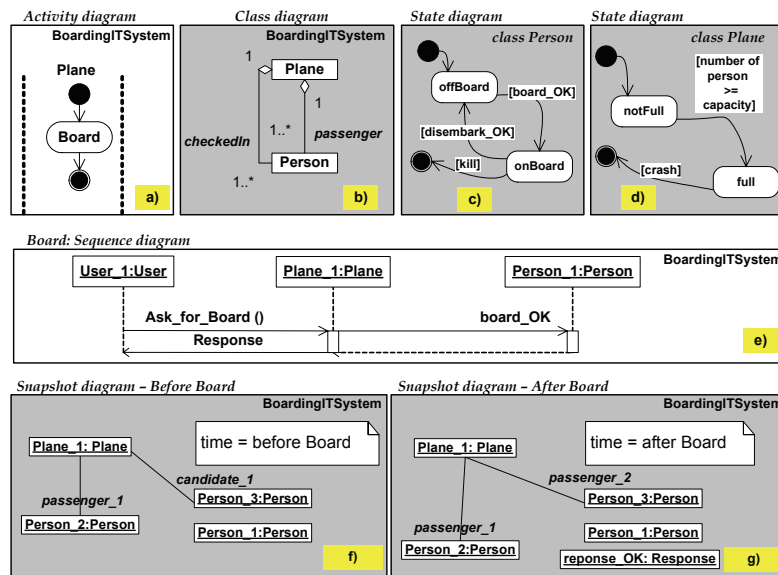


Figure 33. A partial UML specification for action Board, equivalent to figure 32.

A comparison can be made among Visual Contracts and an equivalent model created with UML. Figure 33 is a very simplified version of a UML model equivalent to the VC in figure 32. Although we will not explain in detail the differences, we must point to two main issues:

- 7+1 diagrams, using 5+1 notations were required by the UML model. This is a complex model, made up of different base models, as suggested in chapter 2.
- The UML sets and associations are all joint by default; if disjoint sets or associations exists, the OCL construction {disjoint} must be used to indicate this fact. In Visual Contracts, disjoint sets are default, but joint sets and partially joint sets can be easily specified via the intersections of set-associations.

Regarding how visual contracts satisfy the Gestalt laws we discussed in section 4.1.2.2, we present in table 8 the main points of our approach.

Table 8. Featuring the visual perception dimension of Visual Contracts

GESTALT LAW	How Visual Contracts address this law
PROXIMITY	Elements that describe action structure, typing, communication, constraints, structural change, etc. are grouped together. This is related to the concept of immediacy [AKKOK, N. 2004, SOWA, J.F. 1999]
SIMILARITY	Colors are used to separate moments in time for static, 2-body version of the visual contracts. We also use a minimum number of symbols, in order for the modeler to perceive similarity.
CONNECTEDNESS	Association poses a strong semantics, creating the notion of context of existence. This implies that connections in the model change, just as they do in real systems. Changes are also identified through lines whenever possible (transfer, corresponds, parameters). All elements are connected. The presence of an unconnected element is the sign of an anomaly.
CONTINUITY	Basic formats have been used (rectangles, ovals, straight lines, and oriented arcs)
SYMMETRY	The elements are normally arranged symmetrically around the central discussion element, the action: Parameters (In on the left vs., Out on the right) Corresponds (the initial entity or set vs. the final one) Complementary states, whenever possible
CLOSURE	"A closed contour tends to be seen as an object": We do not agree with this popular point of view. However, the closure is not avoided. It was replaced by an arrow-based notation. In addition, the border of the system is a rectangle. The symbol used as container for the set-associations (the filled circle) also represents the notion of container by itself.
RELATIVE SIZE	N.A. As instances are not shown as elements/dots within a closed curve/shape but as elements in a line This is what makes our notation more scalable
MOTION PERCEPTION	Correspondence problem [WARE, C. 2004] solved by explicit selection and transfer of concerned instances. Animation techniques cannot be used in standard formats of specifications Arrows are used for indicating change (unary and binary).

6.5 Summary

In this chapter we introduced the Visual Contracts. This new concept emerged because we developed a visual notation for **modeling change** in systemic actions. This notation required a number of modeling concepts. A very important one is the **instantaneous cardinality**, because it facilitates the representation of changes in the configuration of systems. In this way, we obtain a compact, intensional model.

In order to define visual contracts, we extend the traditional interpretation of the association and we integrate diagrams that are usually considered as separate. We define the concept of set associations that capture the existence of instances of concepts in a given context. The instances “live” in the association end. We use these set-associations to relate information objects (our term for concepts) to information objects, but also actions to information objects. To represent the pre- and the post-condition on a single diagram, we define a new graphical symbol to express the change of cardinality or state.

The notation presented in this chapter will be used to create visual contracts that can be composed and refined. Such notation might be also translated to formal expressions.

7 Set-Associations in Detail

In this chapter we study the details of the semantics and of the pragmatics of our Visual Contracts.

First, we get acquainted with the algebra of set-associations. This is the most powerful but most complex concept in our approach. Here, we clarify to a certain extent the semantics of our notation. Then, we explain the algebra of contexts of existence, in order to understand the modeling of the lifecycle of objects in a system.

In section 7.2 we discuss briefly some considerations that are important for specifying behaviors. This discussion is complemented by the study of patterns found in Section 7.3.

7.1 Algebra for Sets Associations

In order to use the collection as our unit of specification, we must be able to fulfill the expressive capacity of the set theory. For two collections **A** and **B**, we should be able to express, a minimum number of sets, after Sowa [SOWA, J.F.], p. 100:

- Membership / belongs to
- Subset, Empty set, Disjoint sets, Equal sets
- Union, Intersection
- Universal set, Absolute complement
- Relative complement
- Different types of collections: Bags, sets and sequential lists

As explained in chapter 6, our approach supports integrated semantics. In consequence, it deals directly with the collections that change over time. This is not possible in UML because:

- Collections are defined explicitly in UML only for OCL; the class diagram can represent collections but no manipulation is possible, because the behavior cannot be modeled in this diagram.
- Class diagrams support collections implicitly, but the implicit semantics imply a default case. Types/classes in UML represent implicitly all the instances and the instances are implicitly joint, however this is not clear in the standard [BOOCH, G., *et al*].
- Multiple instances might be on the associations, but the UML semantics do not clearly support this kind of modeling. Practitioners usually use this modeling shorthand: one class for representing the set and another class for representing the elements of the set. We used this notation for a long time during this research, but we chose a more strict policy based on a set-oriented notation.

Even if this is not developed in this report, it is important to note that a number of basic set operations are supported unlike UML: *memberOf*, *equal*, *empty set*. As the representation of collections (sets, sequences, and bags) is implicit in set-associations, we can actually define operations in sets. Set-associations also represent the observer viewpoint, thus enriching the semantics with additional information. Finally, it is also possible to describe properties of objects via the SAs, in the form of $\mathcal{P}(x)$.

On the other hand, unlike sagittal representation and Diskin and Z, we do not deal directly with the nature of the relation itself (total and partial functions; bijective, injective and surjective functions, etc.).

7.1.1 Operations on Set-Associations

The set-associations represent sets. In this section, we give details about the operators on set-associations that correspond to logical operations among sets.

Because the operands are the lines that symbolize the set-associations, we can call this an algebra of lines. This algebra is equivalent to a subset of a relational algebra. We illustrate this concept via the following example:

Let us suppose that our system considers one **Aircraft** and one **Company** who has workers who might board the **Aircraft**. Let us say that the **Aircraft** has a number of instances of **Person** that are **passenger**, whereas the **Company** has a number of instances of **Person** that are **workers**. These SAs are represented here by **passenger_List** and **worker_List**, respectively.

Table 9 resumes the possible scenarios and the way we can use the algebra of set-associations to represent these scenarios.

It is important to remember that cardinality of all set-associations must always be positive, as it corresponds to a number of instances. Therefore, any decrease or similar operation must always respect this rule.

7.1.2 Set-Associations as Contexts of Existence

As discussed in chapter 5, the notion of context of existence allows modeling the lifecycle of objects (systems). This is fundamental for modeling implicitly what is normally known as “referential integrity rules”. As a byproduct, we can use this concept to avoid the problems related to inconsistency of the models.

Unlike the composition techniques found in UML and other approaches, the context of existence is not only structural but can also tie actions with objects, and actions with other actions. For instance, we can model dependencies among actions and data, making explicit the temporal scope of certain objects; in this case, the instances exist only for the duration of the action (e.g. local variables, temporary variables, etc.). This kind of description cannot be built using standard specification techniques.

As described in chapter 5, modeling a context requires defining first the type that acts as a context, and the type that corresponds to the objects that live in that context. A heuristic that can assist the modeler is to consider that the referred side of the set-association is made up of the instances existing in the SA; in other words, when the SA disappears, the instances themselves will disappear too. The context end or referencing side corresponds to an object of the type, and whose lifecycle is the maximum span of time of the lifecycle of the set-association itself. When the object disappears, all the SAs that have as context such an object will disappear, too.

Another aspect of the modeling that can be enhanced via the use of set-associations is to differentiate the references to the instances from the actual instances themselves. This is normally done using an UML relationship <<pointsTo>> among the objects and/or classes. However syntactically correct in UML, it has no semantic content. Therefore, the modeler cannot validate the behavior of the system. In our case, different symbols are used to represent actual instances and references. The actual instances can exist in one context only, but can be referenced from many other contexts. In this way, an “owner” is clearly identified, and the modeler can guarantee the referential integrity of the system.

There are several issues that can now be addressed:

- We must differentiate the type that acts as a context from the one that corresponds to the objects that live in that context. The referred side of the set-association is made up of the instances existing in the SA. If the SA disappears, the instances themselves will disappear too.

Table 9. Logical operations of Set-Associations. Note that cardinality must be always zero or positive ($m \geq n$)

Algebra operator	Graphical equivalent	Description
A set		Sets are disjoint by default 0..20 elements of type Person are known by each aircraft
Empty set		Sets are disjoint by default 0. elements of type Person are known by each aircraft
Two disjoint sets		Total: All m person know by aircraft are also known by company
Intersection of sets		Partial: Only n elements among the m known by Aircraft and n person known by Company.
Two joint sets		Total: From the total m Person known by both Aircrafts, n Person belongs to the list of one of the Aircraft s and (m-n) Person to the other.
Subset		Total: From the total m Person known by a single Aircraft, n Person belongs to one list of the Aircraft and (m-n) Person to the other list of the same Aircraft.
		Total: From the m Person known by the Company, the Aircraft and the Company share a total of n Person.
Subset		Total: From the total m Person known by a single Aircraft, n Person belongs to one list of the Aircraft and (m-n) Person to the other list of the same Aircraft. The Company and the Aircraft share a total of n Person.
Absolutely defined Finite set		Total: From a total of t Person considered for the system description, m person know by aircraft are also known by company Note that (m < t)
Inclusion of Sets		Partial: Only n elements among the m known by Aircraft and n person known by family. Note that (m < t) && (n < t)
Addition of sets		Total: From a total of t instances of Person, m belong to the first Aircraft, n to the second, and the remainder work for the Company

Table 10. Default behavior for deletion of set-associations that do not include actions. The red dotted arrows represent the several deletion processes, the guards are named after the highest context that is being deleted.

Deletion Scenario	How to write this scenario using the VC semantics

- Our semantics for the context of existence is similar to UML composition/aggregation, so we will use the UML notation (black/hollow diamond).
- If there is no purpose, i.e. the entity is not known by other entities or does not interact in the context of an action, the entity should not be modeled. The context of existence makes clear why the entity exists.
- In a given level of granularity, there is no purpose on modeling an entity, because it is too fine-grained or too coarse for its effects to be perceived in that level. In this case, the modeling of the entity should also be avoided (omitted).

7.1.3 Algebra of Contexts of Existence

We have already said that all SAs constitute the context where instances actually exist. This adds a temporal frame for reasoning about the system and, in particular, what happens when an object is deleted. We consider two cases: either the actual instances in their SAs are deleted too (*tight binding*) or they are only references and the actual instances are not deleted (*loose binding*). We use a filled- and a hollow-diamond, respectively, on the side of the containing IO.

Table 11. Default behavior for deletion of set-associations that include actions. The red dotted arrows represent the several deletion processes, the guards are named after the highest context that is being deleted.

Deletion Scenario	How to write this scenario using the VC semantics

Unlike the UML primitives for composition (association, aggregation, composition), but now the modeler can also reason about actions and instance references in the same way (and not only real instances of objects).

Because addition of context and determining what remains is easily found from the operations on set-associations (preceding section), we will focus on the more difficult issue of determining what will disappear from a context. Table 10 shows the default semantics for deleting contexts of existence in the structural dimension.

Reasoning about contexts for actions when deletion occurs is also a difficult issue. Table 11 shows the default semantics for deleting contexts of existence. When the modeler requires a different behavior for one or more of the sub-contexts, he can specify it via the compensation measures (see section 7.3).

7.1.4 Set-Associations as Collections

In section 6.2.2.1 we differentiated the sets in our set-associations from the collections. Nonetheless, collections are fundamental to system specification. Moreover, collections are implicit in our approach, because the set-associations contain sets in the set-theoretic sense. Therefore, it is necessary to provide primitives to our set-associations that allow the modeler to express ordering preferences and the possibility to count with repeated instances.

For the sake of usability of our notation, the nature of the collection is specified in the name of each Set-Association, as the closing term that comes after an underscore symbol. To be precise, the set is named “_Set”, the bag is named “_Bag”, and the sequences use the particles “_List”, “_Seq”, and “_Vector”.

In addition to this, the ordered sequence elements can use the link element in order to indicate causality among events. For example, an element A followed by an element B is written $A \overset{\curvearrowright}{\rightarrow} B$, where A and B are power sets of objects of the same type.

7.2 Specifying Behaviors

Visual Contracts are a declarative approach. This means that you can bind tightly the preconditions to post-conditions or not.

As seen in chapter 2, the basic contractual model behavior makes no concession about postcondition if some precondition does not hold true. This is the default approach of design by contract [MEYER, B. 1988].

The loose binding approach occurs during the refinement, when the nature of an action is no longer atomic. In this case, atomicity is at the level of the sub-actions. This also happens for long-running transactions, as the action can be canceled long enough after the beginning, when some state changes have already taken place [D'SOUZA, D.F. and CAMERON WILLS, A. 1998]. Nevertheless, by studying single actions as not-transactional, some robustness features can be added: all scenarios –especially bad behaviors— should be taken into account. This allows the modeler to design behaviors for conditions not present in normal, well-behaved scenarios. The modeler can also specify that tracing messages are generated.

7.2.1 Transactional vs. Non-Transactional

As we just explained, single actions will be transactional by nature, whereas composite actions are not-transactional for the same reason.

When one predicate is false, the transactional action will fail and rollback completely. In other words, the system can come back to the initial state before the action started executing, thus cancelling the effects of all the actions that were executed. Specifically, in the case of transactional actions, if all preconditions are fired, the whole set of postconditions is automatically true and everything is done. If at least one precondition does not hold true, everything is cancelled. However, here we produce an error message for every case, either successful or not.

On the other hand, In the case of non-transactional actions, each predicate is evaluated independently because the clauses are fired individually. In other words, whenever any one of the single preconditions is fired, the corresponding enabled postconditions can be evaluated. If a previously fired condition is disabled, the corresponding postconditions should follow the same treatment. Hence, compensation measures should be introduced in the design. These compensations measures are executed when the corresponding exception condition happens. This is illustrated in section 10.3 by a complete case study.

It's important to note that **rely-/guarantee-conditions** (explained in chapter 2) apply to the whole execution period of an operation. In some situations, a constraint that spans the whole operation may be too strong. In order to clarify the difference, please read carefully this modified example (original version in [D'SOUZA, D.F. and CAMERON WILLS, A. 1998]) where the supply levels of a product must always be greater than zero (*guarantee* clause) while the enterprise continues to be in business (*rely* clause):

Action:	(retailer, wholesaler)::supplyForYear (amount: Money, from: Date, to: Date)
Pre:	from < to
Rely:	wholesaler.inBusiness
Post:	wholesaler.income += amount And Retailer.outgoings += amount
Guarantee:	retailer.stock->size < 10 and (from:today < to) => Retailer.orders[source=wholesaler]->size>0

The fact that the `wholesaler` is still in business (`wholesaler.inbusiness = true`) is not affected by the operation. However, `the retailer.stock` is affected by this operation.

Dealing with transactional and non-transactional (composite) actions will be illustrated in the case study of section 10.4.

7.3 Design Heuristics and Patterns in Visual Contracts

Visual artifacts are considered to help discover patterns and relationships. The use of Visual Contracts has allowed us to identify a first subset of heuristics and patterns that can be useful for specifying different kinds of systems.

Because this research work did not elaborate in a more advanced and formalized approach to patterns, we present them here as a basis for future work.

The instantaneous cardinality is either zero or a positive integer.

Multiple instances

The different notations we studied in chapters 2, 3 and 4 do not explicitly deal with the fact that the information is actually stored and has to be retrieved lately.

In general, the approach follows the dataflow diagram way (a simple question of the type: is element x there? or, is there any element y that has as value v ?) where it is not clear that the information has been accumulating for a time. For instance, a few works [HATLEY, D. and IMTIAZ, P. 1988, WARD, P. 1985] made explicit the fact that there are information storage places in the system. The colored Petri Nets [DAVID, R. and ALLA, H.H. 1997] also allow the description of systems that can have multiple instances of elements of a type X in a given place of the network; however, this does not literally mean that they are on the same collection. Moreover, that way of specifying systems does not make explicit the need of IT systems to synchronize with the real world.

The languages that support operational semantics like CCS, CSP and Pi-calculus allow you to model stackable data structures, storing and retrieving data in a sequential or random way. But this approach is *a)* not general enough when compared to model-based notations like Z or VDM where there is just an indication of the set where the element belongs to; *b)* execution-oriented, which means that we have already chosen the kind of collection we need (set, bag, or sequence) and that the search criteria is already established.

Because one of our sources of inspiration was Z , we created set-associations as a means to represent collections of instances. We can therefore support the whole CRUD lifecycle on those collections.

Besides, the input parameters can be used to represent complex patterns of sequences of events. This is because of the greedy nature of the pattern-matching algorithm, as explained in section 6.3.1.

Interaction fully informed

One of our design heuristics is that the system should always give result (a response); in this way, the users of the system will always receive a notification of success / failure for each of the actions of the system.

This is a best practice for the design of user interfaces that is useful for the creation of applications that behave correctly.

Subset selection /creation

Try to express all the operations in terms of sets. For most practical cases, the actions can be easily expressed in this way. Use subsets in order to be more specific. This will reduce the intricacy of both the clause predicates and of the expressions required to express such conditions.

Transformation

The operations that cannot be expressed as an operation of sets are usually transformations. The transformations can be simple state-changes (that can be represented using sets and subsets) or more convoluted, algorithmic approaches. For the first case, use the set-associations and the state information. For the latter, use the compute operation directly on the instances (subset) involved.

Decouple Validation Criteria

It is difficult to integrate new information regarding the specific criteria for each type of validation that is done in a system. For instance, it is not easy to add some business rule that should be applied for some or all system actions, or some integrity policy rule that must be respected by the system structure.

More specifically, it is not easy to incorporate rules that deal with meta-information. In other words, if there is some particular criteria for defining what is a valid information (e.g., a demand for social assistance has to comply to a set of criteria, a purchase has to be of less than a maximal amount if it can be approved by a certain role), it is not possible to state this clearly in the models.

The most general approach is to say indicate a state **valid** for the information object or parameter. Whenever required, a more concrete description can be developed. It is usually a **select** operator that permits checking that –for instance— an id exists already in a certain register (list) or that it is not yet registered.

Capacity – Empty / Full

Visual Contracts make explicit the fact that modelers deal with real systems and real objects, and that the corresponding representations should obey physical laws. The capacity problems usually have an impact over the (logical) controllers of the system being developed.

Include all the physical limitations, such as containment, and foresee what happens whenever such limits are surpassed. Compensation/error handling actions should be conceived in order to cope with such situations.

Staircase breakdown of behavior: Dealing with exceptions

Composite behavior will result in a very structured, staircase-like composition of Visual Contracts. The staircase exploits the concept of symmetry and minimizes the number of steps (actions or Visual Contracts) to develop.

This pattern is illustrated in the example of section 10.3.

Containers -> State implicit

As shown in figure 40, a useful shorthand is to make the state explicit in the set-association naming scheme. The upper part of the figure indicates that the IO **Person** can be either **offBoard** or **onBoard**.

The first example of the transfer operation is done in the standard way. This occupies a bit more of space. The example at the bottom of figure 34 uses the explicit state naming. It occupies less space, without any loss in the quantity of information that it carries.

Therefore, the use of state information explicitly in the name helps diminishing the complexity of the diagrams and augments usability.

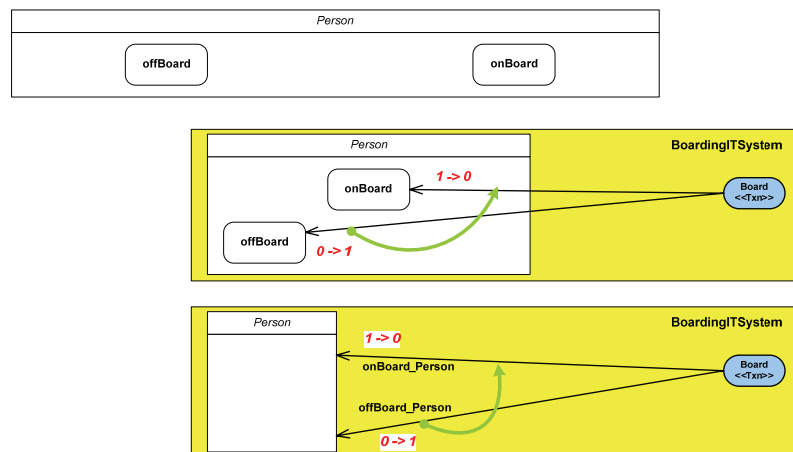


Figure 34. Making explicit the state information of instances belonging to a set-association. The IO Person has two possible states. The two diagrams at the bottom illustrate the implicit and explicit use of state information in set-associations

7.4 Summary

In this chapter we introduced the operators that can be applied to set-associations. These operators allowed us to present and explain the algebra of set-associations and the contexts of existence.

The algebra of set-associations is useful to describe complex scenarios. As the set-associations correspond to collections, we can now represent operations among collections in a declarative form. We illustrated some of the main operations for sets and demonstrated that our visual notation is more scalable than the set-theory.

The contexts of existence allow the modeler to establish containment relationships. We consider its semantics are stronger than the ones used for describing structural properties (i.e. E-R diagrams, class diagrams) since some of the referential integrity constraints are included by default in our approach. This will be better illustrated in the study case of Section 10.5.

Finally, we studied the pragmatics of the language. The pragmatics is related to the interpretation of valid language constructions. In particular, we explained:

- How the semantics of transactional actions is different from the composition semantics that are traditional in non-transactional actions.
- Some patterns and heuristics that we have found during the specification of the study cases.

8 The Language for Visual Contracts

In this chapter we present the details of the language we created for writing Visual Contracts. We introduced the foundations of our specification language in Chapter 5, the notation in Chapter 6, and explained in depth the fundamental notion of set-association in Chapter 7.

In this chapter we explain the details of the language and how to build well-formed Visual Contracts. We explain in detail its primitives and some of its rules and guidelines. Next, we describe the metamodel of our language for Visual Contracts.

By following the rules the designer can build “correct” Visual Contracts that can then be validated. This validation can be either manual or automatic. The manual validation is visual, and is made directly by the designer. The automatic validation requires an additional phase of translation to a first-order language named Alloy. As this language does not support the entire set of primitives of our notation for Visual Contracts, some guidelines indicate the primitives that are not supported.

The metamodel is used for transforming a Visual Contract to its Alloy equivalent. The metamodel is explained in this chapter, but the translation strategy is explained in Chapter 9.

8.1 Primitives for Building Visual Contracts

Table 12 presents the complete set of primitives of the language. The details about the primitives can be found in the right column. The theoretical aspects are explained in section 6.2 and 6.3.

It is important to note that in the table we propose two alternatives for the symbol “action”: a) the normal view (white-box), that shows the internals of the Visual Contract, and b) the external (black-box) view that is used in order to compose a larger action from composite actions. The black-box view is explained in the study case of section 10.3.










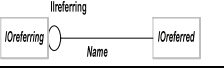











8.2 Rules and Guidelines for Building Visual Contracts

In this section, we list the rules and guidelines that make Visual Contracts well-formed and verifiable. The sub-sections correspond to the groups listed in Table 12, namely: basic elements, relational elements, behavioral elements, and temporal elements.

8.2.1 Basic Elements

- A Visual Contract corresponds to a single action in a single system
- The system is represented by one instance of *Myself*, surrounded by a box with the name of the system.
- *Myself* is the anchor for the representation of the contract. All the cardinalities and relational algebra are computed from the reference to the system (the symbol *Myself*).
- If more than one *Myself* appears, it corresponds to the same system
- The default action if the VC is a normal (white-box) action. An expandable (black-box) action is indicated by a stereotyped action <<+>>.
- When representing a sequence/combination of VCs, each action is normally shown as an expandable action.
- An Information Object represents a type.
- Every Information Object has a default identifier attribute (id).

Table 12. SEAM notation elements required for Visual Contracts

Basic elements		
Modeling element	Symbol	Short definition
Myself		It represents the system (computational object). It is the default context of existence of instances of all other IOs inside the system. See section 5.2.5
Information Object (IO)		It is used for modeling the information of the objects in the environment of the system (working object). It stores the state of the observed object. It may have attributes, each one with one or more different states.
State		Value or set of values that are either directly observable or that constraint the behavior of an IO. States are often stored in the attributes of the IO
Action / Service	a)  b) 	Specifies the effects of system's reactions to a set of events for a given system state: $[P]X[Q]$. Option b is the external (black-box) view of the VC for the action.
Parameters	a)  b)   	They represent events in the environment of a system(working object) and, more specifically, the objects exchanged with the system. The parameters are actually special information object used to communicate (Input or Output) either through the boundary of the system or among action boundaries. a) Notation for input parameters, b) notation for output parameters
Relational description		
Modeling element	Symbol	Short definition
Set-association (SA)		Relationships between information objects (or between actions and information objects). Equivalent to a set of instances of the referred IO as seen from the referring IO
Binding strength		A SA constitutes the context where instances actually exist. This adds a temporal frame for reasoning about the system and, in particular, what happens when an object is deleted. We consider two cases: either the actual instances in their SAs are deleted too (<i>tight binding</i>) or they are only references and the actual instances are not deleted (<i>loose binding</i>).
Intersection	a)  b) 	Operator added information. It indicates whether the relational operator that operates on the SA is applied to either a) a subset (partial) or b) all instances (complete). We can also represent none, by using an X instead of the square.
Behavioral description		
Modeling element	Symbol	Short definition
Instantaneous Cardinality	#	Instantaneous number of instances that compose a set-association(SA). It can be a given integer value, including zero, but also a range of numbers. It can never be less than zero (0)
Cardinality or State change	→	It indicates an initial and final cardinalities of an SA, normally affected in the context of an action
Guard	[condition]	The associated operator (change, transfer, creation, computation, attribute value change, time wait or delay) is effective when the condition predicate holds true
Select		It indicates the criteria used to create a subset from a pre-existing set-association. It provides contextual information for modeler but also used during execution to select subsets from an original set. It is drawn as a box containing a predicate, linking the source and target SAs
Transfer		It transfers a subset of instances when the guard predicate holds. It is frequently part of the change performed by actions. How complete the transfer is can be determined by looking to the intersection on the source side
Instance Creation		An instance of IO is created by using one or more of these operators. The sources for an instance are all the parameters or other IOs that will be used to create it.
Computation		Special transfer + creation operator. The instance(s) on the source side will create a number of instance(s) on the target side after a computation is done on the source IOs. The name of the function is indicated in the bubble.
Attribute value change		When a specific value of one or more IOs is changed, this operator links the new value (the source IO or parameter) and the target instances.
Temporal description		
Modeling element	Symbol	Short definition
Lifecycle		In order to better illustrate the structural/behavioral nature of myself, the three parallel lines show the fact that actions will be composed in order to create the whole lifecycle of the working object.
Time delay		The action will not be executed unless the time-related guard is true. It can be used to show a minimum or maximum time of wait before executing the action. The time is relative to the beginning of the action but when otherwise specified.
Time wait	[Δt >= xx]	The time wait is a general guard that will prevent execution of other guards before the guard is true. It is a relative time within the context of the action, which indicates a maximum time.
Corresponds		It links causally-related information object instances (set-associations). Most general case is to link a number of results are obtained from a corresponding inputs

- The number of instances of a specific Information Object can be constrained. The amount of instances must be indicated in one of the corners of the IO symbol. In

this case, the sum of all the Set-Associations that refer to this IO must be less or equal to the constrain value.

- The default constraint value for the instances of an Information Value is undetermined.
- An Information Object can appear more than once in a Visual Contract. This is normally done to differentiate IOs that have different constraint values.
- If an Information Object appears in the default (unconstrained) way two or more times in a Visual Contract, the modeler is describing the same set or collection of instances with all of them.
- The constraints cannot be mapped to the Alloy representation of Visual Contracts, therefore, it does not appear in the metamodel.

8.2.2 Relational Elements

- A VC shows the effect of the action on the configuration
- The configuration can be described in terms of the Set-Associations. This way of expressing the context is known as expanded.
- A Set-Association represents a collection of instances either of Information Objects or of Actions.
- The number of instances is represented by the instantaneous cardinality (the operator #).
- Cardinality is always positive. As the cardinality applies to real sets, it must be an integer and cannot be negative. The modeler should verify that formulas and dynamic cardinalities give valid results.
- SAs can be either static (no change for a certain Visual Contract) or dynamic (the opposite of static).
- A Set-Association is unidirectional. It has a referring or owner (observer) end and a referred end. The referring side is indicated with a small circle.
- The referring side of the Set-Association can be either the system (Myself), a type (IO) or an action (Action).
- The basic configuration is one system whose Information Objects have only SAs tied exclusively to the system (Myself). In this case, all instances of different types exist only in the system. This kind of Set-Association is called SAMyself, and can be either dynamic or static.
- A more complex configuration is one system whose Information Objects are linked not only to the system (Myself) but also to other Information Objects. A Set-Association linking two Information Objects is called SAIO, and can be either dynamic or static.
- A Set-Association can only have one referring side. Only one IO can appear on the referring end.
- A Set-Association can be linked to one or more referred objects of the same type (Information Object). A SA referring to a single IO represents a single collection. A SA that forks represents a set that is split in different subsets; the IO at the end of a SA must be the same type (IO).
- A Set-Association represents a collection of instances either of Information Objects in a certain state. In this case, the state information must be included in the IO, and the SA points to it directly, going through the border of the IO symbol..
- A Visual Contract can specify an undetermined number of instances of a specific Information Object.
- As two collections can intersect, hence two Set-Associations can also intersect, This is represented via the intersection symbol.
- An intersection represents the subset of two intersecting collections.

- When more than two collections intersect, use more than one intersection symbol.
- An intersection can be either partial or complete. In the metamodel this is represented via the symbols `CompleteInter` and `NCompleteInter`, respectively.
- An intersection can be either static or dynamic. In the metamodel this is represented via the symbols `NormalIntersection` and `TransferIntersection`, respectively.

8.2.3 Behavioral Elements

- An action can receive parameters. Each parameter is known as an `IOParameter` in the metamodel.
- The instances of a given `IOParameter` exist in a SA. This SA is called `SAPar`. A `SAPar` can be either static (no change for a certain Visual Contract) or dynamic (the opposite of static).
- A transfer can link two Set-Associations, two intersections, or one intersection and one SA.
- A transfer is enabled only when the associated predicate is true. No predicate in a Visual Contract should be empty.
- A default value of false is assigned to every predicate in a Visual Contract. Therefore, all transfers are disabled by default.
- A transfer that is done automatically by default can be indicated via a predicate of value true.
- A dynamic intersection is mandatorily the source of a transfer; in this case, the intersection is created with the sole goal of creating a subset that should change its state or go to a different collection.

8.2.4 Temporal Elements

- The temporal elements cannot be mapped to Alloy in the current state of our formalization strategy. Therefore, the temporal elements are strictly graphical and can only be used as documentation for the modeler.

8.3 Metamodel

[LAPLANTE, P.A. and (ED.) 2001] defines metamodel as a type graph with additional constraints. The constraints are the rules used to build models. A metamodel is an explicit model of the constructs and rules needed to build specific models within a domain of interest [OMG].

We will build our metamodel using a lightweight version of the Meta-Object Facility language (MOF) [OMG]. Therefore, fundamental types such as Integers and Strings are already supported.

8.3.1 Basic Elements

The basic elements are (see Table 12) the following: `Myself`, information objects, state, action, and parameters. Figure 35 shows these elements in the metamodel.

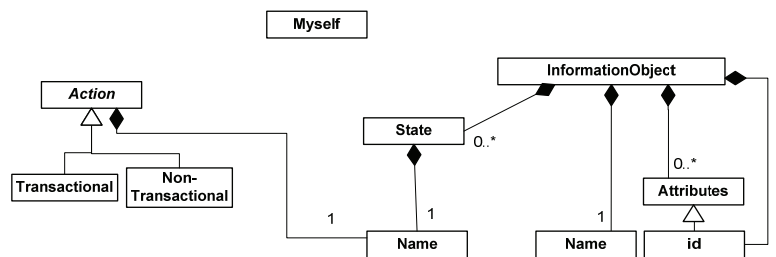


Figure 35. Basic elements in the Visual Contract metamodel

An **InformationObject** may have a number of attributes. The two mandatory attributes are the **Name** and the unique identification number, **Id**. The identification number is required to distinguish instances (Instances are introduced in section A2.2).

InformationObjects can also be characterized by a state space. This space is made up of several possible **states**. A **state** has a **name**. Each instance is in a state that can be different from the state of the other instances.

Another basic element is the **action**. An **action** has a name. Actions can change the global state of the system, but expressing this change requires the notion of set associations (see section A2.2).

For the sake of simplicity, parameters are introduced later, in section A2.3; analogously, we explain transactional

The last basic element is the identity of an instance of the system, **Myself**. As the notion of containment and the relationships have not been introduced, the system remains isolated in figure 78.

8.3.2 Relational Elements

Once we have described the main types (the information objects), it is possible to describe instances of those types. This is normally done in another level of the hierarchy of metamodels.

Because there is only one instance of the system (represented by **Myself**), we can create collections of elements that exist in this context. The Set Association represents a container of collection of elements. This notion allows us to fuse instances on the same level of the types.

Set Associations are named using the particle **sa** at the beginning. There are collections of elements that either belong to the system itself (**SaMyself**), to the communication between the system and its environment (**SaPar**), other ones that are created from other Set Associations during the evolution of the system via actions (**SaVar**), and –finally– collections of information objects related to actions (**SaIO**). These elements are shown in figures 36 and 37.

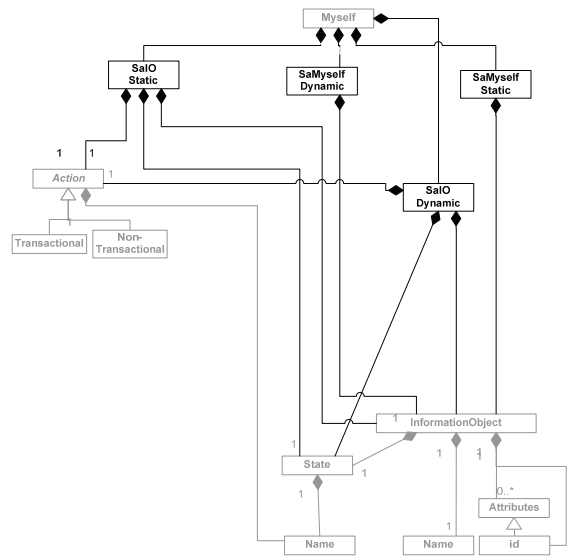


Figure 36. Relational elements in the metamodel of Visual Contracts. First partial view.

These collections of elements (set associations) can either remain constant with time—at least in the context of one action—or change. This is why set associations are specialized onto **StaticAssociation** and **DynamicAssociation**.

Because of their expressive importance, cardinalities are explicitly represented in the metamodel in the set associations **StaticAssociation** and **DynamicAssociation**, respectively **StaticCardinality** and **DynamicCardinality**. The latter will be introduced in section A2.3 as it belongs to the behavioral description.

Set associations have two ends: the owner of referring element, and the referred elements or collection. The referring element is the “observer”. It “knows” a number of elements on the other end. These elements are then bound to the referring object. The nature of this binding (**Bind**) may be either strict containment (**Tight**) or just a reference to elements that belong to other collection (**Reference**) having another owner. An element can be in only one tight collection, but referred by many others. This is illustrated in figure 37.

As we discussed in section 6.3, a number of relational operations may take place. In general, these operations give birth to new collections of elements. This is done via **intersections** from pre-existing collections; these **intersections** can cover the whole collection (**NcompleteInter**) or a part of it (**NNotCompleteInter**). An extreme case is the empty intersection, that allows creating empty collection but it is of limited importance and is not shown in the metamodel. Every **intersection** is linked to a **predicate** that can be evaluated. The predicate is a **sentence** or logical expression that makes reference to the elements of the model. **Sentences** can be as complex as the model itself, and we will not elaborate on them further.

Note that all collections that correspond to the evolution of the system (**SaVars**) include at least one intersection.

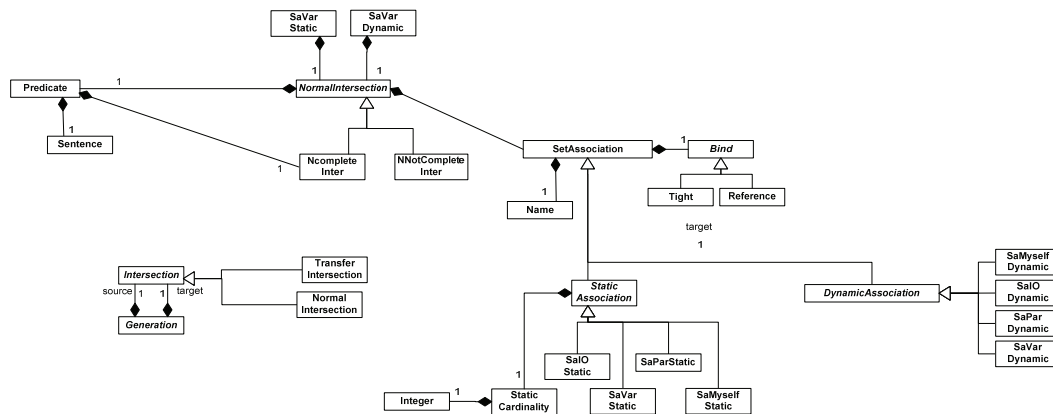


Figure 37. Relational elements in the Visual Contract metamodel. Second partial view

8.3.3 Behavioral Elements

Actions happen in a time lapse. There are at least two identifiable time-points: before the action takes place and after the action takes place. Therefore, including the notion of time allows us to represent the dynamics of the system. Take into account that a Visual Contract explains the effects of one single action.

The state of the system is the configuration of the system: a number of instances of information object, in other words, cardinalities. A change of the state corresponds, thus, to a change of the configuration (i.e. a change in cardinalities). This is shown in the upper left corner of figure 38, where **DynamicCardinality** is represented.

The pragmatics of our modeling language should prescribe that a decrease in one set association requires an increase in another set association (but when the elements are being deleted from the system). This symmetric exchange is symbolized by the **Transfer** operator. The **transfer** can be either complete (**TcompleteIntersection**) or partial (**TNotcompleteIntersection**), just like **intersections** (see section A2.2). Empty transfers occur only when the associated **predicate** does not hold true, which means failure of the action.

Actions can be **transactional** or not (**NonTransactional**). When one **predicate** is false, the transactional **action** will fail and rollback completely, coming back to the initial state before the action started executing. In the case of non-transactional actions, each **predicate** is evaluated independently, and an inconsistent state may be reached.

Parameters are symbolized by the **IOParameters**. Parameters are special **IOs** that are used to communicate with the environment. They can be used for input (**ParIn**) or output (**ParOut**) and belong to the context of the action (via **SaParStatic** or **SaParDynamic**).

Parameters trigger and condition the execution of actions. The **select** operator is used to create sub-collections of elements. It takes on input parameter (**IOparameter**) and applies a predicate that indicates how to create the sub-collection from the target collection.

The creation of new instances can be done via the **InstanceCreation**. It normally takes one input parameter (**IOParameter**, acting as **source**), that will be associated to a new kind of collection (**CreatedSA**). The net effect of the creation process is that there is some new collection at the end.

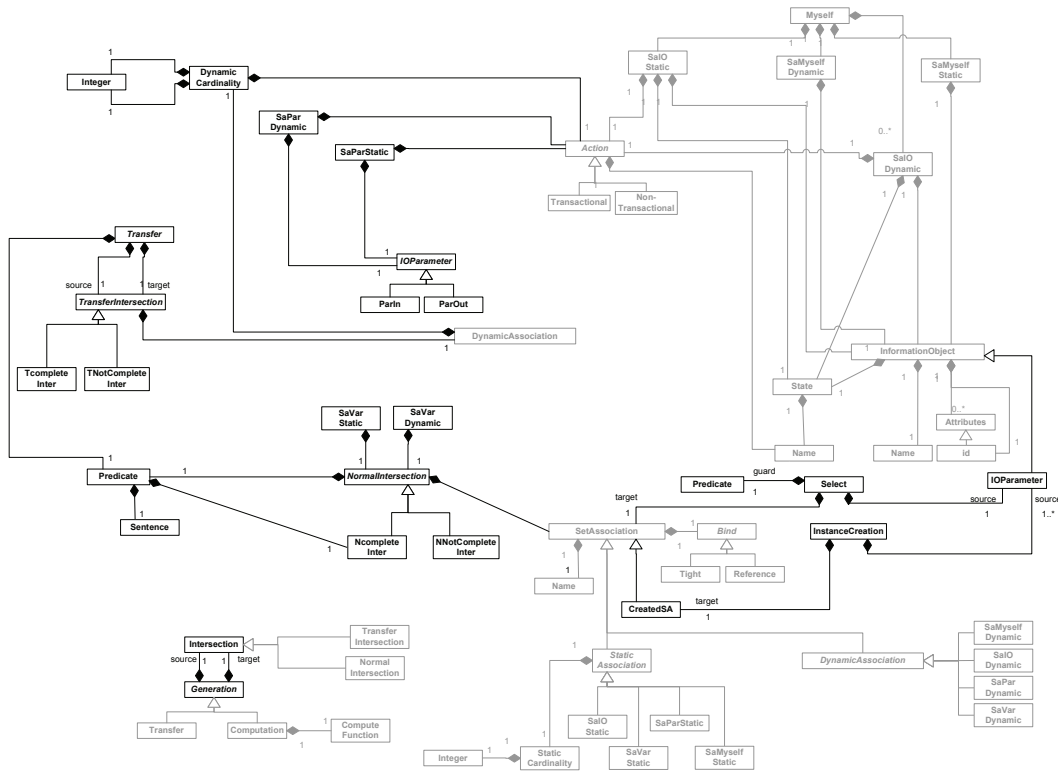


Figure 38. Behavioral elements in the metamodel of Visual Contracts.

8.3.4 Temporal Elements

Three temporal elements exist: lifecycle, time delay and time wait.

The **Lifecycle** is the context of existence of elements, and has been already discussed in section A2.2. **TimeDelay** and **TimeWait** are attached to actions and set associations, respectively. They include a **TimeExpression** that is evaluated in execution time. The logical expression can use absolute or relative time. However, this is still a subject of research and will not be discussed further in this thesis.

8.3.5 Complete Metamodel

The resulting metamodel is shown in figure 39.

8.4 Summary

In this chapter, we explained the primitives of the notation for Visual Contracts and a set of rules to use them correctly in a Visual Contract.

We introduced the most important rules for the construction of Visual Contracts. These rules are essential to guarantee that the use of each primitive in the specification is consistent. As a result, the well-formedness of the Visual Contract should be easier to achieve.

Moreover, we explained the practical aspects of the automatic validation of certain rules. We identified the conflicting points for the translation to Alloy as well as some primitives that are not checkable using the Alloy Analyzer.

Finally, we presented the complete metamodel for Visual Contracts. The metamodel is the foundation for the tool that translates Visual Contracts to Alloy; this tool is presented in Chapter 9.

9 Translating the Visual Contracts to Alloy

Strictly speaking, we should not discuss whether software elements are correct, but whether they are consistent with their specifications. This discussion will continue to use the well-accepted term “correctness”, but we should always remember that the question of correctness does not apply to software elements; it applies to pairs made of a software element and a specification

Bertrand Meyer – Object Oriented Construction of Systems

Our goal is to propose a notation and demonstrate the correctness of the systems built using the Visual Contracts. In this chapter we present the process used for translating Visual Contracts to the Alloy language. As the specifications written in Alloy can be analyzed, we can verify and validate our Visual Contracts.

Our approach is *ad hoc*. We formalize the notation for Visual Contracts (see the Chapter 6), by including some of the semantic subtleties discussed in Chapter 7.

The structure of the chapter is as follows: First, we introduce the general formalization approach. Next, we introduce Alloy language. Later, we show how to translate the main elements of our notation, and of simple examples of Visual Contracts; in particular, we explain how the notion of time is mapped to the Alloy specification. Finally, as the language of Visual Contracts is **compositional**—the meaning of a sentence is a function of the meaning of its parts— we explain our technique to compose behavior using a sequence of Visual Contracts.

9.1 Translating the Visual Contracts

Visual Contracts are built using a graphical, set-oriented notation for modeling systems. Given its set-oriented nature, we should be able to translate it to a model-based specification language such as Z, VDM, B or Alloy.

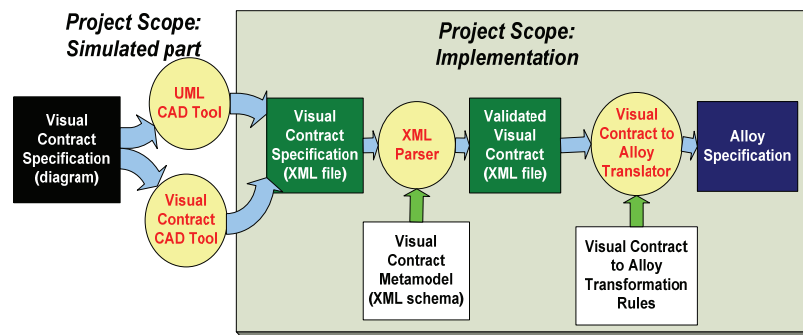


Figure 40. Strategy of translation from Visual Contracts to Alloy

Figure 40 illustrates the process: first, we defined a Visual Contract. Second, this visual contract should somehow be mapped onto a XML file (the output of a CAD tool)⁴⁰. Next, a XML parser applies some transformation rules in order to extract the information contained in the VCML files for generating formal specifications written in Alloy. Finally the formal specifications are used to verify and validate the generated models.

Our goal is to achieve the translation from Visual Contracts to Alloy. The first, manual approach to this translation, as well as the fully detailed translation technique is documented in [DE LA CRUZ, J.D., *et al* 2006a, DE LA CRUZ, J.D., *et al* 2006b, DE LA CRUZ, J.D., *et al*

⁴⁰ This language is known as VCML. It is presented in Section 7.3.

2005]. In this chapter we explain the generalized approach that enables the automatic translation.

9.1.1 Alloy Specification Language

The target language is Alloy, based on first-order logic. Alloy is a modeling language very similar to Z, but that enhances on the user-friendly and on the capabilities of the analysis. Alloy is targeted at the creation of micro-models of software systems that can then be automatically checked for correctness [JACKSON, D.].

A system specification in Alloy is composed of the following parts:

- *signatures* (keyword “sig”): declares types, sets and relations among types
- *facts*: global constraints, invariant properties
- *functions*: parameterized constraints
- *assertions*: theorems to check, concerning system properties
- *commands*: run function and/or check assertion

The *signatures* and *facts* describe the structural and invariant properties of the system. The *functions* describe the dynamics of the system. The **model-checking** is done by “executing” the *functions* and *assertions* found in the *commands* section.

The *commands* section specifies not only the *functions* and *assertions* but also determines the size of the search space. As Alloy is a lightweight formal language, the specifications written in Alloy should be verified in a small state-space and then the modeler should deepen the exploration incrementally by establishing larger search spaces. We have adopted that approach for our work.

Alcoa, the Alloy Analyzer tool [JACKSON, D. 2002] has been used successfully for performing automatic verification of software specifications and of protocols for distributed systems. The Alloy analyzer is built on a SAT prover. Technically, it is a model finder because it uses the SAT prover in order to perform an exhaustive search of configurations for a given size of the search space. As a result, it can generate instances of models that correspond to a specification (and hence, check that the specification has no inconsistency). Whenever one of the constraints of the specification is violated by one of the configurations, in the search space the system specification is considered inconsistent. In this case, the tool generates the corresponding configuration (a *counter-example* of the model) that can be used by the modeler to diagnose and correct the specification.

9.1.2 Representation of Information Objects

An Information Object (IO for short) is the equivalent to a property of the system. It captures the type of the possible states of the observed system. An information object is characterized by a name (\mathcal{N}) and a set of possible states (SS_{IO}), as shown for a hypothetical IO **TypeA** in Figure 41.

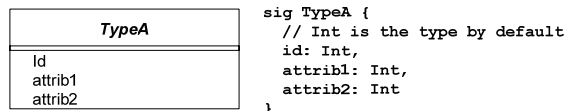


Figure 41. Visual Contract symbol and Alloy specification for an Information Object

An Information Object corresponds to a signature (**sig**) in Alloy. Inside the IO signature, you can specify its attributes. In this case, **TypeA** has three attributes.

We must include state information in order to differentiate the instances of **typeA** – one for **state1** and another for **state2**–. We must then specify two possible values for attribute **attrib3**. These values are the states of this attribute. In order to create the state space in Alloy,

we build a generic state (**attrib3StateSpace**) and inherit from this state (**state1**, **state2**). This two values of state are not exclusive. Note that we use the signature to specify state, too.

In our case, the IO **TypeB** now contains state information: Two of the options to represent the instance (object + state) information can be represented in one single diagram are shown in Figure 42. As there are potentially many other state values in the state space of the system (inheriting from **attrib3StateSpace**), we must specify clearly in the Alloy that we are only interested in the instances that are in one of the two states (**state1**, **state2**) via the invariant of the IO **TypeB**. The invariant is enclosed by the second set of brackets, and says that **attrib3** can be either in **state1** or **state2**.

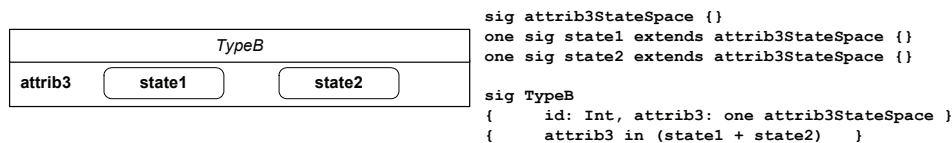


Figure 42. Extended notation for the Information Object, and Alloy equivalent for this extended notation. State information is included

Now we are able to differentiate the collections of instances of object of type **TypeB** in each state. Remember that the IOs represent the “casts” however the instances themselves exist only in the SAs, drawn as unidirectional lines. All instances exist in a context, as shown in figure 43. The system itself has to be made explicit—via the **Myself** IO—as instantiating objects requires that a system exists, and that the description takes place in the context of this system. In more practical terms, set-associations must always be drawn in order to establish how many instances of each IO are present. Hence, in figure 43, two collections exist in the context of this simple IT system (SA linking **Myself** and the IO **TypeB**). Multiple instances of **TypeB** may exist in the context of each collection **TypeB_List**.

A set-association is represented in Alloy as a relation. A relation is a special kind of attribute whose cardinality is variable. In this case, **state1_roleTypeB_List**. As the collection is of type set (indicated by the “_List” suffix), no single object can be repeated in the list. This is the situation for all represented set-associations; therefore, we specify a global invariant (**uniqueId**).

Note that we have included an attribute **capacity**. Because almost all modeled systems are physical systems, we should model the fact that these lists must be limited by some capacity. This is reflected in the invariant that limits the number of instances of IO **TypeB**.

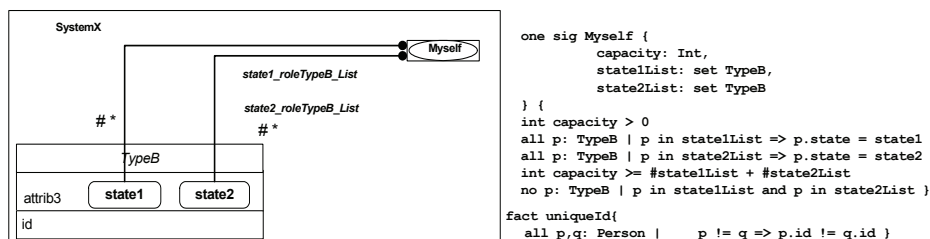


Figure 43. SEAM notation for property definitions in the specification of a system

In the Alloy specification, we can see that the signature **Myself** contains its own attributes as well as the relations that correspond to the set-associations. Because of the principle of exclusion, the instances of **TypeB** that are in one state cannot be on the other simultaneously. As a consequence, in order to guarantee the consistency of the system, each set association

must contain only elements in the corresponding state. This constraint is mapped in Alloy as the first of the two invariants of the signature `Myself`.

9.1.3 Representing Time Statically

The Visual Contract representation, shown in figure 43, can be read as follows: in the system (i.e. `Myself`), there is the knowledge of two `TypeB_List` that can contain multiple instances of `TypeB` each; one is for the instances that are in `state1` and the other for those in `state2`. In summary, this diagram takes the structural invariants just like UML class diagrams [BOOCH, G., *et al*] and entity-relationship diagrams [CHEN, P.].

The Alloy equivalent is the following:

```

module models/moduleX
open util/ordering[Time]

module models/boarding
open util/ordering[Time]
sig Time { }
sig TypeB {
id: Int
}

fact uniqueID {
all p,q: TypeB | p != q => p.id != q.id
}

one sig Myself {
capacity: Int,
state1_roleTypeB_List: set TypeB -> Time,
state2_roleTypeB_List: set TypeB -> Time
} {
int capacity > 0
all t: Time | int capacity >= # state1_roleTypeB_List.t + # state2_roleTypeB_List.t

all t: Time | no p: Person | p in state1_roleTypeB_List.t and p in state1_roleTypeB_List.t
}

```

The temporal constraints consist of a projection of IO instances in the time dimension. This allows us to retrieve the information in vectorial form via the relational operators in Alloy. The Alloy can be read as follows: a set of ordered time points are defined (`sig Time`); a set of instances of `TypeB` are defined (`sig TypeB`) with a unique identifier (`fact uniqueID`). We define also 2 lists or set-associations: `state1_roleTypeB_List` and `state2_roleTypeB_List`. These lists include a relation between an object of `TypeB` and a time point (necessary to simulate the execution sequence).

Invariants do not depend on time, so they should not vary in the time vector. Some invariants are defined in the system: the capacity is never exceeded, and no instance of `TypeB` can be in both lists at the same time. Note that there is no extension in time for the invariant regarding the capacity.

9.1.4 Representing Time in Operations that Change Collection Members Only

An action is required for changes to take place. Some immediate cardinality must change as the result of state change or new instance creation (`Transfer`).

The introduction of actions means that there is at least two points in time to analyze: a point in time before (precondition) and a point in time after the action (postcondition).

Figure 44 shows the SEAM visual contract for operation `Init`. It states that the number of `Person` in `state1_roleTypeB_List` is set to zero, so it goes from some initial value (any,

symbolized by the character ‘*’) to 0. In the practice, this means that all the instances of **TypeB** linked to this **TypeB_List** are deleted.

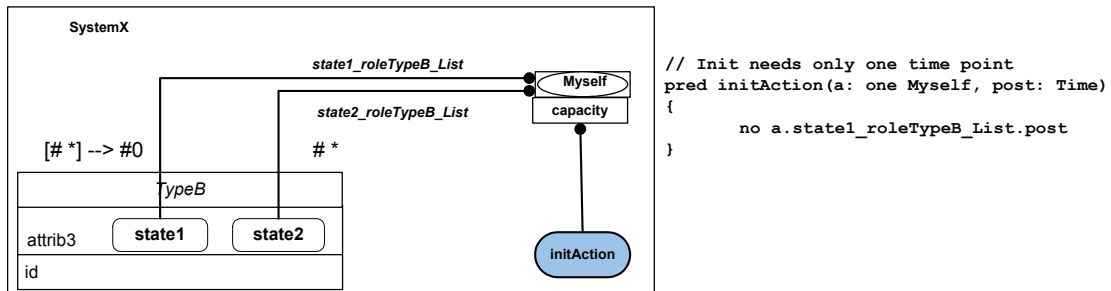


Figure 44. The Visual Contract of action `initAction` and the corresponding Alloy code

As shown in figure 44, the action is equivalent to an Alloy predicate (`pred`). Inside the `pred` `initAction` we specify that no elements exist in the SA `state1_roleTypeB_List` at `tiem` `post`. This point if time is a parameter of the predicate. The other one is the system (`Myself`).

9.1.5 Representing Time in Operations that Change the State of the Objects

Many of the actions that take place in Information Systems can be described as composed of two phases:

- Selecting some information
- Transforming that information

In most basic cases, the transformation is just a change of state of one or more instances. In order to do so, the first phase, is a simple selection of instances. Among many other, some ways of selecting instances are:

- Deselecting some information
- Input/receive selection information
- Apply select information
- Preserve the selected instance information
- Transforming that information
- Take selected instances information
- Change state of each instance to new state
- Confirm change via a message

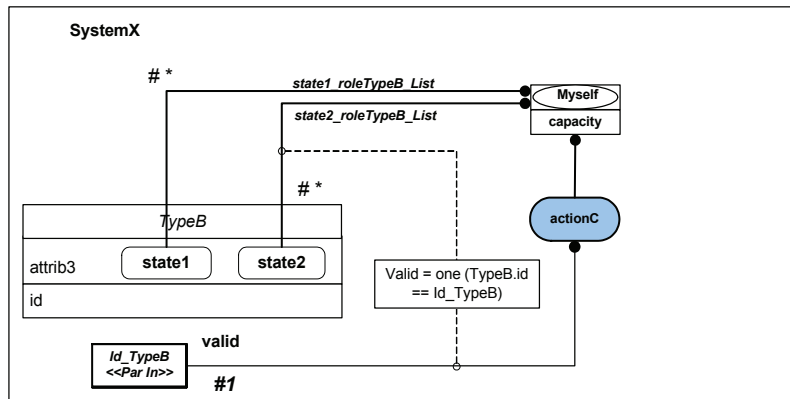


Figure 45. Precondition for action actionC

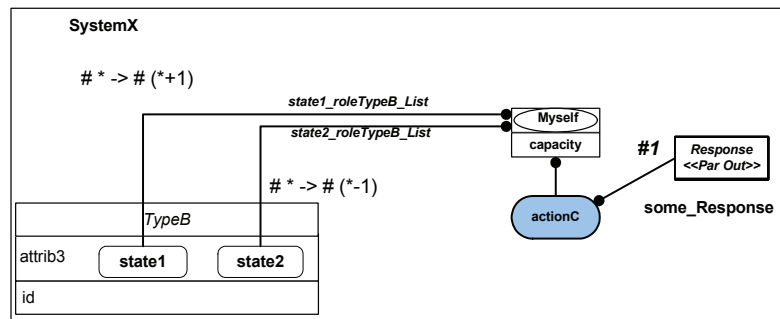


Figure 46. Post-condition for action actionC

As a consequence, the whole visual contract can be created by merging the pre and post conditions (Figure 45 and 46), as shown in Figure 47. Here we make explicit the changes and the instances involved, in order to avoid misunderstandings.

In this case, the specification of the action ActionC has two parameters: the target system (Myself), the identifier of the object to be processed (pid), and two points in time (pre, post). The Alloy code corresponding to figure 47 is:

```

pred ActionC(pid: Int, a: one Myself, pre, post: Time) {
  pre != post
  // pre-condition
  one p: TypeB |      pid = p.id and
                    p in a.state1_roleTypeB_List.pre and

  // post-condition
  one p: TypeB |      pid = p.id and
                    a.state2_roleTypeB_List.post = a.state2_roleTypeB_List.pre + p and
                    a.state1_roleTypeB_List.post = a.state1_roleTypeB_List.pre - p
}

```

The Alloy code can be read as following: the pre condition is that the **Id** of the **TypeB** who wants to Board is in the **state1_roleTypeB_List**. The post condition is that the **Id** is now in the **state2_roleTypeB_List** and is no longer in the **state1_roleTypeB_List**.

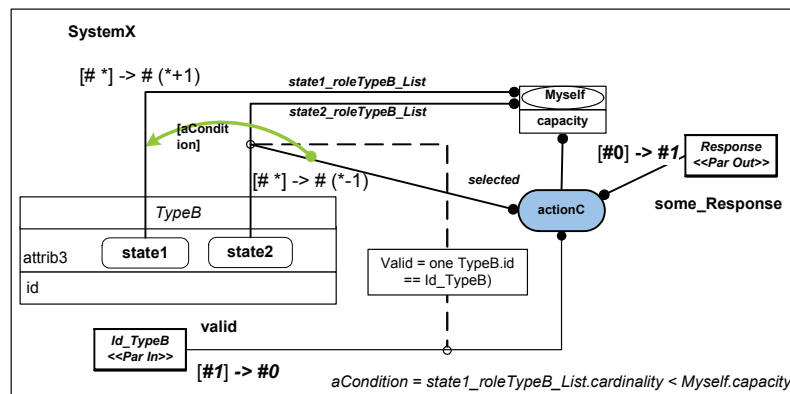


Figure 47. Visual contract for action `actionC`. It illustrates the operators «select», «change», and «transfer».

The visual contract for `actionC` is shown in figure 47. The intermediate processing is kept in the final contract in order to make the changes more understandable. Note that the constraint regarding the aircraft capacity has become a guard for a transfer of instances. The corresponding Alloy code is the following:

```

open util/ordering[Time]
sig Time {}
sig IdTypeB {}
sig attrib3StateSpace {}
one sig State2 extends attrib3StateSpace {}
one sig State1 extends attrib3StateSpace {}
sig TypeB {
  id: Int, state: attrib3StateSpace one -> Time
} {
  all t: Time | state.t in (State2 + State1)
}
fact uniqueID {
  all p, q: TypeB | p != q => p.id != q.id
  all p: TypeB | all a: Myself | p.id != a.capacity
}
one sig Myself {
  capacity: Int,
  state2_roleTypeB_List: set TypeB -> Time, state1_roleTypeB_List: set TypeB -> Time
} {
  int capacity > 0
  all t: Time | all p: TypeB | p in state2_roleTypeB_List.t => p.state.t = State2
  all t: Time | all p: TypeB | p in state1_roleTypeB_List.t => p.state.t = State1
  all t: Time | int capacity >= #state2_roleTypeB_List.t + #state1_roleTypeB_List.t
  all t: Time | no p: TypeB | p in state2_roleTypeB_List.t and p in state1_roleTypeB_List.t
  all t: Time | all p: TypeB | p !in state2_roleTypeB_List.t => p in state1_roleTypeB_List.t
}
pred ActionC (ps: set IdTypeB, a: one Myself, pre, post: Time) {
  pre != post
  // pre-condition
  all pid: ps | one p: TypeB | p.id = pid and p in a.state1_roleTypeB_List.pre
  // post-condition
  some psl: set TypeB | all pid: ps | one p: psl | p.id = pid and
  // state2 list increased
  a.state2_roleTypeB_List.post = a.state2_roleTypeB_List.pre + psl and
  // state1 list decreased
  a.state1_roleTypeB_List.post + psl = a.state1_roleTypeB_List.pre
}

```

This model can now be analyzed by the Alloy Analyzer tool [JACKSON, D. 2002]. The modeler should then select one action or one property to be verified (we call this a scenario). This requires writing some content in the *commands* section of the Alloy specification—as explained in section 9.1.1. The following are two examples of scenarios that can be tested in the specification shown above:

```

run ActionC for 7 but 2 attrib3StateSpace // Execute ActionC and verify the system is consistent in a search space
check uniqueID for 7 but 2 attrib3StateSpace // Verify this property is not violated in this search space

```

A scenario can be more complex. It is possible, for example, to:

- verify a sequence of several actions,
- verify a number of several properties simultaneously, or
- check that the system presents certain properties during the evolution (the execution of the actions). This scenario is a mix of the two preceding ones.

When a sequence of actions is specified, it must be clear what are the inputs and outputs. In particular, the modeler should determine what outputs from one action become the inputs of another action. The Alloy Analyzer connects all the unconnected inputs to object instances in the search space (undeterministic choice); these objects must only be of the same type (information object). This guarantees the robustness of the system.

By executing scenarios that reflect the way we expect your system to behave, the modeler is able to verify that it will indeed do so—long before final implementation—. It should be emphasized that scenarios can be rerun at any time in the development effort, as long as the portion of interest is syntactically legal.

9.1.6 Representing Time for Execution of Sequential VCs

The manual strategy and the original approach to translation targeted individual actions. However, for a practical MDE approach to be feasible, the translating tool should be able to compose a series of actions of the system under study. Interesting behaviors are seldom composed of a single type of actions. Furthermore, the composition can take many forms; in order to deal with complex compositions the notation must provide some operational semantics (e.g. partial order, total order, workflow patterns, etc.). This is not the focus of this research work, so we support only the basic AND and OR operators for sequences.

It is not possible to write an Alloy equivalent of the Visual Contract for a series of actions without introducing the notion of time. Alloy creates and explores the tree of states that are possible from the specification. Time enables differentiating the model instances before and after the action executes, and can be modeled as an ordered vector. Figure 10 shows the code modified for including time.

The approach is the following: a series of actions requires sharing points in time. These share points correspond to some instant between the end of one (or some) action(s), and the start of the following one(s). Then, the modeler can describe the actions and the nature of the sequence, and demonstrate that such a sequence produces a system instance that is still valid. As the elements required for the specification of the system are all the same (IOs, SAs, actions, Myself), the changes remain at the level of cardinalities, as explained in chapter 6. However, some additional runtime parameters have to be configured by the modeler because the nature of the validation tests cannot be automatically determined by the translation algorithm. Examples of parameters to configure are: size of the search space (number of instances per type), sequencing of actions, kinds of properties (properties or functions) to validate.

The Alloy Analyzer tool analyzes a state space specified by the modeler as a scenario. This allows the modeler to find inconsistencies in the model in the specified search space.

9.1.7 The VCML Representation

We have created a XML version of the Visual Contracts that we call Visual Contract Modeling Language (VCML). We created the VCML schema from the metamodel presented in section 8.3.

The general mapping strategy from the graphical form of Visual Contracts and its counterpart written in VCML is straightforward, as illustrated in figure 48. It represents a set association, its attributes and part of the hierarchy of types.

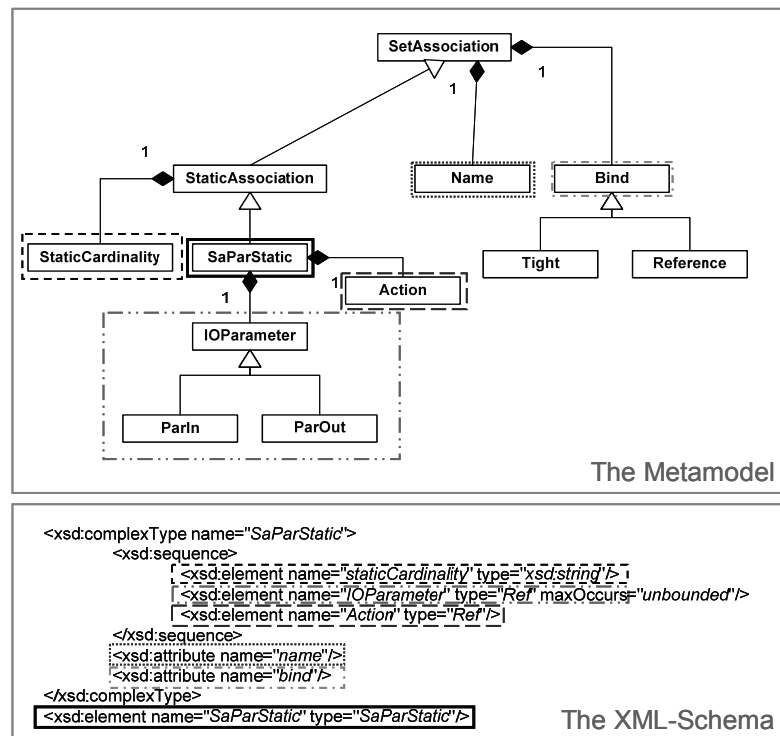


Figure 48. Mapping between model and the corresponding XML schema

A concrete translation to XML is shown in figure 49. It represents the Visual Contract for action Board (from the study case *Passenger Control Boarding System*, chapter 9) to XML. This Visual Contract requires the use of information objects, set associations, myself, parameters, states, and relational operators select and transfer. The translation of the transfer operator is shown with the arrows.

9.1.8 Results

We built a Java application that parses the contracts written in VCML, validates them, and then generates the Alloy model. This model is then fed to the Alloy Analyzer. As we did not build a front-end capturing tool for the notation, we simulated the generation of the VCML from the VC diagrams.

The Alloy Analyzer explores the constrained state space of instances for each object as indicated in the *commands* section of the Alloy model and execute the sequence. The Alloy Analyzer tells if any inconsistency with facts and assertions is found in this state space.

After the exploration of the state space, the Alloy analyzer can confirm whether the system model is consistent or not. It can then be executed, in order to be validated by a stakeholder. An example of a short execution (with only 2 points in time) is illustrated in figure 50. The modeler or any other stakeholder can then validate the behavior of the system, if the behavior represented via these snapshots satisfies her needs.

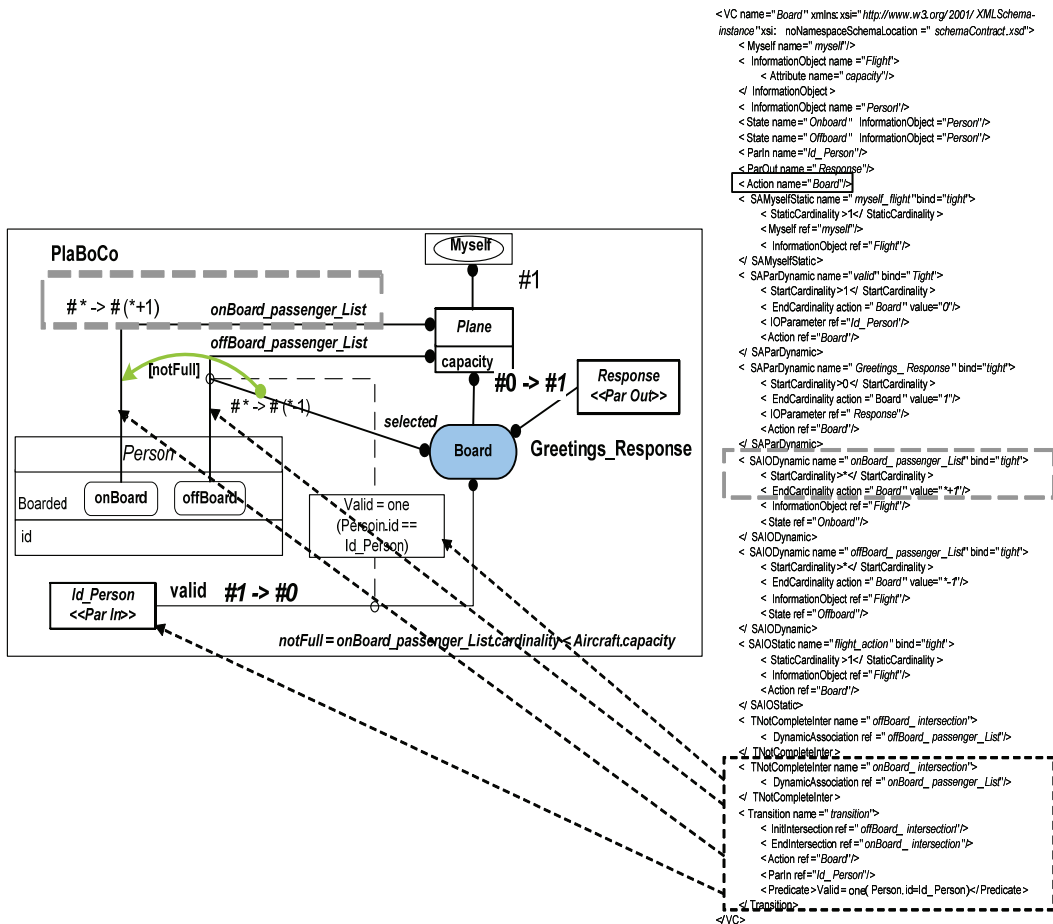


Figure 49. Mapping between Visual Contract for action Board and the corresponding VCML output.

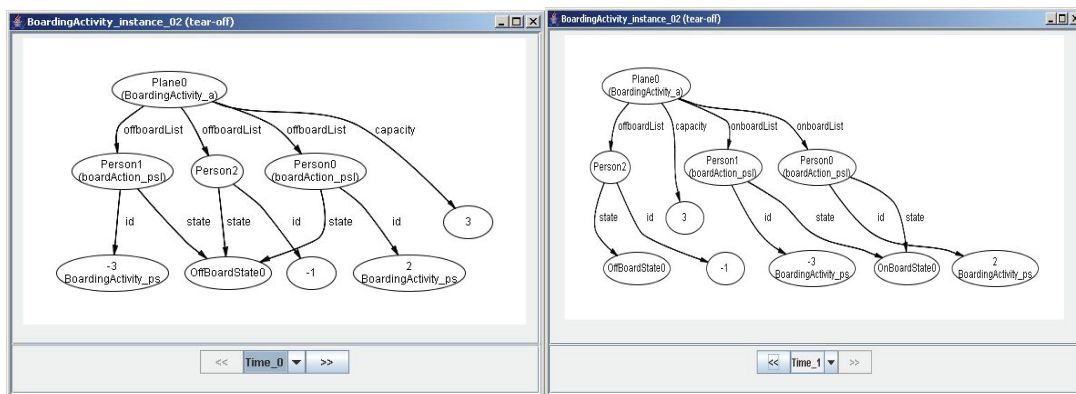


Figure 50. Results of simulating the action Board in the Alloy analyzer. At Time0 the passengers (Person0, Person1, and Person3) have already checked-in. At Time1 two of them (Person0, Person1) effectively embark on the plane. Capacity of the plane is 3

9.2 Summary

We developed a translation strategy, a translation tool and a set of heuristics in order to verify and validate systems specified with Visual Contracts in constrained search spaces.

The strategy to achieve the translation from Visual Contracts to Alloy was successful for several case studies. The semantics of our visual contracts is precisely defined in the first-order language Alloy[JACKSON, D. 2002]. The tools that support Alloy also make possible the illustration of what the service specification represents. This is done by generating instance models that are compatible with the system specification (**verification**). As a consequence, the model can be more useful because the modeler can see what the specification means, hence she can make sure that the system specification corresponds to their needs (**validation**). In addition, the validation takes place at a high level of abstraction. Our tool currently treats only a subset of the elements used in Visual Contracts; it satisfies most of the needs of the case studies (see chapter 9). The tool we developed to translate Visual Contracts to Alloy uses the metamodel presented in Chapter 8 in order to guarantee the extensibility and flexibility of the solution.

PART III – Praxis of Visual Contracts

In this part, we apply the Visual Contracts to concrete cases, and discuss further the three dimensions of the notation (syntax, semantics and pragmatics) from the viewpoint of a practitioner.

In Chapter 10 we present a set of case studies. Each example illustrates different aspects of our approach.

10 Case Studies

“A good knowledge representation language should combine the advantages of natural languages and formal languages. It should be expressive and concise so that we can say everything we need to say succinctly. It should be unambiguous and independent of context, so that we say today will still be interpretable tomorrow. And it should be effective in the sense that there should be an inference procedure that can make new inferences from sentences in our language... however, it is also important not to get too concerned with the specifics of logical notation... The main thing to keep hold of is how a precise, formal language can represent knowledge, and how mechanical procedures can operate on expressions in the language to perform reasoning. The fundamental concepts remain the same no matter what language is being used to represent the knowledge.”
Russell & Norvig – Artificial Intelligence [RUSSELL, S. and NORVIG, P. 1995]

In this chapter we illustrate the use of Visual Contracts via a set of examples of system specifications. These examples illustrate the use of Visual Contracts –from their design to formal reasoning— in different domains.

The first example, the specification of a “User login” module, introduces:

- The use of instantaneous cardinality in set-associations.
- The basic reasoning that can be done with set-associations.

The second example, the “Plan Boarding Control System”, shows in detail:

- The specification of a system whose behavior is composed by three actions.
- The concrete semantics of Visual Contracts. As the Visual Contracts are accompanied by the corresponding translation in Alloy language.

The third case study, the specification of an IT system for a video rental company, illustrates:

- The composition of actions (and of Visual Contracts).
- The successful scenarios (for each action) and the resulting Visual Contracts.
- The unsuccessful scenarios (for each action) and the corresponding Visual Contracts.
- The composition of actions (and of the corresponding Visual Contracts) for both successful and unsuccessful scenarios.
- The simplicity of the specification of compensation measures as a counter-contract of the action.

The fourth example, the “Bookstore”, shows:

- The way to apply the Visual Contracts to a system specification that is not exclusively IT-based.
- The composition of actions (and of Visual Contracts).
- The way to illustrate how different systems collaborate and exchange information.

The final case study, the “Job Yellow Pages Website”, allows us to

- illustrate the more fundamental aspects of set-associations.
- demonstrate the differences among contexts of existence (i.e. reference and concrete instances).

We consider that the diversity in the selection of the study cases give us confidence about the pertinence of our approach for system specification.

10.1 Example: A User Login

Set-associations are a fundamental notion for the creation of Visual Contracts. They are explained in detail in Chapter 7. Set-associations are declarative specifications of collections and of their relationships.

The most powerful feature of specifications written using Visual Contracts is that its declarative semantics avoids the need for loops and other forms of flow control that are required in operational semantics. This example will illustrate how to reason about loops and disjunctive loops for system modeling.

This example consists of a simple login action. The user must enter a valid Id. The user has a maximum of 3 attempts to succeed entering a valid Id.

We will first show the case where the user will fail logging in:

- after the first failed attempt, the system will not generate any definitive response. In this case, one instance of `Id_Person` that is `nonValid` is represented. This is shown in fig. 51.
- a default “try again” **response** may be sent to the user. This has not been represented in figure 51 because it was not specified.

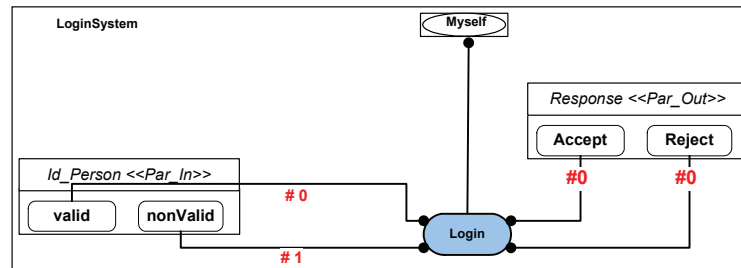


Figure 51. Logging action after a first failed attempt

- After the second failed attempt, the system will not generate any definitive response either. In this case, two instances of `Id_Person` that are `nonValid` are represented. This is shown in figure 52.

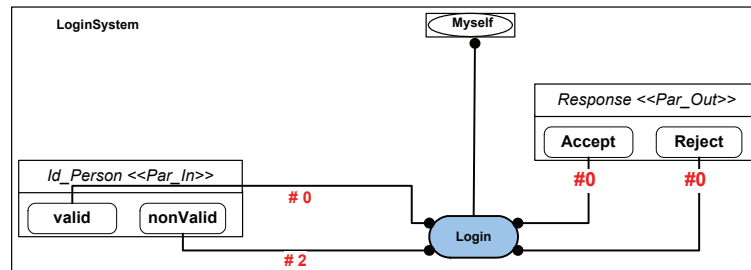


Figure 52. Logging action after a second failed attempt

- After the third failed attempt, the system will arrive to the situation described in figure 53.

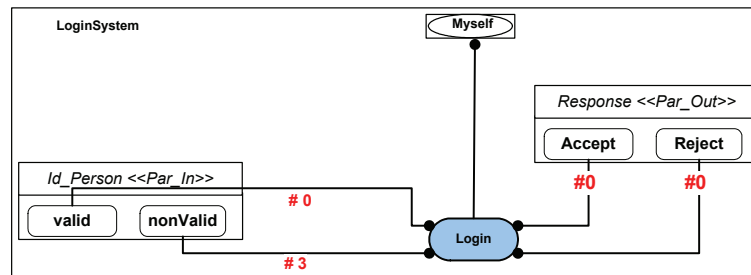


Figure 53. Logging action after a third, final failed attempt

- Next, the action will finish by generating one instance of **Response** of type **Reject**. This is shown in figure 54.

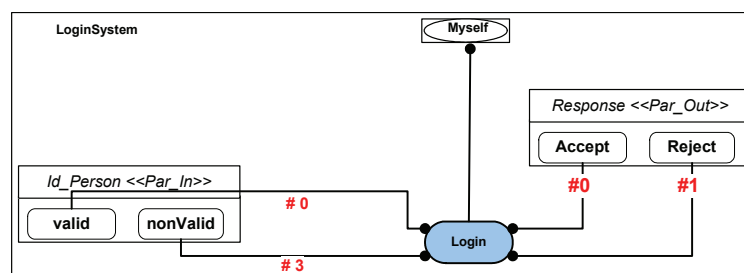


Figure 54. Logging action finished after three attempts. A response is generated.

In summary, the previous situations will produce the Visual Contract shown in figure 55. The conditions on both input parameters will be fired, and this will trigger the **LogIn** action. The **KO** predicate will then hold true. On the postcondition side, the clauses that contain the **KO** predicate will then fire. In this case, this means that the output parameter **Response** of type **Reject** will be given off.

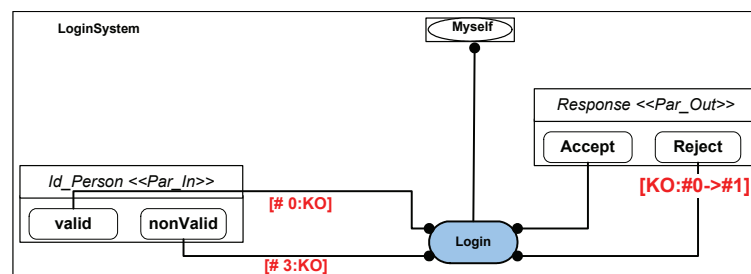


Figure 55. Visual Contract for failed logging action

Let us consider now the scenario case where the user will succeed logging in:

- After the first successful attempt, the action will count one instance of **Id_Person** that is **valid**. If the action has not received more than 2 failed attempts, the successful attempt will make the **OK** predicate **true** and the action will be triggered. This will fire the clauses containing **OK** on the Visual Contract. In this case, the definitive **Response Accept** will be obtained. This is shown in fig. 56.

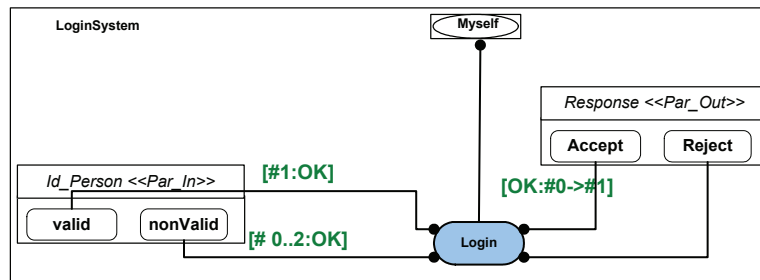


Figure 56. Visual Contract for successful logging action

By merging together the VCs for the failed and successful logging attempts, we produce the Visual Contract in figure 57.

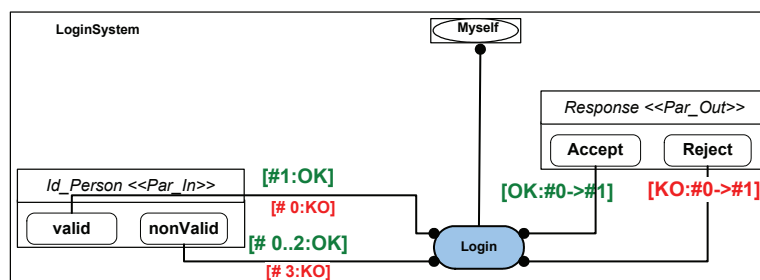


Figure 57. Aggregate Visual Contract for successful & failed logging action

As Dijkstra illustrates in [DIJKSTRA, E.W. 1976], loops are actually abstractions for a process that has been repeated for different instances.

This way of specifying the actions let the modeler room to later add details. The identification for a user (**Id_Person**) can be simple (e.g. a simple word, a magnetic card, a barcode, a digital certificate, the fingerprint, among many others) or composite (e.g. a combination of username and password, or any combination of the simple Ids we mentioned above).

Besides, the criteria for validation are also decoupled from the specification model. The model can then introduce rules for validation that can be as complex as required, without introducing complexity onto the Visual Contract specification.

10.2 Example: the Plane Boarding Control (PlaBoCo) System

This example illustrates the complete modeling and validation procedure for an IT system, as shown in figure 56. We explain the complete process of going from the specification of the invariants of the system, of the Visual Contracts for the services (actions) it renders, to the final translation to Alloy that makes possible to verify and validate the models.

First we introduce the necessary information objects (Section 10.2.1). We then define the operations Init and CheckIn (Section 10.2.2) and then Board (Section 10.2.3). The behavior of the system in abnormal conditions is not shown in this paper.

For each Visual Contract that we present, we define the Alloy equivalent. Alloy is a light weight specification language that we use to define the semantics of our notation.

10.2.1 Definition of Information Objects

For sake of simplicity, the system **PlaBoCo** knows one **Plane** only. In our representation, shown in figure 58, can be read as follows: in the system (i.e. **Myself**), there is the knowledge of one **Plane** with a capacity. This **Plane** has two **passenger_List** that can

contain multiple passengers each; one is for the **passengers** that have checked-in (**offBoard**) and the other one for the ones that have effectively boarded the **Plane** (**onBoard**).

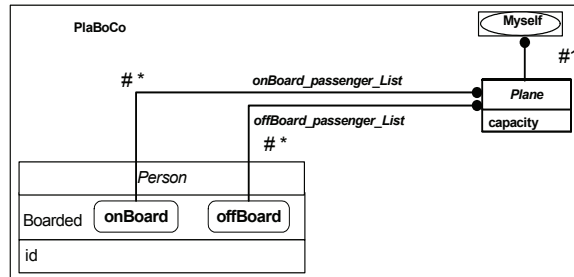


Figure 58. SEAM notation for data definitions for the PlaBoCo

The Alloy equivalent is the following:

```

module models/boarding
open util/ordering[Time]

module models/boarding
open util/ordering[Time]
sig Time { }
sig Person {
id: Int
}

fact uniqueId{
all p,q: Person | p != q => p.id != q.id
}

one sig Plane {
capacity: Int,
onboard_passenger_List:
set Person -> Time,
offboard_passenger_List:
set Person -> Time
} {
int capacity > 0
all t: Time | int capacity >= #onboard_passenger_List.t + #offboard_passenger_List.t

all t: Time | no p: Person | p in onboard_passenger_List.t and p in offboard_passenger_List.t
}

```

The Alloy can be read as follows: a set of ordered time points are defined (**sig Time**); a set of **Person** are defined (**sig Person**) with a unique identifier (**fact uniqueID**). We define also a **Plane** which has a capacity and 2 lists: **onboard_Passenger_List** and **offBoard_Passenger_List**. These lists include a relation between a person and a time point (necessary to simulate the execution sequence). Some invariants are defined in the plane: the capacity is never exceeded, and nobody can be in both lists at the same time.

10.2.2 Operations Init & CheckIn

Figure 59 shows the SEAM visual contract for operation Init. It states that the number of Person in both **passenger_List** (**offBoard**, **onBoard**) is set to zero.

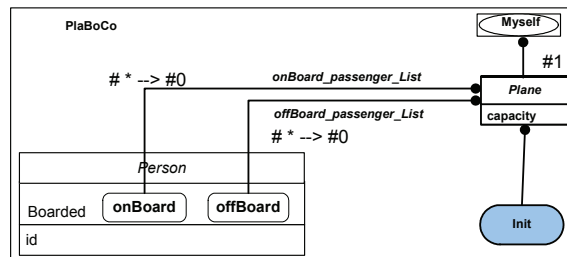


Figure 59. The SEAM contract of `Init`: the cardinality of the `passenger_List` SA changes

The cardinality of each set-association that links `passenger_List` to `Person` goes from some initial value (any, symbolized by the character ‘*’) to 0. In the practice, this means that all the instances of `Person` linked to either the `offBoard_passenger_List` or to `onBoard_passenger_List` are erased.

The action `checkIn`, not presented in this paper, assigns instances of `Person` to the `offBoard_passenger_List`.

10.2.3 Operation Board

The specification of action `Board` is the following: “The plane preconditions are: a) an input parameter represents the identifier of the person that desires to go on board, b) this person has already checked-in, and c) the number of people onboard has not reached the maximum capacity of the plane. The post condition is that the person is now onboard. In addition, the system emits a message confirming the entry of the person into the plane”.

Before creating the Visual Contract for action `Board`, we illustrate the action by making two snapshots: one before and one after the operation `Board`. The situation would be as shown in figures 60 and Figure 61, respectively.

In the precondition, there is originally an instance of `Id_Person` that is considered as valid. The valid condition is defined by a constraint in the diagram: the `Id_Person` should correspond to the id of only one `Person` that has already checked-in (she is in the `offBoard_passenger_List`).

During the action `Board`, the parameter `Id_Person` is validated and the corresponding instance of `Person` in the `offBoard_passenger_List` is referenced by the action via the SA selected.

In the post condition, Figure 61, the selected instance of `Person` will be transferred to the `onBoard_passenger_List`; the number of instances of IO `Person` that are on the `onBoard_passenger_List` shall increment by one (supposedly the one that has been admitted in the precondition); the cardinality change in the corresponding set-association symbolizes this. Simultaneously, the `offBoard_passenger_List` is decremented by one. Finally, a response message is emitted, indicating the success of the operation (represented by the set-association `Greetings_Response`).

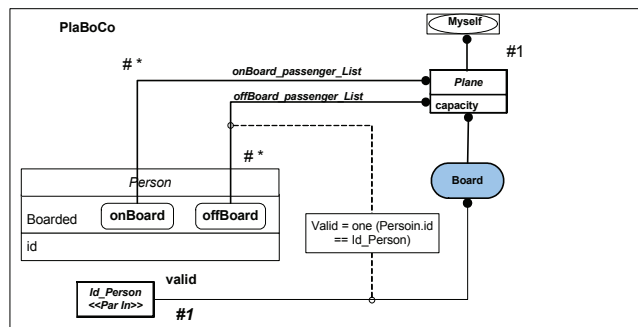


Figure 60. Precondition for action Board

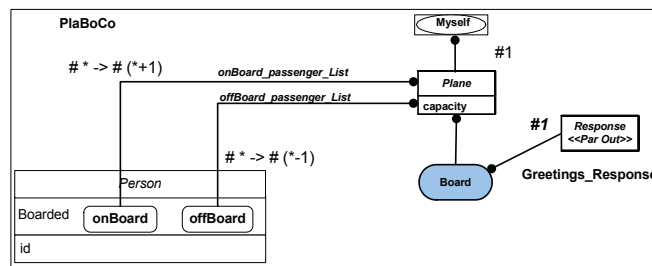


Figure 61. Post condition of action Board

As a consequence, the whole visual contract can be created by merging the pre and post conditions (Figures 60 and 61), as shown in Figure 62. Here we make explicit the changes and the instances involved, in order to avoid misunderstandings.

The instance of the `offBoard_passenger_List` that corresponds to the `Id_Person` is represented by the SA selected, that exists in the context of action `Board`. Remark that this temporary information does appear neither in the pre nor in the post condition.

Notice also that the constraint regarding the `Plane` capacity has become a guard for a transfer of the instance selected of IO `Person` from `offBoard_passenger_List` onto the `onBoard_passenger_List`. This happens because in our visual contracts, conditions can only be expressed in relations with an action, and cannot be taken just as structural axioms, as it is done in the Alloy specification.

The Figure 62 illustrates the evolution of state of the IT system during the contract execution. It is the result of the execution of the Alloy code presented below. It shows a scenario where the plane has two people (`Person0`, `Person1`) in the `offBoard_passenger_List` before the action execution (at time `Time0`). One passenger (`Person1`) actually boards the plane, as can be seen in the nodes that represent the state after the action execution (at time `Time1`). The person `Person0` has changed to the `onBoard_passenger_List`.

The Alloy code corresponding to figure 62 is:

```

pred Board(pid: Int, a: one Plane, pre, post: Time) {
  pre != post
  // pre-condition
  one p: Person |      pid = p.id and
                       p in a.offboard_passenger_List.pre and

  // post-condition
  one p: Person |      pid = p.id and
                       a.onboard_passenger_List.post = a.onboard_passenger_List.pre + p and
                       a.offboard_passenger_List.post = a.offboard_passenger_List.pre - p
}

```

The Alloy code can be read as following: the pre condition is that the Id of the **Person** who wants to **Board** is in the **offBoard_passenger_List**. The post condition is that the **Id** is now in the **onBoard_passenger_List** and is no longer in the **offBoard_passenger_List**.

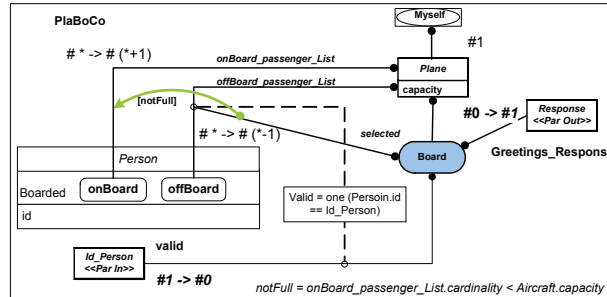


Figure 62. Visual Contract for action Board. It illustrates the «change» operator

Now we are able to differentiate the lists of passengers: one for **onBoard**, and another for **offBoard** ones. IOs represent the “casts” however the instances themselves exist only in the SAs, drawn as unidirectional lines. All instances exist in a context, as shown in figure 63. The system itself has to be made explicit –via the **Myself** IO—because set-associations must always be drawn in order to establish how many instances of each IO are present. Thus, one **y** exist in the context of this simple IT system (SA linking **Myself** and **Plane** IOs has cardinality #1); 2 **Person** (**passenger**) lists exist in the context of a single **Plane** (i.e. **onBoard_passenger_List** and **offBoard_passenger_List**). Multiple **Persons** may exist in the context of each **Passenger_List**. As a **Plane** is a physical system, we should reason by taking into consideration the fact these lists must limited by the capacity of each **Plane**. We have simplified the problem for sake of clarity.

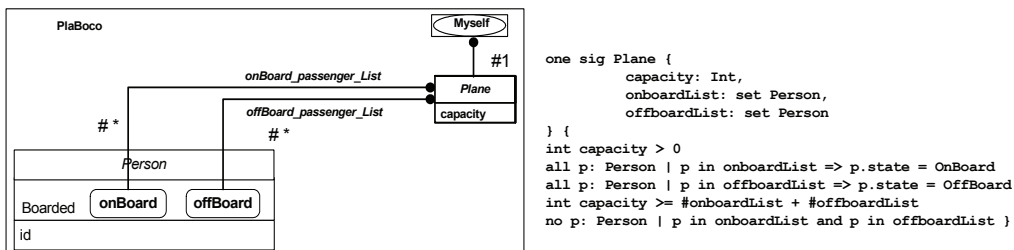


Figure 63. SEAM notation for property definitions in the PlaBoCo specification

The summary of the complete process, from specification to validation, is illustrated in figure 64. Step 1 is the generation of the Visual Contracts, which will be processed –in step 2— in the form of VCML, to generate the Alloy model that will be finally verified and validated in step 3. We included borders in steps 2 and 3 of Figure 64 that are useful to find the correspondence among the VCML model and the Alloy model. This correspondence is, however, non-linear and some details of the translation are due to consistency rules or rules about collections. Understanding the whole translation process can be understood only from a detailed study of the primitives (explained in Chapter 6), the operators (explained in Chapter 7), and the rules (explained in Chapter 8) of Visual Contracts.

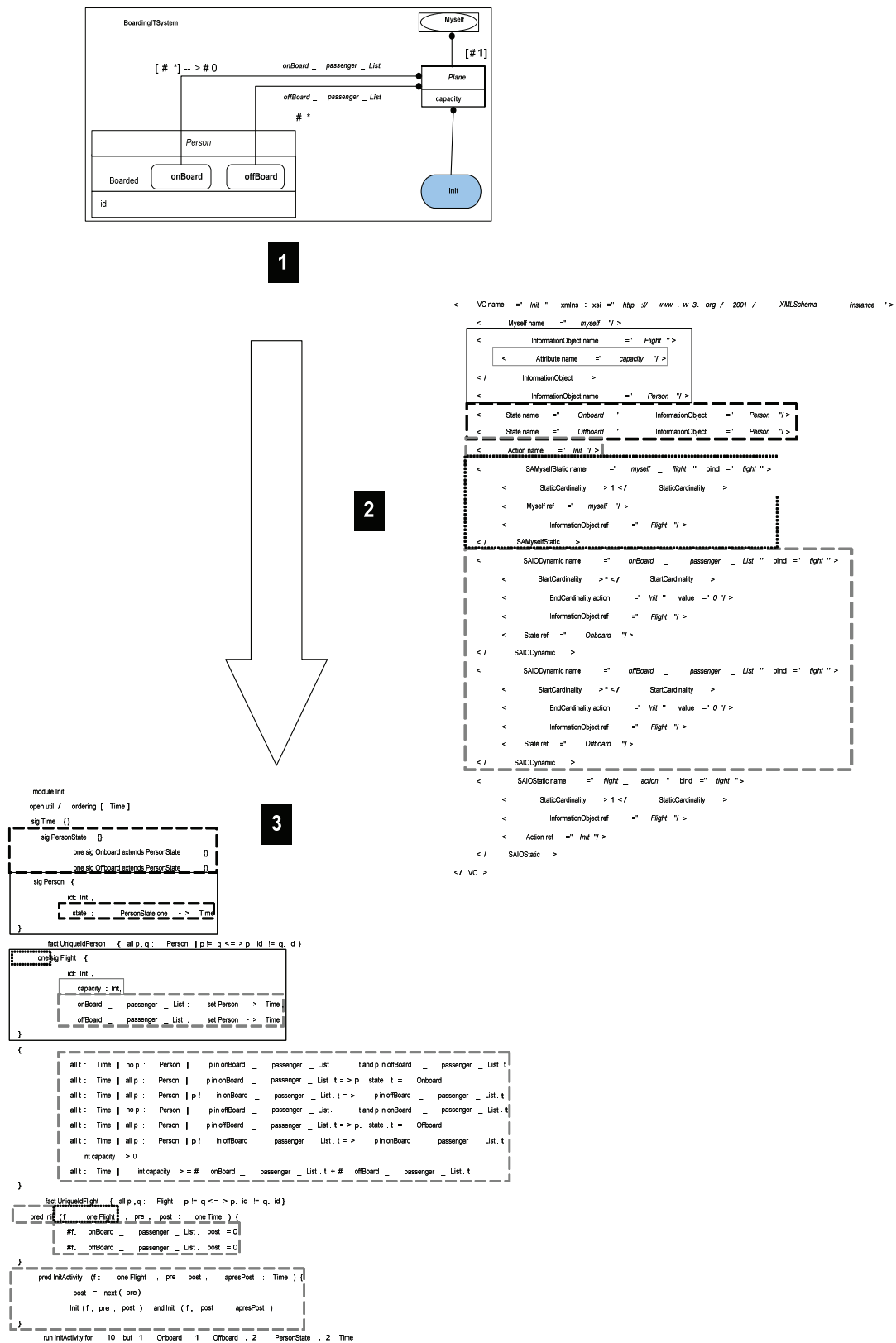


Figure 64. Complete translation example for action Init. From left to right: Visual Contract, VCML, and Alloy model. The border lines indicate the correspondence among the models of steps 2 and 3.

10.3 VideoStore PORT(Point-Of-Rent Terminal)

This example illustrates the use of composite actions in specifications. From the definition in section 7.2, composite descriptions are non-transactional. This means that compensation measures should be designed. We will also develop a whole VC in order to compare the two approaches and assess their complexity and usability.

The PORT system will be used by a store that rents videos. Our customer wants us to specify the new information system terminal (PORT), that will allow automate the renting process. A brief description of the system is the following:

A (valid) user can use the PORT in order to rent one or more (valid) videos. Once the complete list of (valid) videos is entered into the PORT, the user can either confirm (commit) or cancel the rental operation.

A number of assumptions are done, namely:

- A user is a member, and it has therefore an identification as a member
- Each video also has a video identification signature, and it is normally accessible to users.
- Each loan corresponds to one renting operation made by a single user. However, a user may loan several videos in a single loan, and also make several loans (of several videos each) in a series.

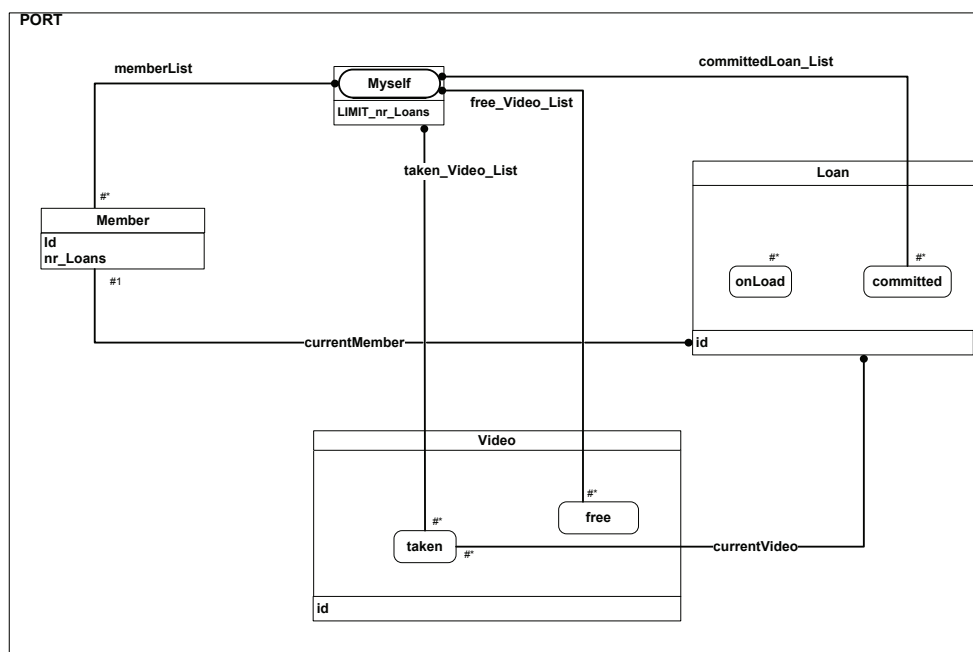


Figure 65. Visual Contract domain model

From the point of view of the domain model, the PORT system can be understood as a series of lists: one for the members, two for the videos, and one for the loans. The members can be either active or inactive (suspended, out-of-the-country, etc.). The videos can be **available** (on-the-shelf) or unavailable (**taken** by a member during the rental operation, or may be in maintenance). The loans are active (either already done—**committed**—or in the process of renting—**onLoad**—) or archived (loans done in the past, stored for legal reasons or for creating statistics). These different sub-types and states and the corresponding lists are shown in the domain model (figure 65).

Note that some fields have been added to the IOs **Myself** and **Member**. As we want to guarantee a fair use of videos, a rule will be introduced in the system in order to limit the number of videos rented per member at any given point of time. This fact is reflected by the constant **LIMIT_nr_loans** in **Myself**, whereas the current amount of videos rented by a **Member** is represented as **nr_Loans**.

10.3.1 Successful Scenario

In this section we will explain what happens when the process finishes successfully. Figure 66 represents the composite behavior of this scenario. Each action in the diagram corresponds to a Visual Contract, as explained in section 6.3.1. Note the difference among information objects that are perceived internally and those that are exchanged with the user (environment).

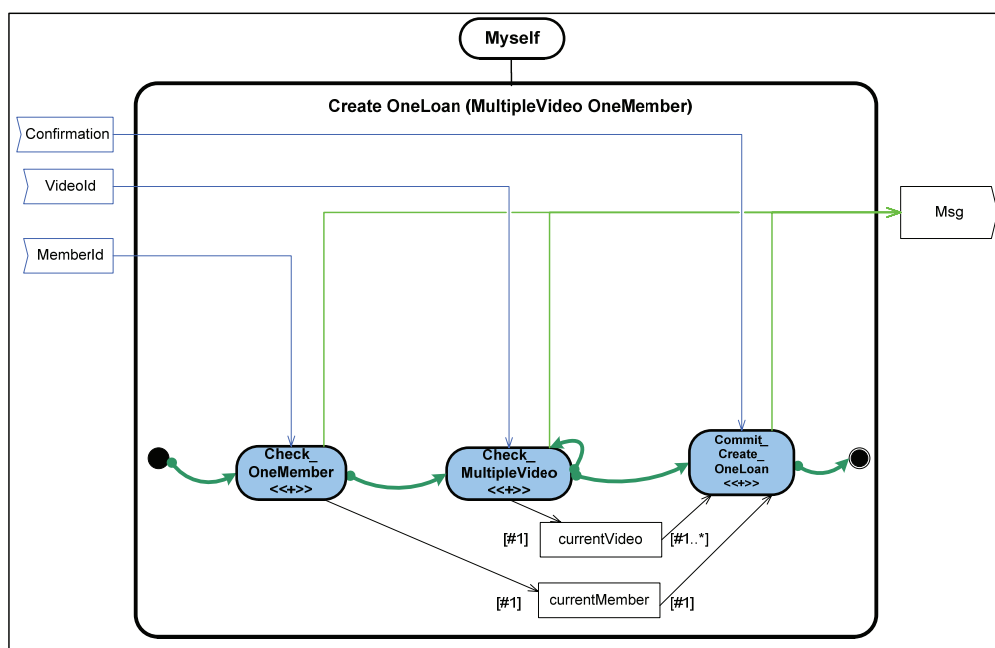


Figure 66. Composite Visual Contract for a successful loan made using the PORT system

Loan Creation

The creation of a loan is implicit in this composite action. Therefore, it is not shown in figure 66. It is a simple action but one that establishes the context that subsequent actions will use. On the other hand, a single **Create_OneLoan** action is not meaningful by itself. Figure 67 illustrates action **Create_OneLoan**.

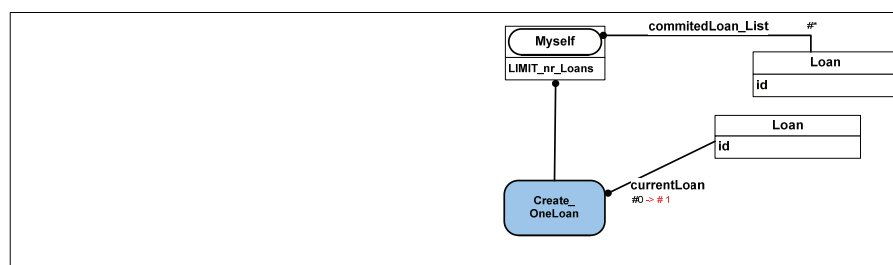


Figure 67. Visual Contract for action Create_OneLoan

Validation of Member

Now we can proceed to validate member information via the **Check_OneMember** action, shown in figure 68. In this case, the system looks for **Member_Id** in the list of members **Member_List**. If it is found—true in this case, because we analyze the successful case—then the loan **currentLoan** will create a reference to this member (**currentMember**).

The creation of a reference is essential, as it introduces the semantics consequences that are appropriate for this kind of binding. In this case, the deletion of the loan **currentLoan** will not force the system to delete the corresponding member **currentMember**.

Once the **member** has been validated, the **PORT** will emit the corresponding message **Msg**.

Validation of Videos

Now we can proceed to validate video information via the **Check_MultipleVideo** action, shown as the Visual Contract of figure 61. In this case, the system looks for each **Video_Id** in the list of free videos **freeVideo_List**. If it is found—again true, because we analyze the successful case—then the loan will create a reference to each of the free videos. The **PORT** will then change their state to taken, in order to avoid any collision problem with other users attempting to rent the same **videos**.

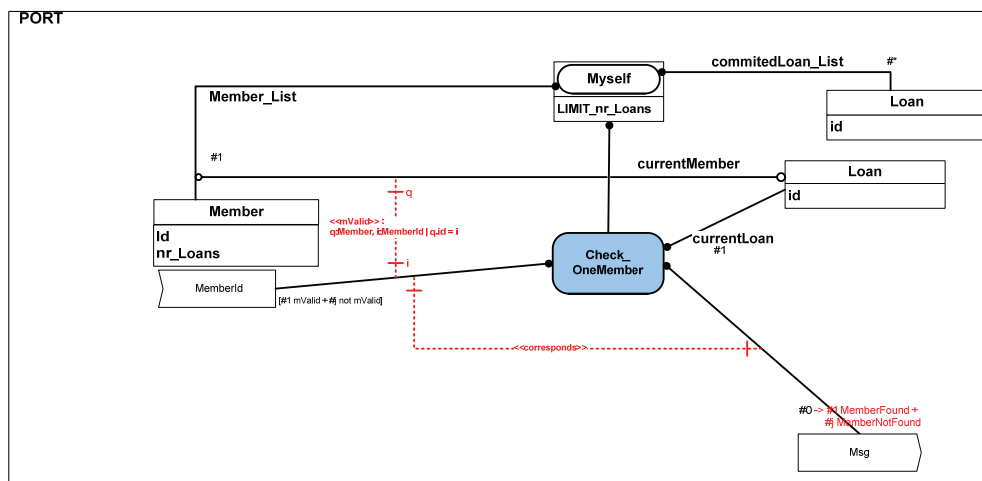


Figure 68. Visual Contract for operation Check_OneMember

The creation of a reference to each video is essential, as it introduces the semantics consequences that are appropriate for this kind of binding. In this case, the deletion of the Loan **currentLoan** will not force the system to delete the corresponding videos in the list **currentVideo_List**.

Once the whole list of videos has been validated, the **PORT** will emit the corresponding message **Msg**.

Committing the preceding actions

The commit action is a simple pattern (explained in Section 7.3). It is useful for confirming/cancelling the action that is taking place. For the sake of space, we will not include it in this description. However, the commit pattern is often not displayed isolated but in the context of the action itself, in order to be meaningful.

Visual Contract as a Whole vs. Visual Contract as a Composite

The summary of the actions in figure 66 can also be shown in the form of a Visual Contract. Figure 69 shows the summarized VC that was obtained from the fusion of figure 67 to 68 (plus the commit pattern).

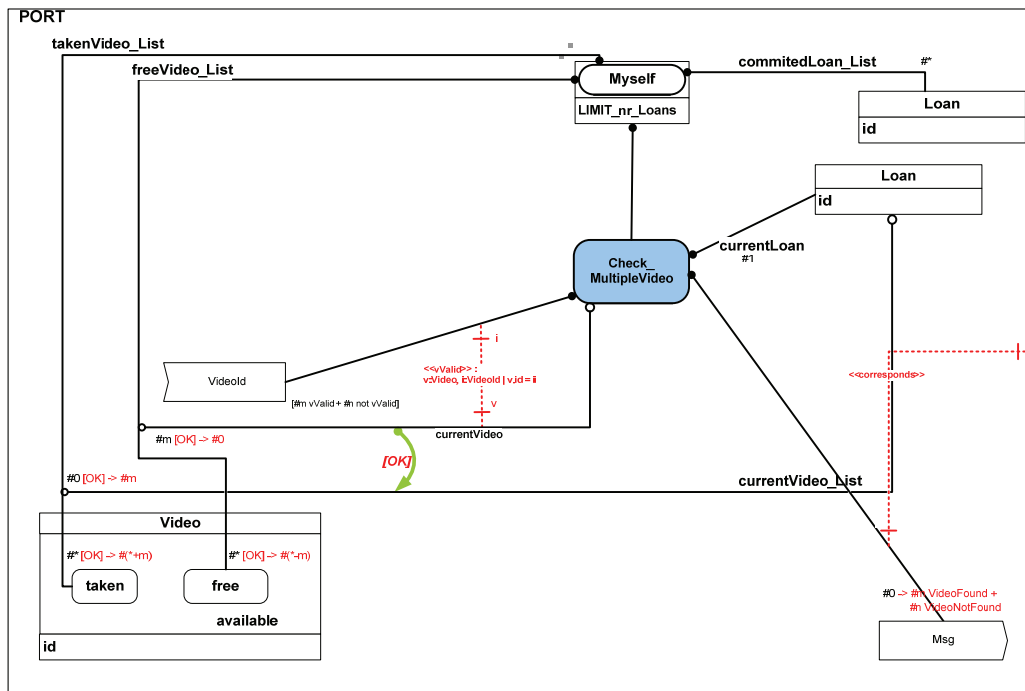


Figure 69. Visual Contract for operation Check_MultipleVideo

It is clear that this is a declarative artifact that contrasts with the operational model of figure 66.

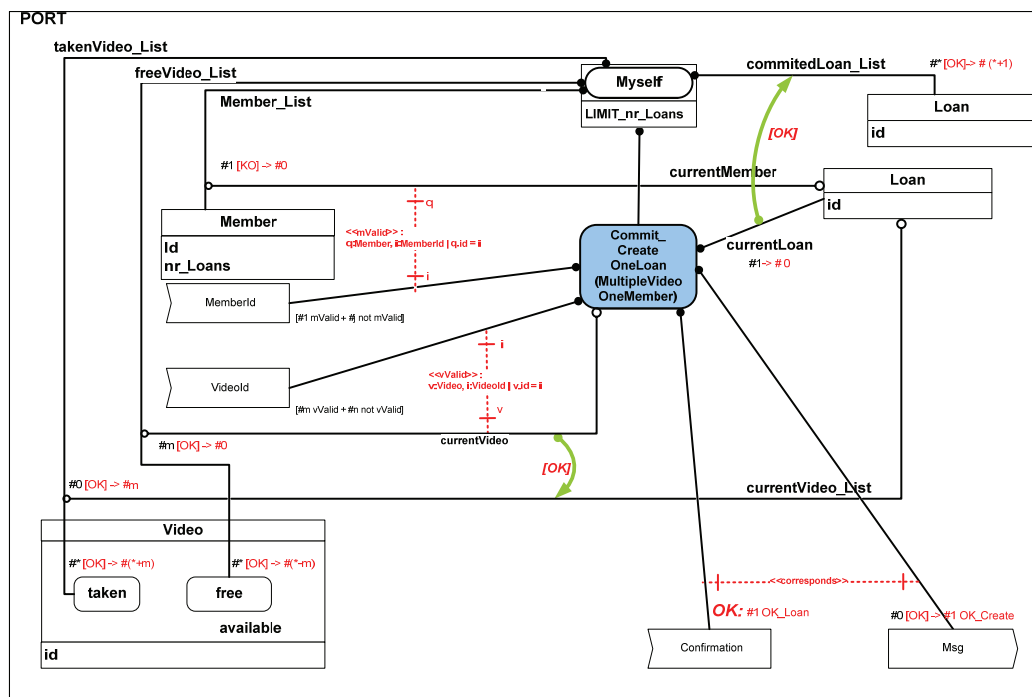


Figure 70. Resulting Visual Contract for the set of operators Create_OneLoan, Check_OneMember, Check_MultipleVideo, and Commit

10.3.2 Non-Successful Scenarios

Figure 71 presents the composition of Visual Contracts required to deal with the different non-successful scenarios. Note that the symmetry is very natural, according to our design goals for the notation. This will help in dealing with the complexity of the task.

The event **Env.Cancel** corresponds to the input of canceling events from the environment.

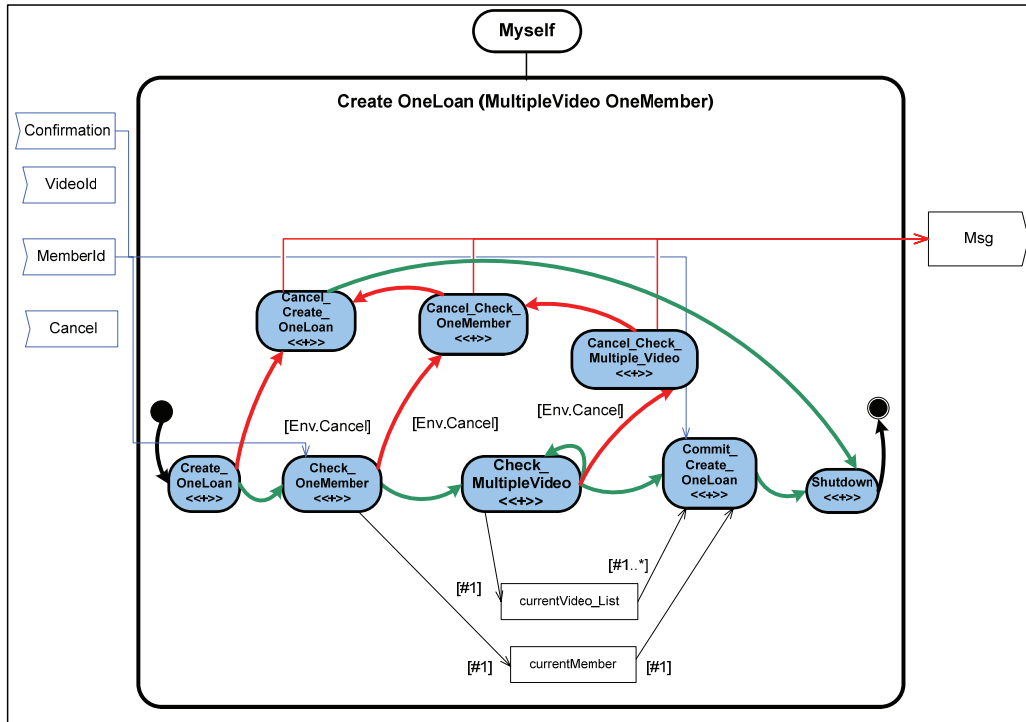


Figure 71. Composite Visual Contract for the PORT system, including error handling and compensation measures

Canceling Loan Creation

The cancellation of the creation of a loan is the simplest canceling operation. It only erases the reference to the **currentLoan**, as shown in figure 72. This is the only compensation measure that generates a message, as it will always execute as the last action in the sequence (see figure 71).

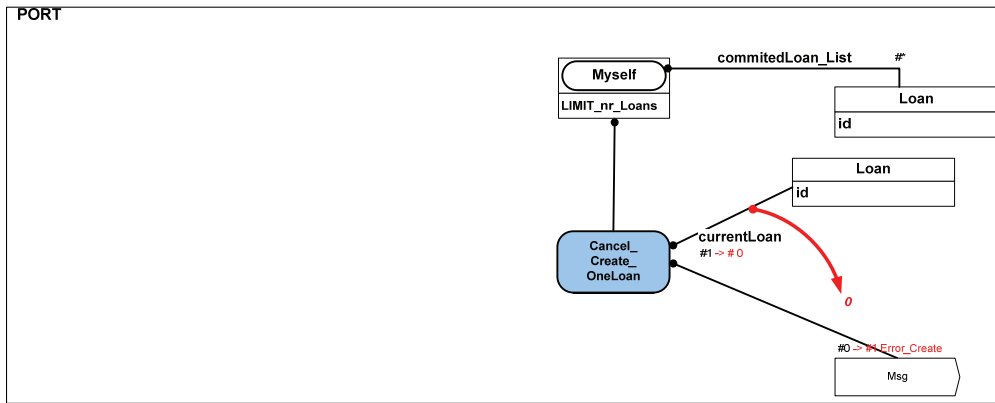


Figure 72. Visual Contract for operation Cancel_Create_OneLoan

Cancel Validation of Member

In the case of the operation **Cancel_Check_OneMember**, shown in figure 73, the system erases the member **currentMember** from the loan **currentLoan**. This corresponds to the semantic consequences that are appropriate for this kind of binding.

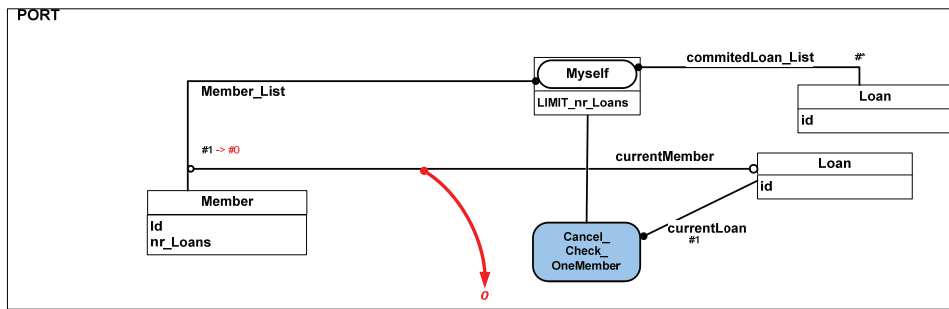


Figure 73. Visual Contract for action Cancel_Check_OneMember

Cancel Validation of Videos

If the cancel action takes place after the video information has been validated then the operation **Cancel_Check_MultipleVideo** must be executed. This canceling action is shown in the Visual Contract of figure 66. In this case, the system takes each **video_Id** in the list **currentVideo_List** and put its back in the list of free videos **freeVideo_List**. This will make the videos available once again for other members, minimizing collisions and unnecessary locking problems with other users attempting to rent the same videos.

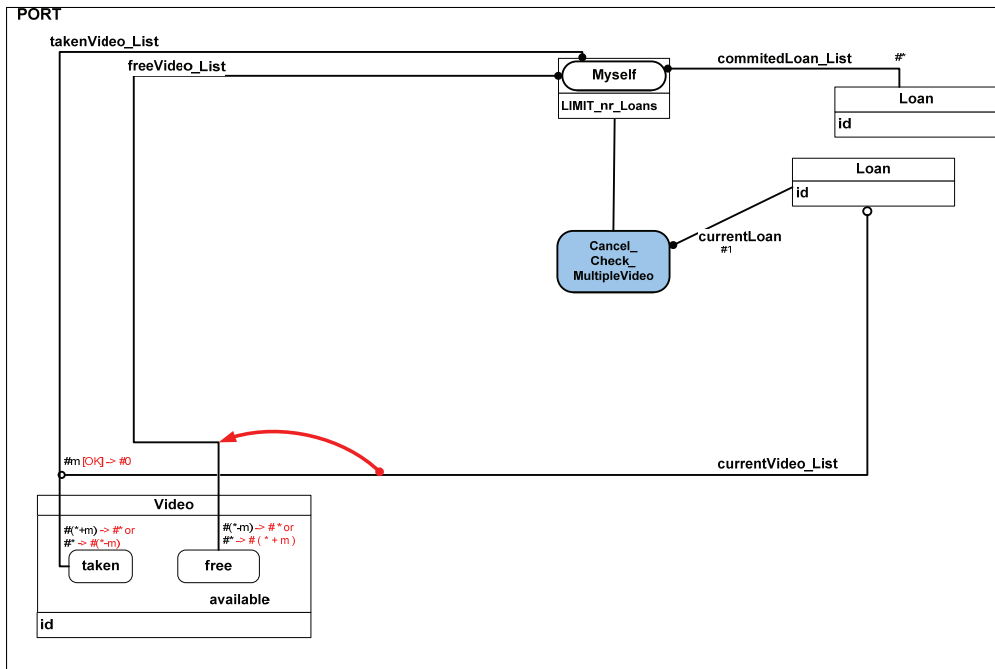


Figure 74. Visual Contract for operation Cancel_Check_MultipleVideo

Canceling the Commit of the Preceding Actions

Because the commit action is committing, it cannot include a cancel event. It is, thus, omitted from this description.

The contract that summarizes the various, specific cancel actions is the following:

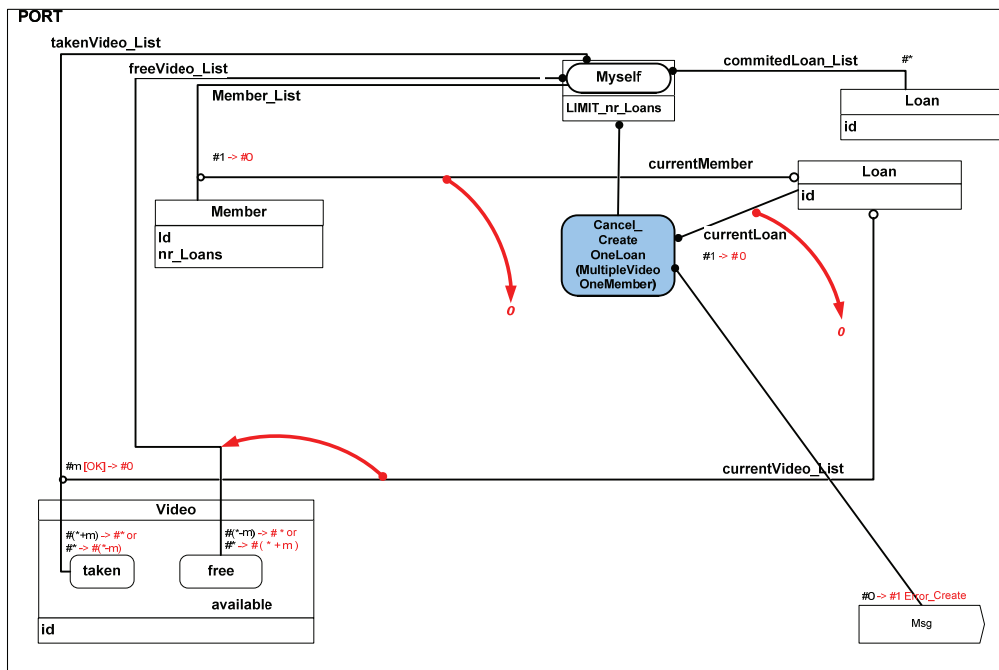


Figure 75. Visual Contract for the composite action of figures 64 to 66

If we splice all the different scenarios, according to the ordering semantics of figure 63, we obtain the VC in fig. 76.

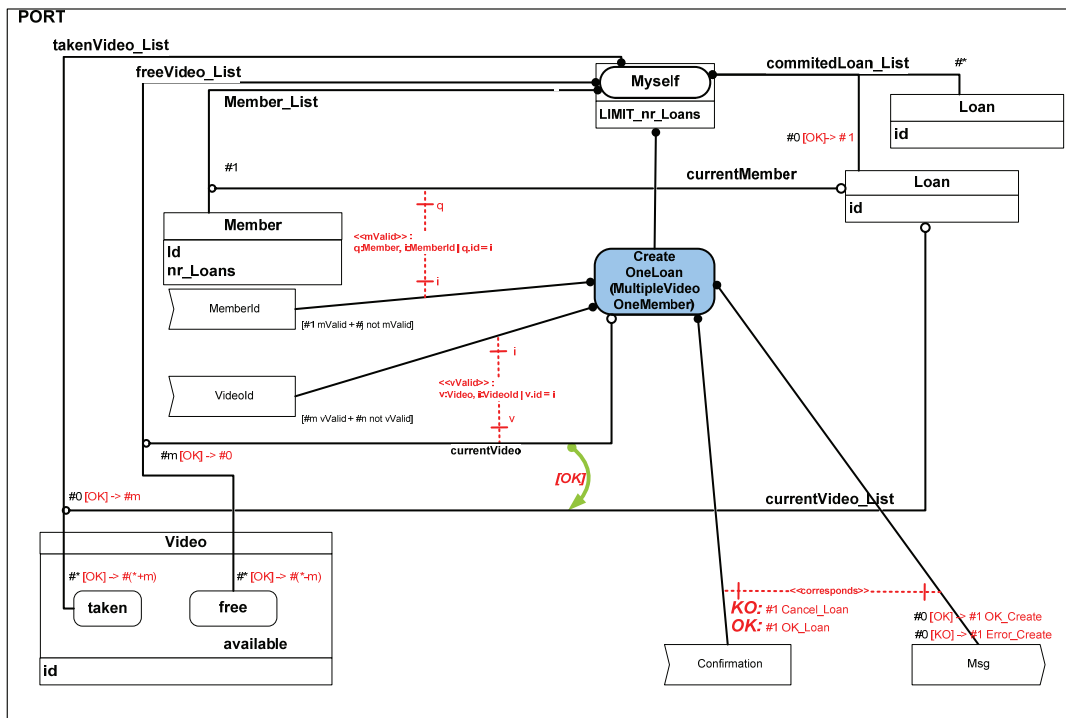


Figure 76. Global Visual Contract for successful and unsuccessful scenarios of the PORT system. It was extracted from figures 70 and 75

Note that the canceling action executed at last (**Cancel_Create_OneLoan**) is responsible for emitting the sole message **Msg**.

10.4 BookStore

This example illustrates the use of the Visual Contract notation in the broader context of SEAM methodology. Our goal is to demonstrate that the Visual Contracts can address the needs of the SEAM models for businesses and organizations.

The figure 77 shows the **Bookstore** and its customer collaborating in order to achieve a joint action **SaleAction**. **Bookstore** and **Customer** are **working objects**. Working objects represent physical system, as compared to information objects, that model information about the properties of the system. The roles **aSeller** and **aBuyer** are typical of supply chains. Actually, these roles will be distributed along the supply chain, and every supplier (**aSeller**) is also a customer (**aBuyer**) of the preceding link on the chain.

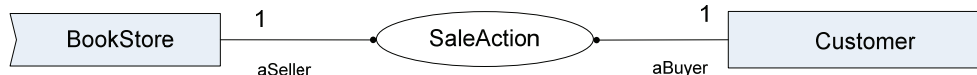


Figure 77. SEAM notation for a sale operation.

The details of the information objects required to collaborate in the **saleAction** required are shown in figure 78.

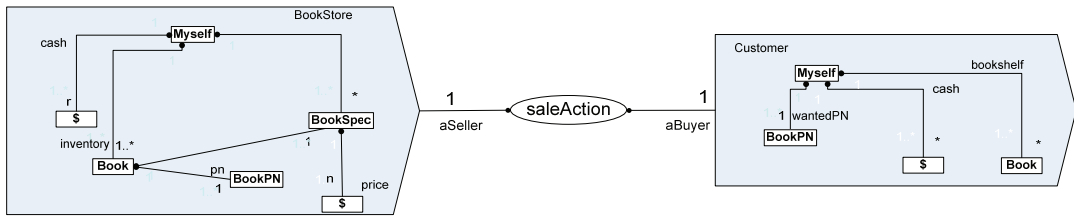


Figure 78. SEAM notation for a sale operation, expanded with the domain models for each working object

This configuration corresponds to the Alloy model shown below:

```

module bookstoremarket
open util/ordering[Time]

sig Time {}
sig String {}

sig Book {
  id: String,
  price: Int
}

fact uniqueID {
  all b: Book | int (b.price) > 0
  all b1, b2 : Book | b1 != b2 ==> b1.id != b2.id
}

lone sig BookStore {
  catalog: Book set -> Time,
  cash: Int one -> Time
}

lone sig Buyer {
  book: Book lone -> Time,
  money: Int one -> Time }
    
```

In order to make more explicit the way the joint action **SaleAction** takes place, it is important to establish the preconditions and postconditions. As a first step in that direction, we develop the informal Visual Contract displayed in figure 79.

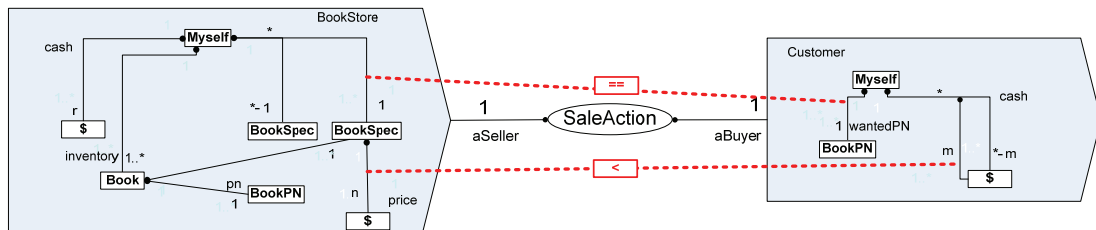


Figure 79. Informal Visual Contract for the operation saleAction. Ad hoc operators are used to link information objects that are exchanged.

In order to formalize the notation, we introduce two transfer operators, as shown in the figure 80. The accompanying clauses contain the predicates that should be satisfied in order for this contract to succeed.

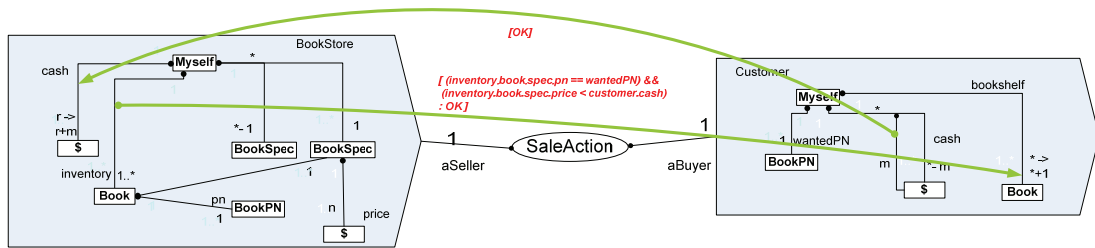


Figure 80. Visual Contract for the operation `saleAction`. The transfer among the two systems is shown explicitly

As the action `saleAction` is still seen as a single action, the contract is apparently simplified. However, the expression of the states (pre- and postconditions) are more complex. The Alloy model can be written as follows:

```

pred saleAction (b: one Book, store: one BookStore, buyer: one Buyer, pre, post: Time) {
  // invariant
  post = next(pre)

  // pre-condition
  no buyer.book.pre
  b in store.catalog.pre
  #store.catalog.pre > 1
  int buyer.money.pre >= int b.price
  int store.cash.pre >= 0

  //post-condition
  b = buyer.book.post
  store.catalog.pre = store.catalog.post + b
  int (buyer.money.post) = int (buyer.money.pre) - int (b.price)
  int (store.cash.post) = int (store.cash.pre) + int (b.price)
}
    
```

Let us now describe the role of each one of the working objects in the context of the `saleAction`. In order to proceed, the Visual contract for each one of the working objects is developed. This minimizes the complexity of the predicates that describe the system while augmenting a bit the complexity of the composition. The resulting VCs are shown in the figure 81. The dotted lines represent the causality among messages.

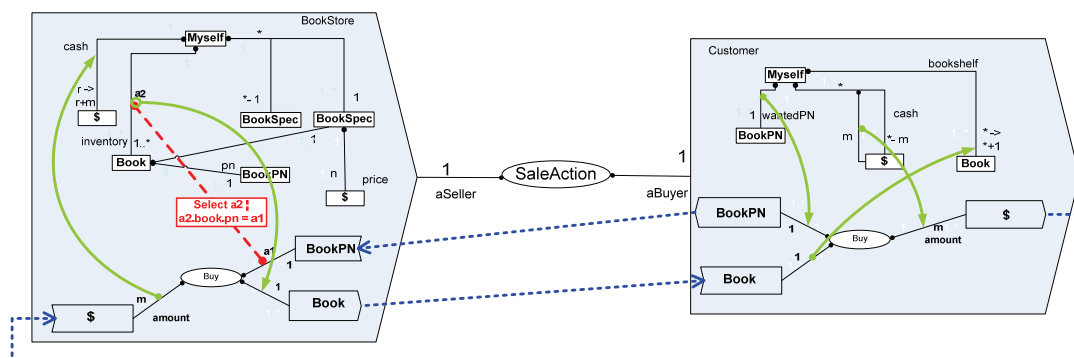


Figure 81. Visual Contract for the operation `saleAction`. The changes are local and the transfer is done via connectors among the parameters that the systems exchange.

In this case, the Alloy Model is the following:

```

pred sellAction (b: one Book, store: one BookStore,
  buyer_book: Book lone -> Time, buyer_money: Int one -> Time, pre, post: Time) {
  // invariant
  post = next(pre)
}
    
```

```

// pre-condition
no buyer_book.pre
b in store.catalog.pre
#store.catalog.pre > 1
int buyer_money.pre >= int b.price
int store.cash.pre >= 0

//post-condition
b = buyer_book.post
store.catalog.pre = store.catalog.post + b
int (buyer_money.post) = int (buyer_money.pre) - int (b.price)
int (store.cash.post) = int (store.cash.pre) + int (b.price)
}

pred buyAction (b: one Book, store_catalog: Book set -> Time, store_cash: Int one -> Time,
  buyer: one Buyer, pre, post: Time) {
  // invariant
  post = next(pre)

  // pre-condition
  no buyer_book.pre
  b in store_catalog.pre
  #store_catalog.pre > 1
  int buyer_money.pre >= int b.price
  int store_cash.pre >= 0

  //post-condition
  b = buyer_book.post
  store_catalog.pre = store_catalog.post + b
  int (buyer_money.post) = int (buyer_money.pre) - int (b.price)
  int (store_cash.post) = int (store_cash.pre) + int (b.price)
}

```

10.5 Example: the Job Yellow Pages Website

This example has been extracted from [WEGMANN, A. 1998]. The main concept is a job search website, where users introduce information about candidates and about job postings. On one hand, the system presents the most appropriate candidates for a job posting that a company introduces. On the other hand, the system displays to candidates the job postings that better match their profile.

We do not discuss the details of how the system works (the actions or services the system delivers). In this case, our goal is to show how the conceptual modeling of the system can be improved easily by introducing the semantics of the notation of Visual Contracts.

A first draft of the system domain model made with UML gives us the class diagram shown in figure 82.

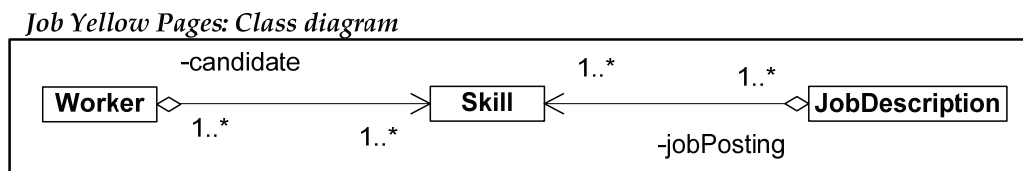


Figure 82. Class diagram for JobYellowPages

If we do a literal translation to the notation of Visual Contracts, we obtain:

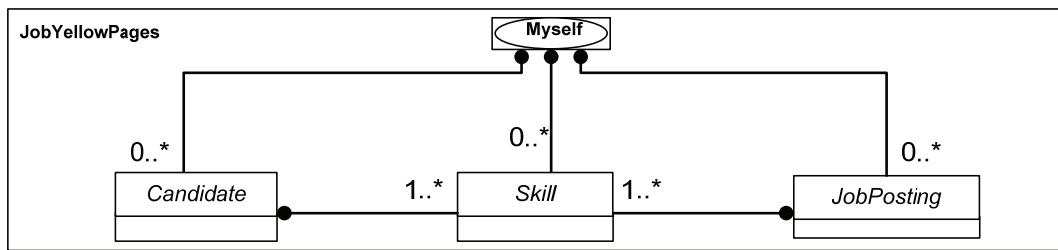


Figure 83. First interpretation of class diagram in terms of Visual Contracts for JobYellowPages

As matching are successful only when the skills possessed by the candidates fits in the skills required by the job posting, we can use an ad hoc representation for the sets, as follows:

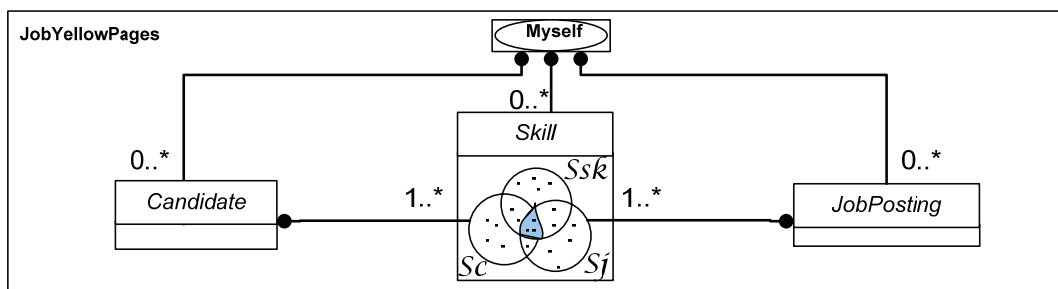


Figure 84. Set-theoretical *ad hoc* interpretation of set-associations of figure 75

Here, the condition that must be satisfied is:

$$S_{sx} \cap S_c \cap S_j \neq \emptyset \quad (14)$$

where:

- S_c : all the skills that the candidate possesses
- S_j : all the skills that the job description requires
- S_{sk} : all the skills that the system might possibly know

However, in this particular case, it is clear that all the skills that the candidate possesses but the system does not know are not useful for doing a job-candidate match; the same can be said about the job description skills. Therefore, it is required that all the skills are known by the system. In other words,

$$S_w \subseteq S_{sx} \wedge S_j \subseteq S_{sx} \quad (15)$$

is true, making equation 14 automatically true. Besides, this equation makes other sets appearing in figure 76 irrelevant. In other words,

$$S_{sx} \cap S_c \cap S_j \equiv S_{sx} \cap S_c \cap S_j \equiv S_{sx} \cap S_c \cap S_j \equiv \emptyset \quad (16)$$

Note that this information cannot be easily extracted from the class diagram in Fig. 74. It could eventually be added via a note or another diagram where this constraint can be added, but it is not immediately available to the modeler, and this may generate mistakes.

Furthermore, in the UML class diagram semantics are rather loose, For instance, we may consider that the skills are erased from the system whenever the candidate or the job description are erased. By making explicit that the owner of the description is neither the candidate nor the job description, we ensure that the system will not arrive to inconsistent states, where referential integrity rules are violated.

We apply here the design heuristics for visual contracts (see Section 7.3), and differentiate the actual instances from the references to those instances. This makes the owner explicit. The final domain model created using Visual Contract notation is shown in figure 85.

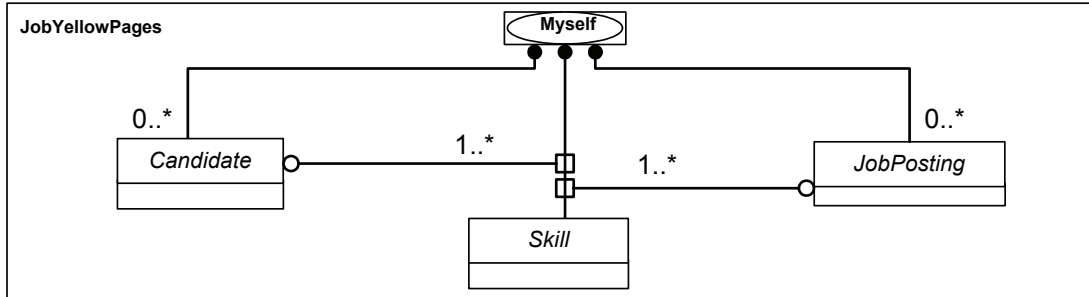


Figure 85. Correct modeling of structural model of JobYellowPages using the Visual Contracts notation

Now the modeler should be able to proceed to the specification of the actions.

10.6 Summary

In this chapter we demonstrated—in a practical fashion—how Visual Contracts are a tool that allows for specifying systems and validating them, even from the very first drafts. In each case, we were able to express a design and to take advantage of the formal reasoning that the semantics of Visual Contracts provide to modelers.

The diversity of the study cases allowed us to illustrate many aspects of the syntax and semantics in practice. We also used several heuristics and patterns. In addition, in two cases we went through the complete process—from conception to validation and verification—using the strategy presented in chapter 8.

The application of Visual Contracts for modeling systems that range from software systems to organizations and supply chains, illustrate the potential impact of this specification artifact.

With the examples, we demonstrated that the Visual Contract can be used in different domains, for IT and non-IT systems, or just as a reasoning tool to better understand the problem at hand.

PART IV – Closing Thoughts

11 Conclusions

In this thesis we sought to improve the state of the art of modeling system specifications, in particular in IT systems. We that current methods propose diagrammatic and textual specification artifacts, and that the composition of these specification artifacts permits creating models of complex systems. However, the most popular of these methods (i.e. UML) separates the different aspects of the specification (i.e. behavior, structure, state and constraints) into different diagrams, making it difficult for modelers to build a holistic understanding of the system.

We proposed a specification artifact called visual contract that exhibits the following main features:

- Graphical / Visual / Diagrammatic
- Compatible with formal methods
- System-centric

The notation was formalized using Alloy and it enables the verification of system specifications. We see this as a necessary component for MDE.

Visual contracts are partially based on the theories developed by Prof. Wegmann's group and embodied in the Systemic Enterprise Architecture Methodology (SEAM) [WEGMANN, A. 2003]. SEAM is itself based on General Systems Thinking [WEINBERG, G. 2001] and the Living Systems Theory [MILLER, J.G. 1995].

The specific goal of our work is to create system specifications that describe in a visual way the four aspects (i.e., structural, behavioral –including communication and synchronization aspects—, and constraints) on a single diagram. This is a complementary approach to the traditional, analytic one, and we are able to express features that are very complex using notations like UML.

One of the goals we had was to create a comprehensible notation for the specification artifact. By using a minimum of elements, and by creating a way to represent instances and collections of instances in a compact form, we are able to express Visual Contracts for the actions of a system.

Accordingly, by being able to specify how the collections of instances change via the executions of the actions of a system, we are able to model the structural changes that happen in the system. This is a meaningful result, as cardinalities are normally used in specifications only as restrictions in the form of invariants. In this case, we can use cardinalities to show the dynamics of the system: how the system is built, and how it evolves through the time.

This modeling of change is also explicit. The causality is represented via the change and transfer operators. The change operator indicates where the change takes place: the collection (set-association), and the nature of the change—i.e. creation, change of state, deletion—, whereas the transfer operator makes explicit the causality —origin, destination and enabling condition— of that cardinality change.

The notation for Visual Contracts facilitates the task of the modeler when compared to the direct use of formal methods. Its visual notation avoids the burden of building high-level logical constructions to specify systems; the inherent complexity of such construction has been pointed out as the main limitation for a wider adoption of the formal methods. In addition, the possibility to perform verifications of the system specification without requiring a full-fledged model is also a major contribution of the approach.

11.1 Limitations of Visual Contracts

The use of Visual Contracts for rule-based systems, potentially with many branches of behavior is not recommended. The select operator supports the use of complex rules, but the verification is not possible, because Alloy does not support any longer the instantiation as the first version did. This means that it is not possible to model accurately the behavior of such systems. If the rules do not map at the inside of the Visual Contract but at the composition level, this requires special operational semantics what we do not provide in this research work.

The modeling of information objects can deal with potentially highly-complex objects. Nevertheless, the modeling of objects with many attributes is not encouraged, because of the complexity of the resulting models. This is a common issue in diagrammatic approaches, as seen in Chapter 4.

The modeling of numerical values is not directly supported. Should the modeler need numerical values, she has to use the basic integer type. It is the only one currently supported by the model-checking algorithms for Alloy. The use of special types is discouraged as the ordering relationships have to be established, and this will make the specification complex and cumbersome to maintain.

The modeling of pure computation, as discussed in Chapter 6, is more efficient with algorithms. No graphical counterpart can compete against the efficiency of algorithms.

11.2 Future Work

We have explored the initial path to the creation of an alternate representation using a matrix to represent a system. The elements of the matrix would be the cardinalities, and some non-linear operators, analogous to the Karnaugh's maps used for Boolean logic.

The matrix representation would enable a block-oriented representation of the actions of the system. This would permit representing contracts in the form of blocks that are similar to transfer functions. This could eventually lead to a new ways of representing information systems, for example.

Another aspect that can be further developed is the semantics for the composition of behavior. Composition requires the creation of operational semantics that complement our declarative approach.

As Visual Contracts illustrate what is possible, we consider the specification of Anti-contracts might be a useful artifact to build more complete specifications. Anti-contracts are particularly interesting as a complement to VCs for building rule-based systems.

The representation of time is limited to the connection of actions. However, by incorporating model checkers that permit instantiation of the time points, the implementation of a modal logic would be feasible. As explained in the metamodel (Chapter 7), the temporal elements of the notation would allow the introduction of delayed processing and of temporal triggers. As discussed in 11.1, the lack of an instantiation mechanism in Alloy precludes this possibility. Nonetheless, this is an interesting feature to have for many scenarios.

The usability of the notation has not been demonstrated. This requires the application of Visual Contracts to more complex study cases, plus the creation of some tool support, because currently the Visual Contracts are drawn using minimal computer support.

Finally, it could be interesting to integrate more formal aspects such as the nature of the relation itself, in the set-associations (total and partial functions; bijective, injective and surjective functions, etc.), and analyze the usefulness of adding this kinds of constraints.

Bibliographical References

- [IEEE 1993] IEEE Standard Dictionary of Computer Terms. New Jersey: IEEE Press, 1993.
- [AALST, W.M.P.V.D., *et al* 2003] Aalst, W. M. P. v. d., Hofstede, A. H. M. t., Kiepuszewski, B., and Barros, A. P., "Workflow Patterns," *Distributed and Parallel Databases*, vol. 14(1), pp. 5-51, 2003.
- [AGG 2006] AGG, "The Attributed Graph Grammar System: A Development Environment for Attributed Graph Transformation Systems". Accessed in 29 Jun,2007. I. f. S. u. T. Informatik, Ed. Berlin, Germany: Technische Universität Berlin. <http://tfs.cs.tu-berlin.de/agg/>
- [AGRAWAL, A. 2003] Agrawal, A., "Graph Rewriting And Transformation (GReAT): A Solution For The Model Integrated Computing (MIC) Bottleneck," in *18th IEEE International Conference on Automated Software Engineering (ASE 2003)*. Montreal, Canada: IEEE Computer Society, 2003, pp. 364-368.
- [AKKOK, N. 2004] Akkok, N., *Towards the Principles of Designing Diagrammatic Modeling Languages: Some Visual, Cognitive and Foundational Aspects*, Doctoral Dissertation presented to Department of Informatics, Faculty of Mathematics and Natural Sciences. Oslo: University of Oslo, 2004.
- [AL-AHMAD, W. 2001] Al-Ahmad, W., "On the Interaction of Programming by Contract and Liskov Substitution Principle," 2001, pp. 421-423.
- [ARGAWAL, R. and SINHA, A.P. 2003] Argawal, R. and Sinha, A. P., "Object-oriented modeling with UML: a study of developers' perceptions," in *Communications of the ACM (CACM)*, vol. 46, 2003, pp. 248-256.
- [ATKINSON, C., *et al* 2002] Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Paech, B., Wüst, J., and Zettel, J., *Component-based Product Line Engineering with UML*, in The Addison-Wesley Object Technology Series, 1 ed: Addison-Wesley, 2002.
- [AUDI, R. 1999] Audi, R., "Cambridge dictionary of philosophy," Cambridge University Press, 1999.
- [BARDOHL, R., *et al* 2004] Bardohl, R., Ehrig, H., Lara, J. d., and Taentzer, G., "Integrating Meta-modelling Aspects with Graph Transformation for Efficient Visual Language Definition and Model Manipulation," presented at *Fundamental Approaches to Software Engineering, 7th International Conference, FASE 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004*, Barcelona, Spain, 2004, pp. 214-228.
- [BARESI, L. and PEZZÈ, M. 2005] Baresi, L. and Pezzè, M., "Formal interpreters for diagram notations," *ACM Transaction on Software Engineering Methodology*, vol. 14(1), pp. 42-84, 2005.
- [BARWISE, J. and ETCEMENDY, J. 1993] Barwise, J. and Etchemendy, J., *The language of first-order logic : including the Macintosh version of Tarski's world 4.0*, in CSLI Publications, 3rd ed: Center for the Study of Language and Information, 1993.
- [BARWISE, J. and ETCEMENDY, J. 2002] Barwise, J. and Etchemendy, J., "Language, Proof and Logic," Center for the Study of Language and Information, 2002.
- [BATE, I., *et al* 2003] Bate, I., Hawkins, R., and McDermid, J., "A contract-based approach to designing safe systems," in *Proceedings of the 8th Australian workshop on Safety critical systems and software - Volume 33*. Canberra, Australia: Australian Computer Society, Inc., 2003, pp. 25-36.
- [BEIZER, B. 1990] Beizer, B., *Software Testing techniques*. New York: Van Nostrand Reinhold, 1990.
- [BIENVENIDO, J.F. and FLORES-PARRA, I.M. 2004] Bienvenido, J. F. and Flores-Parra, I. M., "Automatic Generation of the Behavior Definition of Distributed Design Tools from Task Method Diagrams and Method Flux Diagrams by Diagram Composition," presented at *Diagrammatic Representation and Inference, Third International Conference, Diagrams 2004*, Cambridge, UK, 2004, pp. 435-437.
- [BLACK, P.E. 1998] Black, P. E., *Axiomatic Semantics Verification of a Secure Web Server*, Doctoral Dissertation presented to Department of Computer Science: Brigham Young University, 1998.
- [BMI-DTF 2007] BMI-DTF, "Business Process Management Initiative". Accessed in June 19,2007. www.bpmi.org
- [BOGACZ, S. and TRAFTON, J.G. 2002] Bogacz, S. and Trafton, J. G., "Understanding Static and Dynamic Visualizations," presented at *Diagrammatic Representation and Inference, Second International Conference, Diagrams 2002*, Callaway Gardens, GA, USA, 2002, pp. 347-349.
- [BOOCH, G., *et al* 1998] Booch, G., Jacobson, I., Rumbaugh, J., and Rumbaugh, J., *The Unified Modeling Language User Guide*: Addison-Wesley Pub Co, 1998.
- [BOSWORTH, R. 2004] Bosworth, R., "Automatic Proofs for Scalecharts," presented at *Diagrammatic Representation and Inference, Third International Conference, Diagrams 2004*, Cambridge, UK, 2004, pp. 227-230.
- [BOTTONI, P., *et al* 2001] Bottoni, P., Koch, M., Parisi-Presicce, F., and Taentzer, G., "A Visualization of OCL Using Collaborations," presented at *UML 2001 - The Unified Modeling Language, Modeling Languages, Concepts, and Tools, 4th International Conference. Lecture Notes in Computer Science 2185*, Toronto, Ontario, Canada, 2001, pp. 257-271.

- [BOWEN, J. 1996] Bowen, J., "The ARIANE 5 Flight 501 Failure Report, annotated version". Accessed in: Inquiry Board, European Space Agency (ESA). <http://www.cafm.sbu.ac.uk/cs/people/jpb/teaching/ethics/ariane5anot.html>
- [BROOKS, F. 1987] Brooks, F., "No Silver Bullet: Essence and Accidents of Software Engineering," *IEEE Computer*, vol. 20(4), pp. 10-19, 1987.
- [BRUEL, J.-M. 1998] Bruel, J.-M., "Integrating Formal and Informal Specification Techniques. Why? How?," presented at *Workshop on Industrial-strenght Formal Techniques*, Boca Raton, Florida, USA, 1998, pp. 50-57.
- [BRUEL, J.-M., et al 2000] Bruel, J.-M., Lilius, J., Moreira, A. M. D., and France, R. B., "Defining Precise Semantics for UML," in *Object-Oriented Technology, ECOOP 2000 Workshops, Panels, and Posters*, Sophia Antipolis and Cannes, France, June 12-16, 2000, Proceedings, vol. 1964, Lecture Notes in Computer Science, J. Malenfant, S. Moisan, and A. M. D. Moreira, Eds.: Springer, 2000, pp. 113-122.
- [BUTCHER, K.R. and KINTSCH, W. 2004] Butcher, K. R. and Kintsch, W., "Learning with Diagrams: Effects on Inferences and the Integration of Information," presented at *Diagrammatic Representation and Inference, Third International Conference, Diagrams 2004*, Cambridge, UK, 2004
- [CATALYSIS 2002] Catalysis, "Catalysis Concept Map". Accessed in June: www.catalysis.org.
- [CHEN, P. 1976] Chen, P., "The entity-relationship model—toward a unified view of data," *ACM Transactions on Database Systems*, vol. 1(1), pp. 9-36, 1976.
- [CHEN, P. 1977] Chen, P., "The Entity-Relationship Model - A basis for the Enterprise View of Data.," presented at *AFIPS National Computer Conference*, 1977, pp. p. 77-84.
- [CHENG, P.C.-H. 2004] Cheng, P. C.-H., "Why Diagrams Are (Sometimes) Six Times Easier than Words: Benefits beyond Locational Indexing," presented at *Diagrammatic Representation and Inference, Third International Conference, Diagrams 2004, Cambridge, UK, March 22-24, 2004, Proceedings*, 2004, pp. 242-260.
- [CHERUBINI, M., et al 2007] Cherubini, M., Venolia, G., DeLine, R., and Ko, A. J., "Let's go to the whiteboard: how and why software developers use drawings," presented at *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*, San Jose, California, USA, 2007, pp. 557-566.
- [COCKBURN, A. 2001] Cockburn, A., *Writing effective use cases*, 1 ed: Addison-Wesley, 2001.
- [COLEMAN, D., et al 1994] Coleman, D., Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H., Hayes, F., and Jeremaes, P., *Object-oriented development: the Fusion method*, 1 ed. Englewood Cliffs: Prentice-Hall, Inc., 1994.
- [COMBINATORICS 2007] Combinatorics, "Combinatorics". Accessed in June 20th, 2007. <http://www.combinatorics.org/Surveys/ds5/VennEJC.html>
- [COOK, S. and DANIELS, J. 1994] Cook, S. and Daniels, J., *Designing Object Systems: Object-Oriented Modelling with Syntropy*: Prentice Hall, 1994.
- [COX, R., et al 2004] Cox, R., Romero, P., Boulay, B. d., and Lutz, R., "A Cognitive Processing Perspective on Student Programmers' "Graphicacy"," presented at *Diagrammatic Representation and Inference, Third International Conference, Diagrams 2004, Cambridge, UK, March 22-24, 2004, Proceedings*, 2004, pp. 344-346.
- [CRNKOVIC, I. 2002] Crnkovic, I., "Building Reliable Component-Based Systems." Edited book, prompt to publication, 2002.
- [DAVID, R. and ALLA, H.H. 1997] David, R. and Alla, H. H., *Du GRAFCET aux réseaux de Petri*, in *Automatique*, 2nd ed. Paris: Hermes, 1997.
- [DE LA CRUZ, J.D., et al 2006a] De la Cruz, J. D., Le, L. S., and Wegmann, A., "Validation of Visual Contracts for Services," presented at *4th International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems, MSVVEIS 2006*, Paphos, Cyprus, 2006a
- [DE LA CRUZ, J.D., et al 2006b] De la Cruz, J. D., Lê, L.-S., and Wegmann, A., "Visual Contracts - A way to reason about states and cardinalities in IT system specifications," presented at *8th International Conference on Enterprise Information Systems - ICEIS 2006*, Paphos, Cyprus, 2006b, pp. 298-303.
- [DE LA CRUZ, J.D., et al 2005] De la Cruz, J. D., Wegmann, A., and Regev, G., "Expressing Systemic Contexts in Visual Models of System Specifications", in *Proc. of 1st Workshop on Context Modeling and Decision Support*. Accessed in October 31, 2005. A. Gachet and R. Sprague, Eds. Paris, France: CEUR Workshop Proceedings. http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-144/04_deLaCruz.pdf
- [DE WIN, B., et al 2002] De Win, B., Piessens, F., and Joosen, W., "On the importance of the separation-of-concerns principle in secure software engineering," presented at *ACSA Workshop on the Application of Engineering Principles to System Security Design (WAEPSSD)*, 2002
- [DIJKSTRA, E.W. 1976] Dijkstra, E. W., *A Discipline of Programming*, 1 ed. Englewood Cliffs: Prentice Hall, 1976.

- [DINH-TRONG, T.T., *et al* 2006] Dinh-Trong, T. T., Ghosh, S., and France, R. B., "A Systematic Approach to Generate Inputs to Test UML Design Models," presented at *17th International Symposium on Software Reliability Engineering (ISSRE 2006)*, Raleigh, North Carolina, USA, 2006, pp. 95-104.
- [DISKIN, Z. 1995] Diskin, Z., "Formalizing Graphical Schemas for Conceptual Modeling: Sketch-based Logic vs. Heuristic Pictures," Laboratory for Database Design, University of Latvia, Riga, Latvia, Technical Report FIS/LDBD-95-03, July 1995 1995.
- [DISKIN, Z. and KADISH, B. 1998] Diskin, Z. and Kadish, B., "The Arrow Manifesto: Towards software engineering based on comprehensible yet rigorous graphical specifications," Laboratory for Database Design, University of Latvia, Riga, Latvia 1998.
- [DOBING, B. and PARSONS, J. 2006] Dobing, B. and Parsons, J., "How UML is used," *Communications of the ACM*, vol. 49(5), pp. 109-113, 2006.
- [DORI, D. 2002a] Dori, D., *Object-Process Methodology: A Holistic Systems Paradigm*, 1 ed: Springer Verlag, 2002a.
- [DORI, D. 2002b] Dori, D., "Why significant UML change is unlikely," in *Communications of the ACM (CACM)*, vol. 45, 2002b, pp. 82-85.
- [D'SOUZA, D.F. and CAMERON WILLS, A. 1998] D'Souza, D. F. and Cameron Wills, A., *Objects, components, and frameworks with UML: The Catalysis approach*, 1 ed: Addison Wesley Longman, inc., 1998.
- [ECMA-INTERNATIONAL 2006] ECMA-International, "Standard ECMA-367 —Eiffel: Analysis, Design and Programming Language". Accessed in June,2006, 2nd ed: ECMA International. <http://www.ecma-international.org/publications/standards/Ecma-367.htm>
- [ELKAN, C. 1991] Elkan, C., "Reasoning about action in first-order logic," presented at *Conference of the Canadian Society for Computational Studies of Intelligence (CSCSI)*, 1991, pp. 221-227.
- [ERIKSSON, H.-E. and PENKER, M. 2000] Eriksson, H.-E. and Penker, M., *Business modeling with UML Business Patterns at work*: Wiley, 2000.
- [EVANS, A., *et al* 1998] Evans, A., France, R. B., Lano, K., and Rumpe, B., "The UML as a Formal Modeling Notation," in *The Unified Modeling Language, «UML»'98: Beyond the Notation*, First International Workshop, Mulhouse, France, June 3-4, 1998, Selected Papers, vol. 1618, Lecture Notes in Computer Science, J. Bézivin and P.-A. Muller, Eds. Mulhouse, France: Springer, 1998, pp. 336-348.
- [FIRESMITH, D.G. 1996] Firesmith, D. G., "Pattern Language for Testing Object-Oriented Software," in *Object Magazine*, vol. 5, 1996, pp. 32-38.
- [FISH, A., *et al* 2005] Fish, A., Flower, J., and Howse, J., "The semantics of augmented constraint diagrams," *Journal of Visual Language Computing*, vol. 16(6), pp. 541-573, 2005.
- [FLOYD, R.W. 1967] Floyd, R. W., "Assigning meanings to programs," in *Proceedings of Symposia in Applied Mathematics*, vol. 19, *Mathematical Aspects of Computer Science*, 1967, pp. 19-32.
- [FRAPPIER, M., *et al* 2002] Frappier, M., Fraikin, B., Laleau, R., and Richard, M., "Automatic Production of Information Systems," American Association for Artificial Intelligence, Menlo Park, California., Stanford, CA, Technical Report SS-02-05. 2002.
- [GERVAIS, F. 2004] F. Gervais, "[EB4 : Vers une méthode combinée de spécification formelle des systèmes d'information](#)" Examen de spécialité, Doctorat Informatique, Université de Sherbrooke (Québec), Canada, 2004
- [GIL, J., *et al* 2001] Gil, J., Howse, J., and Kent, S., "Towards a Formalization of Constraint Diagrams," presented at *2002 IEEE CS International Symposium on Human-Centric Computing Languages and Environments (HCC 2001)*, Stresa, Italy, 2001
- [GIL, J. and KENT, S. 1998] Gil, J. and Kent, S., "Three Dimensional Software Modeling," presented at *International Conference on Software Engineering, ICSE 98*, Kyoto, Japan, 1998
- [GIL, J.Y., *et al* 2000] Gil, J. Y., Howse, J., Taylor, J., and Kent, S., "Projections in Venn-Euler Diagrams," in *Proceedings of the 2000 IEEE International Symposium on Visual Languages (VL'00)*: IEEE Computer Society, 2000, pp. 119.
- [GIRARD, J.-Y. 1987] Girard, J.-Y., *Proof Theory and Logical Complexity*. Napoli: Bibliopolis, 1987.
- [GLASER, H., *et al* 2001] Glaser, H., Hartel, P. H., Leuschel, M., and Martin, A., "Declarative Languages in Education," in *Encyclopedia of Microcomputers*, vol. 27, A. Kent and J. G. Williams, Eds. New York: Marcel Dekker Inc, 2001, pp. 79-102.
- [GLASGOW, J., *et al* 1995] Glasgow, J., Narayanan, N. H., and Chandrasekaran, B., "Diagrammatic Reasoning: Cognitive and Computational Perspectives," AAAI Press, 1995, pp. 807.
- [GOSLING, J., *et al* 2005] Gosling, J., Joy, B., Steele, G., and Bracha, G., *The Java Language Specification*, in Java, 3rd ed: Addison-Wesley Professional, 2005.
- [GRIES, D. 1981] Gries, D., *The science of programming*, in Text and monographs in computer science, 1 ed. New York: Springer-Verlag, 1981.
- [GRIES, D. and LEVIN, G. 1980] Gries, D. and Levin, G., "Assignment and procedure call proof rules," presented at *TOPLAS 2*, 1980, pp. 564-579.
- [GRUNDY, J. 1993] Grundy, J., *A Method of Program Refinement*, Doctoral Dissertation presented to Fitzwilliam College: University of Cambridge, 1993.

- [GUTTAG, J.V. and HORNING, J.J. 1978] Guttag, J. V. and Horning, J. J., "The Algebraic Specification of Abstract Data Types," *ACTA Informatica*, vol. 10, pp. 27-52, 1978.
- [HAMMER, E. 1995] Hammer, E., "Logic and Visual Information". Studies in Logic, Language, and Computation. Stanford: CSLI Publications and FoLLI, 1995..
- [HAREL, D. 1987] Harel, D., "Statecharts: A Visual Formulation for Complex Systems," *Science of Computer Programming*, vol. 8(3), pp. 231-274, 1987.
- [HAREL, D. 1992] Harel, D., "Biting the Silver Bullet - Toward a Brighter Future for System Development," *IEEE Computer*, vol. 25(1), pp. 8-20, 1992.
- [HATLEY, D. and IMTIAZ, P. 1988] Hatley, D. and Imtiaz, P., *Strategies for Real-Time System Specification*. New York: Dorset House, 1988.
- [HECKEL, R. and SAUER, S. 2001] Heckel, R. and Sauer, S., "Strengthening UML Collaboration Diagrams by State Transformations," presented at *Proc. Fundamental Approaches to Software Engineering (FASE'2001)*, Genova, Italy, 2001, pp. pp. 109-12.
- [HELM, R., et al 1990] Helm, R., Holland, I. M., and Gangopadhyay, D., "Contracts: Specifying Behavioural Compositions in Object-Oriented Systems.," presented at *OOPSLA/ECOOP 1990*, Ottawa, Canada, 1990, pp. pp. 169-180.
- [HOARE, C.A.R. 1969] Hoare, C. A. R., "An Axiomatic Basis for Computer Programming," *Communications of the ACM (CACM)*, vol. 12(10), pp. 576-580, 1969.
- [HOARE, C.A.R. 1972] Hoare, C. A. R., "Proof of correctness of data representations," in *Acta Informatica*, vol. 1, 1972, pp. 271-283.
- [HOARE, C.A.R. 1985] Hoare, C. A. R., *Communicating Sequential Processes*, in Computer Science Series: Prentice-Hall International, 1985.
- [HOARE, C.A.R. and WIRTH, N. 1973] Hoare, C. A. R. and Wirth, N., "An Axiomatic Definition of the Programming Language PASCAL," *ACTA Informatica*, vol. 2, pp. 335-355, 1973.
- [HODGES, W. 1993] Hodges, W., *Model theory*, in Encyclopedia of mathematics and its applications vol. 42. Cambridge: Cambridge University Press, 1993.
- [HOWSE, J., et al 2001] Howse, J., Molina, F., Taylor, J., Kent, S., and Gil, J., "Spider Diagrams: A Diagrammatic Reasoning System," *Journal of Visual Languages and Computing*, vol. 12(3), pp. 299-324, 2001.
- [IGARASHI, S., et al 1975] Igarashi, S., London, R. L., and Luckham, D. C., "Automatic program verification: a logical basis and its implementation," in *Acta Informatica*, vol. 1, 1975, pp. 145-182.
- [ILLINGWORTH, V. and (ED.) 1996] Illingworth, V. and (Ed.), *Dictionary of computing*. Oxford: Oxford University Press, 1996.
- [INSTITUTE FOR LOGIC, C.A.D.S.-U.O.K. and GOTHENBURG, C.U.-. 2001] Institute for Logic, C. a. D. S.-U. o. K. and Gothenburg, C. U.-. "KeY: Authoring Tool for OCL Constraints", 0.7 ed. <http://i12www.ira.uka.de/~projekt/specifications.htm>
- [IRANI, P. 2004] Irani, P., "Notations for Software Engineering Class Structures," presented at *Diagrammatic Representation and Inference, Third International Conference, Diagrams 2004*, Cambridge, UK, 2004, pp. 441-445.
- [ISE 2001] ISE, "Building bug-free O-O software: An introduction to Design by Contract (TM)". Accessed in: <http://www.eifel.com/doc/manuals/technology/contract>
- [ISO/IEC 1995-1996] ISO/IEC, *10746-1, 2, 3, 4 | ITU-T Recommendation X.901, X.902, X.903, X.904. Open Distributed Processing - Reference Model.*: http://isotc.iso.ch/livelink/livelink/fetch/2000/2489/Ittf_Home/PubliclyAvailableStandards.htm, 1995-1996.
- [ISO/IEC 1996] ISO/IEC, *ISO/IEC 10746-1: Overview*, in Information Technology - Open Distributed Processing - Reference Model. Geneva: ISO/IEC, 1996.
- [ISO/IEC and ITU-T 1998] ISO/IEC and ITU-T, "Recommendation X.901, X.902, X.903, X.904, "Open Distributed Processing - Reference Model", ISO and ITU-T, Recommendation 1995-98 1998.
- [JACKSON, D. 2002] Jackson, D., "Alloy: a lightweight object modelling notation," *ACM Trans. Softw. Eng. Methodol.*, vol. 11(2), pp. 256-290, 2002.
- [JACKSON, D. 2005] Jackson, D., "Alloy Reference Manual". Accessed in October, 2006: MIT Software Design Group. alloy.mit.edu/reference-manual.pdf
- [JACKSON, D. and RINARD, M. 2000] Jackson, D. and Rinard, M., "The Future of Software Analysis," in *The Future of Software Engineering*, A. Finkelstein, Ed.: ACM Press, 2000.
- [JARRATT, T., et al 2004] Jarratt, T., Keller, R., Nair, S., Eckert, C., and Clarkson, P. J., "Visualization Techniques for Product Change and Product Modelling in Complex Design," presented at *Diagrammatic Representation and Inference, Third International Conference, Diagrams 2004*, Cambridge, UK, 2004, pp. 388-391.
- [JÉZÉQUEL, J.-M. and MEYER, B. 1997] Jézéquel, J.-M. and Meyer, B., "Design by Contract: The lessons of Ariane," in *IEEE Computer*, vol. 30, 1997, pp. 129-130.

- [JÉZÉQUEL, J.-M., *et al* 2000] Jézéquel, J.-M., Train, M., and Mingins, C., *Design patterns and contracts*, 1 ed: Addison Wesley Longman, 2000.
- [JIN, Y., *et al* 2004] Jin, Y., Esser, R., and Janneck, J. W., "A method for describing the syntax and semantics of UML statecharts," *Software and System Modeling*, vol. 3(2), pp. 150-163, 2004.
- [JONES, C.B. 1990] Jones, C. B., *Systematic Software development using VDM*, 2 ed. Englewood Cliffs, NJ: Prentice Hall International, 1990.
- [KENT, S. and GIL, J. 1998] Kent, S. and Gil, J., "Visualising action contracts in object-oriented modelling," *IEEE Proceedings - Software*, vol. 145(2-3, April-June), pp. 70-78, 1998.
- [KHAN, K., *et al* 2000] Khan, K., Han, J., and Zheng, Y., "Security Characterisation of Software Components and Their Composition," 2000, pp. 240-249.
- [KOCH, S. 1999] Koch, S., "Using Weakes Precondition for Software Process Model Reuse," presented at *Fifth Americas Conference on Information Systems (AMCIS 1999)*, Milwaukee, WI, 1999, pp. 741-743.
- [KOWALSKI, R.A. and SERGOT, M.J. 1986] Kowalski, R. A. and Sergot, M. J., "A Logic-based Calculus of Events," *New Generation Computing*, vol. 4(1), pp. 67-95, 1986.
- [KRUCHTEN, P. 2000] Kruchten, P., *The rational unified process: An introduction*, in Object Technology Series, 2 ed: Addison-Wesley, 2000.
- [LAMPOR, L. 1984] Lamport, L., "An Axiomatic Semantics of Concurrent Programming Languages," presented at *Logics and Models of Concurrent Systems*, Colle-sur-Loup, France, 1984, pp. 77-122.
- [LAMPOR, L. and SCHNEIDER, F. 1989] Lamport, L. and Schneider, F., "Pretending Atomicity," SRC Research, Research Report May 1989.
- [LAMSWEERDE, A.V. and LETIER, E. 2002] Lamsweerde, A. v. and Letier, E., "From Object Orientation to Goal Orientation: A Paradigm Shift for Requirements Engineering," presented at *Radical Innovations of Software and Systems Engineering in the Future, 9th International Workshop, RISSEF 2002, Venice, Italy, October 7-11, 2002, Revised Papers*, 2002, pp. 325-340.
- [LANO, K. and EVANS, A. 1999] Lano, K. and Evans, A., "Rigorous Development in UML," presented at *FASE 1999 (Part of ETAPS 1999)*, Amsterdam, The Netherlands, 1999, pp. 129-144.
- [LAPLANTE, P.A. and (ED.) 2001] Laplante, P. A. and (ed.), *Dictionary of computer science, engineering and technology*: CRC Press, 2001.
- [LARA, J.D. and VANGHELUWE, H. 2002] Lara, J. d. and Vangheluwe, H., "AToM3: A Tool for Multi-formalism and Meta-modelling," presented at *Fundamental Approaches to Software Engineering, 5th International Conference, FASE 2002, held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002*, Grenoble, France, 2002, pp. 174-188.
- [LARKIN, J. and SIMON, H. 1987] Larkin, J. and Simon, H., "Why a diagram is (sometimes) worth ten thousand words," *Cognitive Science*, vol. 11, pp. 65-99, 1987.
- [LARMAN, C. 1997] Larman, C., *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*: Prentice Hall, 1997.
- [LE, L.S. and WEGMANN, A. 2005] Le, L. S. and Wegmann, A., "Definition of an Object-Oriented Modeling Language for Enterprise Architecture," presented at *Hawaii International Conference on System Sciences (HICSS'05)*, Hawaii, USA, 2005
- [LE MOIGNE, J.-L. 1993] Le Moigne, J.-L., *Modelisation des systemes complexes (La)*: Dunod, 1993.
- [LENAT, D.B. and GUHA, R.V. 1990] Lenat, D. B. and Guha, R. V., *Building Large Knowledge-Based Systems: Representation and Inference in the CYC Project*. Reading, Massachusetts: Addison-Wesley, 1990.
- [LIEBERMAN, H. 1986] Lieberman, H., "Using prototypical objects to implement shared behavior in object-oriented systems" in *Conference proceedings on Object-oriented programming systems, languages and applications* Portland, Oregon, United States ACM Press, 1986 pp. 214-223
- [LISKOV, B. and WING, J.M. 1993] Liskov, B. and Wing, J. M., "A New Definition of the Subtype Relation," in *Proceedings ECOOP '93 - 7th European Conference on Object-Oriented Programming*, vol. 707, LCNS. Kaiserslautern, Germany: Springer Berlin / Heidelberg, 1993.
- [LISKOV, B. and WING, J.M. 1994] Liskov, B. and Wing, J. M., "A Behavioral Notion of Subtyping," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16(6), pp. 1811-1841, 1994.
- [LISKOV, B. and ZILLES, S. 1974] Liskov, B. and Zilles, S., "Programming with abstract data types," presented at *Proc. ACM SIGPLAN Conf. on Very High Level Languages*, 1974, pp. 50-60.
- [LOHMANN, M., *et al* 2005] Lohmann, M., Sauer, S., and Engels, G., "Executable Visual Contracts," presented at *IEEE VL/HCC'05*, Dallas, Texas, USA, 2005
- [MARCA, D.A. and MCGOWAN, C.L. 1988] Marca, D. A. and McGowan, C. L., *SADT : structured analysis and design technique*. New York: McGraw-Hill, 1988.
- [MELLOR, S.J., *et al* 2003] Mellor, S. J., Clark, A. N., and Futagami, T., "Guest Editors' Introduction: Model-Driven Development," in *IEEE Software*, vol. 20, 2003, pp. 14-18.
- [MELLOR, S.J., *et al* 1999] Mellor, S. J., Tockey, S. R., Arthaud, R., and Leblanc, P., "An Action Language for UML: Proposal for a Precise Execution Semantics," in *Selected papers from the First International Workshop on The Unified Modeling Language UML'98: Beyond the Notation*: Springer-Verlag, 1999, pp. 307-318.

- [MERRIAM-WEBSTER 2005] Merriam-Webster, "Merriam-Webster OnLine". Accessed in April 2005. www.m-w.com. Ed.: Merriam-Webster, Inc.
- [MEYER, B. 1988] Meyer, B., *Object-Oriented Software Construction*, in Computer Science: Prentice Hall International, 1988.
- [MEYER, B. 1992] Meyer, B., "Applying "Design by Contract", in *IEEE Computer*, vol. 25, 1992, pp. 40-51.
- [MEYER, B. 1997] Meyer, B., *Object-Oriented software construction*, 2 ed: Prentice-Hall, 1997.
- [MEYER, B. 2001] Meyer, B., "Product or Service?," in *Software Development Magazine*: www.sdmagazine.com, 2001.
- [MILLER, G.A. 1956] Miller, G. A., "The Magical Number Seven, Plus or Minus Two: Some Limits on our Capacity for Processing Information," *Psychological Review*, vol. 63, pp. 81-97, 1956.
- [MILLER, J.G. 1995] Miller, J. G., *Living Systems*, 2 ed: University Press of Colorado, 1995.
- [MILNER, R. 1980] Milner, R., *Calculus of Communicating Systems*, in LNCS vol. 92: Springer-Verlag, 1980.
- [MILNER, R. 1989] Milner, R., *Communication and Concurrency*, 1 ed: Prentice Hall, 1989.
- [MILNER, R. 1999] Milner, R., *Communicating and mobile systems: The Pi-calculus*, 1 ed. Cambridge: Cambridge University Press, 1999.
- [MITCHELL, R. and MCKIM, J.] Mitchell, R. and McKim, J., "Extending a method for devising software contracts"
- [MITCHELL, R. and MCKIM, J. 2002] Mitchell, R. and McKim, J., *Design by Contract, by example*, in Object-oriented programming (Computer science), 1 ed: Addison-Wesley, 2002.
- [MONIN, J.-F. 2000] Monin, J.-F., *Introduction aux Méthodes Formelles*, 2nd ed. Paris: Hermes Science, 2000.
- [NARAYANAN, N.H. and HEGARTY, M. 2000] Narayanan, N. H. and Hegarty, M., "Communicating Dynamic Behaviors: Are Interactive Multimedia Presentations Better than Static Mixed-Mode Presentations?," presented at *Theory and Application of Diagrams, First International Conference, Diagrams 2000*, Edinburgh, Scotland, 2000, pp. 178-193.
- [NAUMENKO, A. 2002] Naumenko, A., *Triune Continuum Paradigm: a paradigm for General System Modeling and its applications for UML and RM-ODP*, Doctoral Dissertation presented to School of Computer Science and Communications Systems. Lausanne: EPFL, 2002.
- [NELSON, T., et al 2000] Nelson, T., Cowan, D., and Alencar, P., "A Model for Describing Object-Oriented Systems from Multiple Perspectives," in *FASE 2000*, vol. 1783, *Lecture Notes in Computer Science*, T. S. E. Maibaum, Ed. Berlin, Germany: Springer, 2000, pp. p. 237-248.
- [OMG 1997] OMG, "CORBA/IIOP Specifications". Accessed in October, 2006. http://www.omg.org/technology/documents/corba_spec_catalog.htm
- [OMG 2003] OMG, "Unified Modeling Language: Superstructure 2.0 Final adopted specification, ptc/03-08-02". Accessed in 2004. <http://www.omg.org/docs/ptc/03-08-02.pdf>
- [OMG 2004] OMG, "UML Profile for Relationships, v1.0". Accessed in 10.10, 2005. <http://www.omg.org/cgi-bin/doc?formal/2004-02-07>
- [OMG 2005a] OMG, "Business Process Modeling Notation (BPMN) Information". Accessed in 10.10, 2005. <http://www.bpmn.org/>
- [OMG 2005b] OMG, "SysML Specification v. 0.9 Draft, <http://www.sysml.org/artifacts.htm>". Accessed in <http://www.sysml.org/artifacts.htm>
- [OMG 2005c] OMG, "Unified Modeling Language (UML)". Accessed in: www.omg.org, www.omg.org
- [OMG 2006] OMG, "Modeling and Metadata Specifications". Accessed in October, 2006. http://www.omg.org/technology/documents/modeling_spec_catalog.htm
- [OMMERING, R.C.V., et al 2001] Ommering, R. C. v., Krikhaar, R. L., and Feijs, L. M. G., "Languages for formalizing, visualizing and verifying software architectures," *Computer Languages*, vol. 27(1/3), pp. 3-18, 2001.
- [ORMSC, O.-A.B. 2001] ORMSC, O.-A. B., "Model Driven Architecture (MDA)," Object Management Group, Document number orms/2001-07-01 July 9 2001.
- [PARNAS, D.L. 1972] Parnas, D. L., "On the Criteria To Be Used in Decomposing Systems into Modules," *Communications of the ACM*, vol. 15(12), pp. 1053-1058, 1972.
- [PARNAS, D.L. and LAWTON, A. 1998] Parnas, D. L. and Lawton, A., "Precisely Annotated Hierarchical Pictures of Programs," McMaster University, Hamilton, Ontario, Canada, Technical Report March 18, 1998 1998.
- [PETERSON, J. 1981] Peterson, J., *Petri Net Theory and the Modeling of Systems*: Prentice Hall, 1981.
- [PEZZÈ, M. and BARESI, L. 2000] Pezzè, M. and Baresi, L., "Can Graph Grammars make Formal Methods more Human?," presented at Workshop on Graph Transformation and Visual Modeling Techniques, co-located with ICALP 2000, and published in ICALP Workshops 2000, Geneva (Switzerland), 2000, pp. 387-394.
- [PREISS, O. 2004] Preiss, O., *Foundations of systems and properties*, Doctoral Dissertation presented to School of Computer Science and Communications Systems. Lausanne, Switzerland: EPFL, 2004.
- [PRICE, S. 2004] Price, S., "Processing Animation: Integrating Information from Animated Diagrams," presented at *Diagrammatic Representation and Inference, Third International Conference, Diagrams 2004*, Cambridge, UK, 2004, pp. 360-364.

- [P UML 2002] pUML, "The precise UML group". Accessed in. <http://www.cs.york.ac.uk/puml/index.html>
- [QUINE, W.V.O. 1937] Quine, W. V. O., "Logic Based on Inclusion and Abstraction," *Journal of Symbolic Logic*, vol. 2(4), pp. 145-152, 1937.
- [REGEV, G. and WEGMANN, A. 2003] Regev, G. and Wegmann, A., *Defining Early IT System Requirements with Regulation Principles: The Lightswitch Approach*, Doctoral Dissertation presented to School of Computer Science and Communications Systems. Lausanne: EPFL, 2003.
- [REGEV, G. and WEGMANN, A. 2005] Regev, G. and Wegmann, A., "Where do Goals Come from: the Underlying Principles of Goal-Oriented Requirements Engineering," presented at *RE '05: Proceedings of the 13th IEEE International Conference on Requirements Engineering (RE'05)*, Washington, DC, USA, 2005, pp. 253-362.
- [REITER, R. 1991] Reiter, R., "The frame problem in the situation calculus: a simple solution (sometimes) and a completeness result for goal regression," in *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, V. I. Lifschitz, Ed.: Academic Press, 1991, pp. 359-380.
- [RICHTERS, M. and GOGOLLA, M. 1998] Richters, M. and Gogolla, M., "On Formalizing the UML Object Constraint Language OCL," presented at *Conceptual Modeling - ER '98, 17th International Conference on Conceptual Modeling*, Singapore, 1998, pp. 449-464.
- [RIK ESHUIS, R.W. 2001] Rik Eshuis, R. W., "A Real-Time Execution Semantics for UML Activity Diagrams," presented at *FASE 2001*, Genova, Italy, 2001, pp. pp. 76-90.
- [ROYER, J.-C. 2004] Royer, J.-C., "Checking Class Schema Usefulness," *Journal of Object Technology*, vol. 3(1), pp. 157-176, 2004.
- [RUMBAUGH, J.R., et al 1991] Rumbaugh, J. R., Blaha, M. R., Lorensen, W., Eddy, F., and Premerlani, W., *Object-Oriented Modeling and Design*: Prentice Hall, 1991.
- [RUSSELL, S. and NORVIG, P. 1995] Russell, S. and Norvig, P., *Artificial Intelligence: a Modern Approach*, in Artificial Intelligence. Englewood Cliffs, New Jersey, USA: Prentice Hall, 1995.
- [SCHÄTZ, B., et al 2002] Schätz, B., Pretschner, A., Huber, F., and Philipps, J., "Model-based development of embedded systems," in *Advances in Object-Oriented Information Systems, OOIS 2002 Workshops*, vol. LNCS 2426, J.-M. Bruel and Z. Bellahsene, Eds. Montpellier, France: Springer, 2002, pp. 298-312.
- [SCHUBERT, L.K. 1976] Schubert, L. K., "Extending the Expressive Power of Semantic Networks," *Artificial Intelligence*, vol. 7(2), pp. 163-198, 1976.
- [SCOTT, K. 2002] Scott, K., *The unified process explained*, 1 ed: Addison-Wesley, 2002.
- [SELIC, B. 2003] Selic, B., "The Pragmatics of Model-Driven Development," in *IEEE Software*, vol. 20, 2003, pp. 19-25.
- [SENDALL, S. 2002] Sendall, S., *Specifying Reactive System Behavior*, Doctoral Dissertation presented. Ecublens: Faculté Informatique et Communications. Ecole Polytechnique Fédérale de Lausanne, 2002.
- [SENDALL, S. and STROHMEIER, A. 2002] Sendall, S. and Strohmeier, A., "Using OCL and UML to Specify System Behavior," in *Object Modeling with the OCL, The Rationale behind the Object Constraint Language*, vol. 2263, *Lecture Notes in Computer Science*, T. Clark and J. Warmer, Eds.: Springer, 2002, pp. 250-280.
- [SETHI, R. 1996] Sethi, R., *Programming languages: concepts and constructs*, 1 ed: Addison-Wesley, 1996.
- [SHIMOJIMA, A. 2002] Shimojima, A., "The Inferential-Expressive Trade-Off: A Case Study of Tabular Representations," presented at *Diagrammatic Representation and Inference, Second International Conference, Diagrams 2002, Callaway Gardens, GA, USA, April 18-20, 2002, Proceedings*, 2002, pp. 116-130.
- [SHIMOJIMA, A. 2004] Shimojima, A., "Inferential and Expressive Capacities of Graphical Representations: Survey and Some Generalizations," presented at *Diagrammatic Representation and Inference, Third International Conference, Diagrams 2004, Cambridge, UK, March 22-24, 2004, Proceedings*, 2004, pp. 18-21.
- [SHIN, S.-J. 1995] Shin, S.-J., *The Logical Status of Diagrams*: Cambridge University Press, 1995.
- [SOWA, J.F. 1999] Sowa, J. F., *Knowledge Representation: Logical, Philosophical, and Computational Foundations*, 1 ed: Brooks/Cole Pub Co, 1999.
- [SPIVEY, M. 1990] Spivey, M., *The Z Notation: A Reference Manual*. Englewood Cliffs, NJ: Prentice Hall International, 1990.
- [SPADE, P.V. 2006] Spade, Paul Vincent, "William of Ockham", *The Stanford Encyclopedia of Philosophy* (Fall 2006 Edition), Edward N. Zalta (ed.), URL = <<http://plato.stanford.edu/archives/fall2006/entries/ockham/>>
- [STAPLETON, G., et al 2004] Stapleton, G., Howse, J., Taylor, J., and Thompson, S., "What Can Spider Diagrams Say?," presented at *Diagrammatic Representation and Inference, Third International Conference, Diagrams 2004*, Cambridge, UK, 2004, pp. 112-127.
- [STEVENS, P. 2001] Stevens, P., "On Use Cases and Their Relationships in the Unified Modelling Language," presented at *FASE 2001*, Genova, Italy, 2001, pp. pp. 140-155.
- [SWOBODA, N. and ALLWEIN, G. 2002] Swoboda, N. and Allwein, G., "Modeling Heterogeneous Systems," presented at *Diagrammatic Representation and Inference, Second International Conference, Diagrams 2002*, Callaway Gardens, GA, USA, 2002, pp. 131-145.

- [SZYPERSKI, C. 2002] Szyperski, C., "Services Rendered", in *Software Development Magazine*. Accessed in January: www.sdmagazine.com. www.sdmagazine.com/print/documentID=20209
- [TAENTZER, G. 1999] Taentzer, G., "AGG: A Tool Environment for Algebraic Graph Transformation," presented at *Applications of Graph Transformations with Industrial Relevance, International Workshop, AGTIVE'99*, Kerkrade, The Netherlands, 1999, pp. 481-488.
- [THOMAS, L., et al 2004] Thomas, L., Ratcliffe, M., and Thomasson, B. J., "Can Object (Instance) Diagrams Help First Year Students Understand Program Behaviour?," presented at *Diagrammatic Representation and Inference, Third International Conference, Diagrams 2004*, Cambridge, UK, 2004, pp. 368-371.
- [TUCKER, S., et al 1997] Tucker, S., Taft, R., and Duff, A., *Ada 95 Reference Manual: Language and Standard Libraries*, in LNCS vol. 1246: Springer, 1997.
- [TUFTTE, E.R. 1997] Tufte, E. R., *Visual Explanations: Images and Quantities, Evidence and Narrative: Graphics Press*, 1997.
- [UNION, I.T. 1992] Union, I. T., "ITU-T Recommendation Z.100 (08/2002) Specification and Description Language (SDL)". Accessed in October, 2006: International Telecommunication Union. <http://www.itu.int/ITU-T/studygroups/com17/languages/>
- [WARD, P. 1985] Ward, P., *Structured Development for Real-Time Systems*. Englewood Cliffs: Yourdon Press, 1985.
- [WARE, C. 2004] Ware, C., *Information Visualization: Perception for Design*, in *Interactive Technologies: Morgan Kaufmann*, 2004.
- [WARMER, J. and KLEPPE, A. 1999] Warmer, J. and Kleppe, A., *The Object Constraint Language, Precise Modeling with UML*: Addison-Wesley, 1999.
- [WARMER, J.B. and KLEPPE, A.G. 1998] Warmer, J. B. and Kleppe, A. G., *The Object Constraint Language: Precise Modeling With Uml*, in *Object Technology Series: Addison-Wesley Professional*, 1998.
- [WEGMANN, A. 2001] Wegmann, A., *xC Method (version 1.0 - January 2001)*. Lausanne, Suisse: ICA-EPFL, 2001.
- [WEGMANN, A. 2003] Wegmann, A., "On Systemic Enterprise Architecture Methodology (SEAM)," presented at *5th International Conference on Enterprise Information Systems, ICEIS 2003*, Angers, France, 2003, pp. 483-490.
- [WEGMANN, A., et al 2005] Wegmann, A., Balabko, P., Le, L.-S., Regev, G., and Rychkova, I., "A Method and Tool for Business-IT Alignment in Enterprise Architecture," presented at *CAiSE'05*, Porto, Portugal, 2005
- [WEGMANN, A. and GENILLOU, G. 2000] Wegmann, A. and Genilloud, G., "The Role of "Roles" in Use Case Diagram," presented at *3rd International Conference on the Unified Modeling Language (UML2000)*, York, UK, 2000, pp. 210-224.
- [WEGMANN, A. and NAUMENKO, A. 2001] Wegmann, A. and Naumenko, A., "Conceptual Modeling of Complex Systems Using an RM-ODP Based Ontology," presented at *5th IEEE International Enterprise Distributed Object Computing Conference - EDOC 2001*, Seattle, USA, 2001, pp. 200-211.
- [WEINBERG, G. 1975] Weinberg, G., *An introduction to General Systems Thinking*. New York: Wiley & Sons, 1975.
- [WEINBERG, G. 2001] Weinberg, G., *An Introduction to General Systems Thinking: Silver Anniversary Edition*: Dorset House Publishing, 2001.
- [WEINBERG, G. and WEINBERG, D. 2001] Weinberg, G. and Weinberg, D., *General Principles of Systems Design*: Dorset House Publishing, 2001.
- [WFMC 2005] WfMC, "WfMC Documents and Interfaces". Accessed in 10.10.2005. <http://www.wfmc.org/standards/standards.htm>
- [WINTERSTEIN, D., et al 2002] Winterstein, D., Bundy, A., Gurr, C. A., and Jamnik, M., "Using Animation in Diagrammatic Theorem Proving," presented at *Diagrammatic Representation and Inference, Second International Conference, Diagrams 2002*, Callaway Gardens, GA, USA, 2002, pp. 46-60.
- [WINTERSTEIN, D., et al 2004] Winterstein, D., Bundy, A., Gurr, C. A., and Jamnik, M., "An Experimental Comparison of Diagrammatic and Algebraic Logics," presented at *Diagrammatic Representation and Inference, Third International Conference, Diagrams 2004*, Cambridge, UK, 2004, pp. 432-434.
- [WIRFS-BROCK, R., et al 1990] Wirfs-Brock, R., Wilkerson, B., and Wiener, L., *Designing Object-Oriented Software*, 1 ed. Englewood Cliffs: Prentice Hall, 1990.
- [WIRTH, N. 1995] Wirth, N., "A Plea for Lean Software," in *IEEE Computer*, 1995, pp. 64-68.
- [YOURDON, E. 1988] Yourdon, E., *Modern Structured Analysis*. Upper Saddle River: Prentice Hall PTR, 1988.

Curriculum Vitae

José Diego De la Cruz

Academic Experience	Research Assistant	2005-2007 2002-2003
	Fellowship Holder EPFL , Switzerland	2000-2002
	Instructor, Project lead Universidad del Cauca , Colombia	1998-2000
Industrial Experience	<i>Ensures Consulting</i> Senior Consultant Switzerland	2007-present
	IT Consultant	2003-2005 1998-2001
	<i>Control Electrónico Ltda</i> Division Mgr, Project lead Colombia	1996-1998
	<i>Antares Tecnología Ltda</i> Project lead, Engineer Colombia	1995-1996
	<i>CTD Project</i> Research Engineer Colombia	1994
	<i>Universidad del Cauca</i> (IT Services) Programmer Colombia	1989-1994
Education	Ph.D student EPFL , Switzerland	2004-present
	Operational Manager in Project Management CEFCO , Switzerland	2004-2005
	Graduate School in Computer Science EPFL , Switzerland	2001-2002
	Postgrade in Language and Speech Engineering EPFL , Switzerland	2001-2002
	Ingeniero en Electrónica y Telecomunicaciones Universidad del Cauca , Colombia Mención de Honor (<i>Equivalent to Suma Cum Laude</i>)	1988-1994

Publications

Monographies and edited books

J. D. De La Cruz. *SDDT- Symbolic Distributed Real-Time Debugging Tool*. Thesis. Universidad del Cauca. 1997.

J. D. De La Cruz. *SDDT – Symbolic Debugging Real-Time Distributed Tool*. Thesis, **Mención de Honor** (Equivalent to *Suma Cum Laude*). Universidad del Cauca. 1997.

J. D. De La Cruz, et al (ed.). *Join II – Segundas jornadas de Informática Universitaria*. Proceedings. Universidad del Cauca. 1992.

J. D. De La Cruz, C.A. Yopez. *Las teorías políticas en la historia*. Prix à la meilleure monographie pour baccalauréat. Colegio S.J. Berchmans. 1986.

Journals

A. Wegmann, G. Regev, J. D. de la Cruz, L.-S. Lê, and I. Rychkova. Teaching Enterprise and Service-Oriented Architecture in Practice. *Journal of Enterprise Architecture*, (accepted for publication), 2007.

Arteaga, G. - Acosta, D.A. - De la Cruz, J.D. - Rendón, A. *SMART - Sistema Modular para Aplicaciones en RI y Telemática*. In: Revista Colombiana de Telecomunicaciones p.36 - 43 , 2000

Conference Papers

A. Wegmann, G. Regev, I. Rychkova, L.-S. Lê, J. D. De La Cruz, and P. Julia. Business-IT Alignment with SEAM for Enterprise Architecture. In *The 11th IEEE International EDOC Conference (EDOC 2007)*. IEEE, 2007.

De la Cruz, José D. - Lê, Lam-Son - Wegmann, Alain. *VISUAL CONTRACTS: A way to reason about states and cardinalities in IT system specifications*. In: Proceedings of the Eighth International Conference on Enterprise Information Systems p.298-303 , 2006

De la Cruz, J.D. - Tamura, E. - Rendón, A. *Ambiente CTD: desarrollo de sistemas de tiempo real utilizando especificaciones ejecutables*. In: Memorias del VIII Congreso Latinoamericano de Control Automático CLCA'98 , 1998

Workshops Papers

A. Wegmann, L.-S. Lê, J. D. de la Cruz, I. Rychkova, and G. Regev. An Example of a Hierarchical System Model Using SEAM and its Formalization in Alloy. In *4th International Workshop on ODP for Enterprise Computing (WODPEC 2007)*, 2007.

De la Cruz, José D. - Lê, Lam-Son - Wegmann, Alain. *Validation of Visual Contracts for Services*. In: Fourth International Workshop on "Modeling, Simulation, Verification and Validation of Enterprise Information Systems , 2006

De La Cruz, José Diego - Regev, Gil - Wegmann, Alain. *Expressing Systemic Contexts in Visual Models of System Specifications*. In: Workshop on Context Modeling and Decision Support , 2005