THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par : *l'Institut Supérieur de l'Aéronautique et de l'Espace (ISAE)*

Présentée et soutenue le *28 Juin 2016* **par :**

VINCENT LECRUBIER

Un langage formel pour la conception, la spécification et la vérification d'interfaces homme-machine embarquées critiques

---

JURY

LAURENCE NIGAY	Professeur des Universités	Présidente
CHRISTOPHE KOLSKI	Professeur des Universités	Rapporteur
PATRICK GIRARD	Professeur des Universités	Rapporteur
PHILIPPE PALANQUE	Professeur des Universités	Examinateur
BRUNO D'AUSBOURG	Ingénieur de recherche HDR	Directeur de thèse
YAMINE AÏT-AMEUR	Professeur des Universités	Directeur de thèse

---

École doctorale et spécialité :
MITT : Domaine STIC : Réseaux, Télécoms, Systèmes et Architecture
Unité de Recherche :
Office National d'Études et de Recherche Aérospatiale (ONERA) : DTIM
Directeur(s) de Thèse :
Bruno d'AUSBOURG et *Yamine AÏT-AMEUR*
Rapporteurs :
Christophe KOLSKI et *Patrick GIRARD*

Remerciements

Tout d'abord, je tiens à exprimer tout ma gratitude à mes deux co-directeurs de thèse, Bruno d'Ausbourg et Yamine Aït-Ameur. De l'eau a coulé sous les ponts depuis ma présentation de projet de fin d'études en 2011 jusqu'aujourd'hui, mais vous avez toujours été là pour moi. Pourtant, gérer un thésard sportif de haut niveau, Breton qui plus est, ce n'est pas facile tous les jours ! Je vous remercie de m'avoir consacré tout ce temps et cette énergie, dans les moments simples comme dans les moments difficiles. Merci d'avoir été si généreux et compréhensifs. Scientifiquement, ce travail vous doit beaucoup. La liberté que vous m'avez donné m'a permis d'explorer des terrains très variés et instructifs, et vos recadrages occasionnels m'ont permis de ne pas perdre pied. Vous avez joué des rôles complémentaires dans l'encadrement de ma thèse, Bruno avec ta bienveillance au quotidien, et Yamine avec tes interventions salutaires à des moments clés. Nos débats scientifiques tumultueux resteront dans ma mémoire comme de grands moments de stimulation intellectuelle. Vous faites tous les deux une belle équipe et je suis fier d'avoir travaillé avec vous.

Je tiens à remercier les membres du Jury pour me faire l'honneur de leur présence à ma soutenance de thèse, et pour l'intérêt qu'ils ont porté à mes travaux. C'est un grand honneur de voir de si grands noms du monde de la recherche associés au mien. Je ne peux qu'imaginer la difficulté du travail de rapporteur de Thèse, merci donc à Patrick Girard et Christophe Kolski pour votre investissement et vos retours. Merci à Philippe Palanque et Laurence Nigay pour m'accompagner dans cette étape importante qu'est la soutenance d'une thèse.

Un grand merci aux équipes du DTIM de l'ONERA Toulouse pour votre accueil et votre sympathie tout au long de ces quelques années. J'aurais aimé pouvoir être plus présent au sein du DTIM, et la bonne ambiance et l'esprit de collaboration au sein du département restent quelque chose d'exemplaire à mes yeux. Merci en particulier aux collègues avec lesquels j'ai fait des TP de C à l'ISAE, pour m'avoir donné cette chance et donné de goût d'enseigner: Pierre Siron, Martin Adelantado, Eric Noulard, Olivier Poitou, Vincent Vidal, Jean-Loup Bussenot...

Merci aux copains doctorants du DTIM, les anciens comme les plus jeunes, pour tous les moments passés ensemble, de l'ONERA à Rome en passant par le centre ville de Toulouse. On a travaillé, on a discuté d'un paquet de choses très intéressantes, et on s'est aussi bien amusés pendant tout ce temps ! Je garderai de superbes souvenirs de tout cela ! Je m'excuse de ne pas pouvoir tous vous citer, je vais me limiter à deux noms: Tomasz Kloda, merci pour tout, mon "frère de thèse", et Rémi Wyss, merci pour ta force et ta bonne humeur jusqu'au bout, et ça c'est cadeau, je t'avais dit que je le placerais :  !

Merci à l'EDMITT, à l'ISAE pour avoir été si accommodants et compréhensifs face à ma situation de complexe de sportif-thésard. Merci à la fondation ISAE-SUPAERO, et en particulier aux équipes de OSE l'ISAE pour m'avoir donné l'occasion d'être parrain de l'association aux côtés d'illustres anciens Supaéros, et de retransmettre aux jeunes générations la passion et le rêve de l'aérospatial, de la science et du progrès. Je remercie particulièrement Joël Daste, qui m'a appelé au téléphone un beau jour d'Août 2006 pour m'annoncer que j'étais admis à Supaéro, et qui a toujours cru en moi depuis.

Merci aux copains, anciens ou récents coéquipiers, entraineurs et bénévoles du Kayak, de l'Équipe de France, du Pôle France de Toulouse, du Pôle France de Rennes, du Toulouse Flatwater Crew et aussi évidemment du CKCIR, ma deuxième famille. Vous avez aussi joué votre rôle dans cette thèse, étant donné le temps que j'ai passé à y travailler et à y réfléchir en votre compagnie, que ce soit sur mon Kayak, ou pendant les stages et compétitions. Merci à vous pour tous ces superbes moments d'entrainement, d'effort, de voyage, de détente, et de découvertes. Le sport de plein air, c'est clairement l'un des meilleurs moyens d'être bien dans sa peau et dans sa tête, et ça se voit avec vous ! Je ne vais pas tous vous citer car je sais que vous vous reconnaitrez dans ces remerciements. Je vais me limiter à un nom: Sylvain Homo, merci mon pote, j'aurais aimé pouvoir pagayer un peu plus avec toi.

Enfin, comment terminer ces longs remerciements autrement que par ma famille. Merci à ma soeur Marine pour tes encouragements et ton soutien infaillible depuis toujours. Un grand merci à Tilly, qui réussit à supporter les contraintes combinées d'un thésard et d'un sportif depuis de nombreuses années, tout en apportant son énergie et sa magie à ma vie. Merci de toujours être là pour moi, y compris quand plusieurs centaines de kilomètres nous séparent. Finalement, je tiens à remercier ceux sans qui absolument rien de tout cela n'aurait été possible, mes parents Sylvie et Gilbert. Vous avez vécu une vie entière pour vos enfants, et je ne saurais jamais exprimer toute ma gratitude, pour tout ce que je vous doit. Maman, merci pour absolument tout dans ces 29 ans, et toute l'énergie que tu m'a donné. Papa, là où tu es, je sais que tu serais fier de moi, et ce travail t'es dédié.

Contents

I Version courte en Français **1**

1 Des défis à relever **2**

1.1 Contexte 2

1.2 Le problème 3

1.3 Vers une solution 4

2 État de l'art **6**

2.1 Modèles d'architecture d'Interface Homme-Machine (IHM) 7

2.2 Modèles d'architecture de réseaux 7

2.3 Modèles basés sur des interacteurs 7

2.4 Développement d'IHM à l'aide de langages impératifs 7

2.5 Langages dédiés (DSLs) 7

2.6 Langages de balisage 7

2.7 Bibliothèques basées sur le (DOM) 7

2.8 Programmation fonctionnelle réactive (FRP) 7

2.9 Modélisation graphique du comportement des IHM 7

2.10 Modèles de tâches 7

2.11 Vérification formelle d'IHM 7

2.12 Outils commerciaux pour les IHM critiques 7

3 Architecture **8**

4 LIDL **9**

5 Implémentation **10**

6 Application à un cas d'étude **11**

7 Conclusion **12**

II Full version **13**

1 Challenges **14**

1.1 Context . . . 14

1.1.1 Critical embedded software . . . 14

1.1.2 HMI software . . . 16

1.1.3 Critical embedded HMI software . . . 18

1.2 The problem . . . 19

1.2.1 New Human-Machine Interface (HMI) techniques time-to-market . . . 20

1.2.2 Complexity of the design process . . . 20

1.2.3 Difficulty of specification . . . 21

1.2.4 Difficulty of implementation . . . 22

1.2.5 Difficulty of verification . . . 22

1.2.6 Reusability difficulties . . . 23

1.2.7 Integrating the human factor into the process . . . 24

1.3 Requirements for a solution . . . 24

1.3.1 Formal semantics . . . 24

1.3.2 Team work and collaboration . . . 24

1.3.3 Projection and Traceability . . . 25

1.3.4 Modularity . . . 25

1.3.5 Abstraction . . . 25

1.3.6 User Centered Design . . . 26

1.3.7 Support formal verification . . . 26

1.3.8 Flexibility . . . 26

1.3.9 Synopsis of requirements . . . 26

2 State of the art **28**

2.1 HMI architecture . . . 29

2.1.1 Seeheim . . . 29

2.1.2 Arch . . . 30

2.1.3 MVC . . . 31

2.1.4 PAC . . . 32

2.1.5 PAC Amodeus . . . 33

2.1.6 MVP . . . 33

2.1.7 Cocoa . . . 34

2.1.8 Swing . . . 35
2.1.9 MVVM . . . 36
2.1.10 Qt . . . 37
2.1.11 Flux . . . 38
2.1.12 ARINC . . . 39
2.1.13 Conclusion . . . 40
2.2 Network architecture . . . 42
2.2.1 OSI . . . 42
2.2.2 Geomorphic . . . 43
2.2.3 Conclusion . . . 44
2.3 Interactors . . . 46
2.3.1 CNUCE . . . 46
2.3.2 York . . . 47
2.3.3 CERT . . . 47
2.3.4 Conclusion . . . 48
2.4 Imperative languages . . . 50
2.4.1 Qt . . . 50
2.4.2 Swing . . . 52
2.4.3 Cocoa . . . 53
2.4.4 Conclusion . . . 54
2.5 Textual DSLs . . . 56
2.5.1 QML . . . 56
2.5.2 JavaFX DSLs . . . 57
2.5.3 Conclusion . . . 58
2.6 UI markup . . . 60
2.6.1 HTML . . . 60
2.6.2 XML DSLs . . . 61
2.6.3 ARINC DF . . . 61
2.6.4 Conclusion . . . 62
2.7 DOM frameworks . . . 65
2.7.1 jQuery . . . 65
2.7.2 AngularJS . . . 66
2.7.3 React . . . 67
2.7.4 Conclusion . . . 68

2.8 FRP . . . 70
2.8.1 Reactive banana . . . 70
2.8.2 Elm . . . 71
2.8.3 Conclusion . . . 72
2.9 Graphical . . . 74
2.9.1 ICOs . . . 74
2.9.2 FILL . . . 75
2.9.3 Max/MSP . . . 77
2.9.4 Storyboards . . . 77
2.9.5 Conclusion . . . 78
2.10 Task models . . . 81
2.10.1 CTT . . . 81
2.10.2 EOFM . . . 81
2.10.3 Petri nets . . . 82
2.10.4 Conclusion . . . 83
2.11 UI verification . . . 86
2.11.1 Model Checking . . . 86
2.11.2 Model discovery . . . 87
2.11.3 Theorem proving . . . 88
2.11.4 Conclusion . . . 89
2.12 Commercial tools . . . 91
2.12.1 Presagis . . . 91
2.12.2 Esterel . . . 92
2.12.3 Conclusion . . . 93
2.13 Conclusion . . . 95

3 The geomorphic view of interactive systems 97
3.1 Introduction . . . 97
3.2 Presentation of the model . . . 98
3.2.1 Interactors . . . 98
3.2.2 Layers and channels . . . 99
3.2.3 Interactive systems and attachments . . . 100
3.2.4 Overview . . . 101
3.3 Common patterns . . . 104
3.3.1 Rendering . . . 104

3.3.2 Multimodality . . . 105
3.3.3 Mobility . . . 106
3.3.4 Client-server . . . 106
3.3.5 Interface . . . 108
3.3.6 Routing . . . 108
3.4 Example systems . . . 109
3.4.1 WIMP Interfaces . . . 109
3.4.2 Multimodal interfaces . . . 113
3.4.3 ARINC661 CDS . . . 116
3.4.4 Web application . . . 119
3.5 Conclusion . . . 123
3.5.1 Limitations . . . 123
3.5.2 Compliance with requirements . . . 123
3.5.3 Perspectives . . . 123

4 LIDL 125
4.1 Getting started . . . 125
4.1.1 A first simple example . . . 126
4.1.2 A second example . . . 127
4.2 Presentation of the language . . . 129
4.2.1 Definitions . . . 129
4.2.2 Data . . . 130
4.2.3 Interfaces . . . 131
4.2.4 Interactions . . . 132
4.2.5 Execution . . . 134
4.2.6 Base interactions . . . 136
4.3 Conclusion . . . 140
4.3.1 Overview . . . 140
4.3.2 Assessment of requirements . . . 141

5 LIDL implementation 142
5.1 Compiler . . . 142
5.1.1 Overview . . . 143
5.1.2 Architecture . . . 143
5.1.3 Example compilation . . . 145

5.1.4 Limitations and perspectives . . . 149
5.2 Standard library . . . 151
5.2.1 Data operators . . . 152
5.2.2 Behaviours . . . 153
5.2.3 State related interactions . . . 154
5.2.4 Abstract widget library . . . 156
5.2.5 Concrete widget library . . . 157
5.2.6 Task modelling . . . 160
5.3 Canvas . . . 162
5.3.1 Architecture . . . 162
5.3.2 Interface . . . 163
5.3.3 Execution model . . . 163
5.3.4 Representation of graphics . . . 165
5.3.5 Limitations and perspectives . . . 166
5.4 Sandbox . . . 166
5.4.1 Architecture . . . 166
5.4.2 Usage . . . 167
5.4.3 Limitations and perspectives . . . 169

6 Case study **171**
6.1 Presentation . . . 171
6.1.1 Overview . . . 171
6.1.2 Architecture . . . 171
6.2 Model . . . 174
6.2.1 The Alarm object . . . 174
6.2.2 The Model object . . . 182
6.2.3 Conclusion . . . 183
6.3 Abstract interface . . . 185
6.3.1 Button . . . 186
6.3.2 Alarm Item . . . 188
6.3.3 Alarm List . . . 191
6.3.4 Alarm management User Interface (UI) . . . 193
6.3.5 Conclusion . . . 194
6.4 Concrete interface . . . 196
6.4.1 Button . . . 196

6.4.2 Alarm item . . . 199
6.4.3 Alarm List . . . 200
6.4.4 Complete UI . . . 201
6.4.5 Conclusion . . . 202
6.5 Conclusion . . . 204
6.5.1 Overview of the model . . . 204
6.5.2 Assessment of requirements . . . 205

7 Conclusion **207**
7.1 Overview . . . 207
7.2 Perspectives . . . 208
7.2.1 Geomorphic model . . . 208
7.2.2 LIDL Interaction Description Language (LIDL) . . . 209

III Appendix **218**

A Emerging concepts in LIDL **219**
A.1 Interaction hierarchy rotation . . . 219
A.2 Variables resolving operators . . . 222
A.3 Generics . . . 224
A.4 Data type interactions . . . 224
A.4.1 Construction . . . 225
A.4.2 Comparison . . . 225
A.4.3 Mutation . . . 225

B Compiler graph transformations **227**
B.1 Definition expansion . . . 227
B.2 Base interaction . . . 227
B.2.1 Compilation process . . . 227

C LIDL reference manual **235**
C.1 Definitions . . . 235
C.1.1 Syntax . . . 235
C.1.2 Simple definitions . . . 235
C.1.3 Complex definitions . . . 235
C.1.4 Definitions scope . . . 237

C.1.5 Definition validity . . . 239
C.1.6 LIDL System . . . 239
C.2 Data . . . 240
C.2.1 Syntax . . . 240
C.2.2 Atomic data types . . . 240
C.2.3 Composite data types . . . 241
C.2.4 Function types . . . 242
C.2.5 Structural typing . . . 242
C.2.6 Data activation . . . 244
C.3 Interfaces . . . 244
C.3.1 Syntax . . . 244
C.3.2 Atomic interfaces . . . 245
C.3.3 Composite interfaces . . . 245
C.3.4 Interface operations . . . 248
C.3.5 Remarks and rationales . . . 250
C.4 Interactions . . . 250
C.4.1 Syntax . . . 250
C.4.2 Interaction expression . . . 251
C.4.3 Interaction operator . . . 251
C.4.4 Interaction operand . . . 253
C.4.5 Each interaction is a signal . . . 253
C.4.6 Remarks and rationales . . . 253
C.5 Base interactions . . . 254
C.5.1 Function . . . 254
C.5.2 Identifier . . . 255
C.5.3 Behaviour . . . 257
C.5.4 State . . . 258
C.5.5 Composition . . . 258
C.5.6 Function Application . . . 260
C.6 Semantics . . . 264
C.6.1 Definition expansion . . . 264
C.6.2 Interpretation . . . 264
C.6.3 Compilation . . . 264

List of Tables

1.1 Identified requirements 27

2.1 Requirements vs HMI architecture state of the art 41
2.2 Requirements vs network architecture models state of the art 45
2.3 Requirements vs interactor based models state of the art 49
2.4 Requirements vs imperative programming of UIs state of the art 55
2.5 Requirements vs textual UIs Domain-Specific Languages (DSLs) state of the art . . . 59
2.6 Requirements vs UIs markup languages state of the art 64
2.7 Requirements vs Document Object Model (DOM) based frameworks state of the art . 69
2.8 Requirements vs Functional Reactive Programming (FRP) state of the art 73
2.9 Requirements vs graphical modelling of UI behaviour state of the art 80
2.10 Requirements vs task modelling state of the art 85
2.11 Requirements vs formal methods for UIs behaviour state of the art 90
2.12 Requirements vs state of the art commercial tools 94
2.13 Requirements vs state of the art 96

4.1 Several notation options 133
4.2 Correspondence between Mealy machines and LIDL programs. 135
4.3 Timing diagram of an execution of the LIDL program of Listing 4.1 136
4.4 Assessment of LIDL solutions to requirements 141

5.1 Timing diagram of an execution of nested state interaction 155

6.1 Timing diagram of an execution of the abstract button widget 187
6.2 Execution of the alarm item interaction 190
6.3 A synopsis of the requirements and what the case study tells us about them 206

A.1 Timing diagram of an execution of the program of Listing A.4 224

C.1 Example simple definitions 237
C.2 Composite data types examples 241

C.3 Interfaces examples . . . 245
C.4 Atomic interfaces examples . . . 245
C.5 Interfaces examples . . . 247
C.6 Some example of toDataType. . . . 248
C.7 Some example of conjugation. . . . 249
C.8 Some example of Interface union. . . . 249
C.9 Examples of interaction operators . . . 252
C.10 Examples of valid functions operators . . . 255
C.11 Examples of valid identifiers operators . . . 256
C.12 Timing diagram of an execution of an identifier interaction . . . 256
C.13 Timing diagram of an execution of the previous interaction . . . 257
C.14 Timing diagram of an execution of the stateful interaction . . . 259
C.15 Timing diagram of an execution of a composition interaction . . . 260
C.16 Timing diagram of an execution of a function application interaction . . . 261
C.17 Timing diagram of an execution of a function application interaction . . . 262
C.18 Timing diagram of an execution of a function application interaction . . . 262
C.19 Timing diagram of an execution of a function application interaction . . . 263
C.20 Timing diagram of an execution of a function application interaction . . . 263

List of Figures

1.1 Le contexte de cette thèse, à l'intersection de deux disciplines 3
1.2 Le délai entre l'apparition des technologies et leur application aux systèmes critiques . 4

1.1 The context of this thesis is at the crossroads of broader domains 15
1.2 A typical development process for critical embedded software 16
1.3 A typical development process for HMI software 18
1.4 A typical development process for critical embedded HMI software 19
1.5 The delay before application of new interaction techniques in the field of critical systems 20

2.1 Positioning of the groups of approaches presented, along with their section number. . . 29
2.2 The Seeheim Model 30
2.3 The Arch Model 31
2.4 The MVC Model as presented in [31] 32
2.5 The Presentation Abstraction Control Model 33
2.6 The PAC-Amodeus Model 34
2.7 The MVP Model 34
2.8 The Cocoa version of the MVC Model 35
2.9 The separable model architecture of Swing 36
2.10 The Presentation Model (PM) and Model View View-Model (MVVM) architecture . . 37
2.11 The architecture of Qt programs 38
2.12 The Flux architecture Model 39
2.13 The logical architecture model of an ARINC 661 system 40
2.14 The classical Open Systems Interconnection (OSI) Model 43
2.15 Geomorphic view of the classic Internet architecture. 44
2.16 Centro Nazionale Universitario di Calcolo Elettronico (CNUCE) interactors internal architecture 47
2.17 York interactors internal architecture 48
2.18 An architecture based on Centre d'Études et de Recherche de Toulouse (CERT) interactors 48
2.19 A system described using Interactive Cooperative Objects (ICOs) 75

2.20 A Formal Interaction Logic Language (FILL) system (left) with its representation as a petri net (right) . . . 76
2.21 An example of Max/MSP program with the "patching view" (left) and "presentation view" (right) . . . 78
2.22 An example of Apple storyboard . . . 79
2.23 A Concur Task Tree (CTT) task model . . . 82
2.24 A Enhanced Operator Function Model (EOFM) description of a task model . . . 83
2.25 Object Petri nets to describe user tasks as in [104] . . . 84
2.26 General workflow for model checking of UIs . . . 86
2.27 Workflow of model discovery and theorem discovery . . . 87
2.28 State spaces of two different number entry systems described in [114] . . . 88
2.29 Workflow for the validation of UIs using proof methods . . . 89
2.30 Set of Commercial Off The Shelf (COTS) tools used to create DO 178B UI Systems . 91
2.31 The set of Presagis tools in the context of ARINC 661 systems . . . 92

3.1 Six different entities that can interact . . . 98
3.2 Some interactors depitcted using the graphical representation . . . 99
3.3 Interactors and channels within layers . . . 100
3.4 Two interactive systems . . . 101
3.5 Metamodel of the geomorphic view of interactive systems as a Unified Modelling Language (UML) class diagram . . . 102
3.6 A view of a simple model with emphasis on layers . . . 102
3.7 A view of a simple model with emphasis on interactive systems . . . 103
3.8 A view of a complex model with emphasis on layers (Vertical view) . . . 103
3.9 A view of a complex model with emphasis on interactive systems (Horizontal view) . . 104
3.10 Rendering Pattern . . . 105
3.11 Multimodality pattern . . . 106
3.12 Mobility pattern . . . 107
3.13 Client-Server Pattern . . . 107
3.14 Interface Pattern . . . 108
3.15 Routing Pattern . . . 109
3.16 A vertical geomorphic view of a typical WIMP Architecture (Arch Model in Yellow) . 110
3.17 An horizontal geomorphic view of a typical WIMP Architecture . . . 111
3.18 Three different feedbacks for a "Add to cart" Button . . . 112
3.19 Feedback loops in a WIMP architecture shown on an vertical view . . . 112

3.20 Feedback loops in a WIMP architecture shown on an horizontal view 113
3.21 A vertical geomorphic view of a multimodal UI architecture 114
3.22 An horizontal geomorphic view of a multimodal UI architecture 114
3.23 A parallel between the choice of the easiest path and the choice of the easiest modality 115
3.24 A geomorphic view of a configuration with two redundant ARINC 661 CDS 116
3.25 Another geomorphic view of a configuration with two redundant ARINC 661 CDS . . 117
3.26 Feedback loops in an ARINC 661 system 118
3.27 A vertical geomorphic view of the typical architecture of web applications 120
3.28 An horizontal geomorphic view of the typical architecture of web applications 120
3.29 Two feedback loops in a Web UI 121
3.30 A common architecture for modern web UI with use of a local cache to reduce feedback delays 122

4.1 An overview of the four components of interactive systems 125
4.2 A simple UI 127
4.3 Execution of LIDL programs 134
4.4 A mealy machine corresponding to the program of Listing 4.1 135

5.1 LIDL compilation process 143
5.2 Architecture of the LIDL compiler 144
5.3 Compilation graph after the *addDefinitionToGraph* step 145
5.4 Compilation graph after the *referentialTransparency* step 146
5.5 Compilation graph after the *expandDefinitions* phase 146
5.6 Compilation graph after the *instantiateInterfaces* phase 147
5.7 Compilation graph after the *linkIdentifiers* phase 147
5.8 Compilation graph after the *behaviourSeparation* phase 148
5.9 Compilation graph after the *functionApplicationLinking* phase 148
5.10 Compilation graph after the *affectationLinking* phase 148
5.11 Compilation graph after the *orderGraph* phase 149
5.12 Geomorphic view of the LIDL Canvas environment 163
5.13 Sequence diagram of event handling in LIDL Canvas 164
5.14 Example graphics rendered in LIDL canvas 165
5.15 Architecture of the LIDL sandbox 167
5.16 As screenshot of the LIDL sandbox 168
5.17 As screenshot of the LIDL sandbox 168

5.18 Data flow within the LIDL sandbox 170

6.1 A screenshot of the HMI described in the case study 172
6.2 Geomorphic view of the architecture of the system described in the case study 173
6.3 UML class diagram of the model managed by the UI 174
6.4 Finite State Machine representing the life cycle of Alarms 179
6.5 A view of the architecture model at this point in the development 184
6.6 Component hierarchy of the UI. 186
6.7 A view of the architecture model at this point in the development 195
6.8 Mapping between concrete and abstract buttons 196
6.9 Mapping between concrete and abstract interface of a alarm list item 199
6.10 Mapping between concrete and abstract interface of a alarm list 200
6.11 Mapping between concrete and abstract interface of the complete UI 201
6.12 A view of the architecture model at this point in the development 203
6.13 Final overview of the architecture of the system 204
6.14 Three dimensions of UI architecture 205

A.1 A behaviour with three arguments, and its three rotations 221
A.2 The interaction composition tree and data flow of Listing A.4 223

B.1 The referential transparency rule 229
B.2 The identifier elimination rule 229
B.3 The behaviour separation rule 230
B.4 The function literal linking rule 230
B.5 The synchronous function call transformation rule 231
B.6 The asynchronous function call transformation rule 231
B.7 The composition ordering rule 232
B.8 The composition matching rule 233
B.9 The composition transformation rule 233
B.10 The multiple resolving rule 234

C.1 Graphical representation of an example definition tree. 236
C.2 Graphical representation of definitions scoping in LIDL. The red definition can refer to blue definitions because they are its ancestors and to yellow definitions because they are direct children of its ancestors, but not to itself. See Definition C.1 237

C.3 Graphical representation of definitions scoping in LIDL. The red definition is exposed to purple definitions because they are its descendants, and to green definitions because they are the descendants of its parent, but not to itself. See Corollary C.1 238
C.4 A metaphore of atomic data types. Pieces of data are represented on top, and their associated data types are represented as “molds" that allow to create them and check wheter a piece of data is of a certain type. 241
C.5 A metaphore of composite data types. Composite data types (on the right) are data types that fit composed data (left) 242
C.6 The main point for composite data types in LIDL: Resolving circular dependencies by going deeper in the data structure. 243
C.7 A metaphore of structural typing. Data types with different names but the same structure are equivalent. 243
C.8 A metaphore of data activation. Some pieces of data might not be present but the data is still a valid instance of its data type. 244
C.9 A metaphore of atomic interfaces. Interfaces are pipes that let data (left) go in one direction: either out (red) our in (blue). The data can be either atomic (top) or composite (bottom) 246
C.10 A metaphore of composite interfaces. Composite interfaces are (black) pipes that contain other pipes. In this figure, one of the four interfaces is an atomic interface. 247
C.11 A metaphore of interactions. Interactions are components which have several interfaces and can be composed in order to make more complex systems. 250
C.12 Graphical view of an exemple interaction expression made of 5 interactions. 251
C.13 A metaphore of interaction expressions. This figure represents a metaphore of the expression of Figure C.12 252
C.14 Sequence diagram of the execution of a LIDL runtime in an event-driven environnement 265
C.15 Sequence diagram of the execution of a LIDL runtime in an event-driven environnement, Focus on the LIDL Runtime 266

List of Listings

1.1 An example snippet that can be encountered in an industrial context Detailed Function Specification (DFS) . . . 22
2.1 The code of a simple QT application . . . 51
2.2 Code of a simple Java-swing application . . . 52
2.3 Code of a simple Swift Cocoa application . . . 54
2.4 The code of a simple application using Qt Modelling Language (QML) . . . 57
2.5 The code of a simple GroovyFX UI . . . 58
2.6 A simple UI described in HyperText Markup Language (HTML) and Javascript (JS) . . . 61
2.7 An ARINC 661 eXtensible Markup Language (XML) definition file describing a simple application . . . 63
2.8 A simple UI described in HTML and using jQuery . . . 65
2.9 Code of a simple application with Angular JS . . . 66
2.10 The code of a simple React Application . . . 67
2.11 The code of a simple application using the Reactive banana Haskell library . . . 71
2.12 The code of a simple application using elm . . . 72
4.1 An interaction with one input, one state variable and one output . . . 135
5.1 LIDL code of our example interaction . . . 145
5.2 Code generated from our example interaction . . . 150
5.3 Definitions of several data operators in the LIDL standard library . . . 152
5.4 An example of composition of behaviours . . . 153
5.5 Definitions of several behaviour interactions in the LIDL standard library . . . 154
5.6 Definitions of several state related interactions in the LIDL standard library . . . 156
5.7 An example abstract UI using standard widgets . . . 157
5.8 Definition of the abstract button widget . . . 157
5.9 An example concrete UI specified using standard LIDL components . . . 158

5.10 The interfaces used in definitions of Windows, Icons, Menu and Pointer interface (WIMP) interfaces . . . 159
5.11 . . . 160
5.12 An example task model modelled using elements of the standard library . . . 161
5.13 The task interface, part of the standard library . . . 161
5.14 Definition of the sequential task composition operator . . . 161
5.15 Example LIDL code describing the graphics of Figure 5.14 . . . 165
6.1 The Alarm data type . . . 175
6.2 A constructor for the Alarm data type . . . 176
6.3 An example of use of the mutation of the alarm data type . . . 177
6.4 Mutations of the alarm data type . . . 178
6.5 LIDL implementation of the Finite State Machine (FSM) of the alarm life cycle . . . 180
6.6 Integration of the FSM to specify a mutation of Alarm objects . . . 181
6.7 An interaction that tells if an action is accepted by the FSM in a given state . . . 181
6.8 The Model data type and some associated interactions . . . 182
6.9 An example declarative description of manipulations of a model . . . 183
6.10 Definition of the behaviour of the UI Model interactor . . . 184
6.11 Definition of the behaviour of the user interactor in the model layer . . . 184
6.12 Definition of the behaviour of the rover interactor in the model layer . . . 185
6.13 The global interactions between interactors in the Model layer . . . 185
6.14 The Alarm Item component interface . . . 188
6.15 The Alarm Item interaction . . . 189
6.16 The alarm list interface . . . 191
6.17 The alarm list interaction . . . 192
6.18 The alarm management interface . . . 193
6.19 The interaction of the alarm management abstract UI . . . 194
6.20 Definition of the behaviour of the Abstract UI interactor . . . 194
6.21 The global interaction between interactors in the Model and Abstract UI layers . . . 195
6.22 The definition of a concrete button widget . . . 198
6.23 Definition of the concrete alarm item component . . . 200
6.24 Definition of the concrete alarm list component . . . 201
6.25 Definition of the concrete alarm list component . . . 202
6.26 Definition of the behaviour of the Abstract UI interactor . . . 202
6.27 The global interaction between the user and the UI, taking into account 3 layers . . . 203

A.1 Definition of a behaviour associated with the `(sin(x))` interaction 219
A.2 Two expressions describing the same computation. 220
A.3 Three valid ways to see a “button to increment a value” 222
A.4 A simple LIDL program involving the variable `(x)` 222
A.5 225
A.6 225
A.7 225
A.8 226
C.1 An example of nested complex definitions 236
C.2 Actual examples of complex definitions in LIDL 238
C.3 239
C.4 A version of functionSum in Javascript 255
C.5 A version of functionSum in C 256

List of Acronyms

AFDX Avionics Full DupleX

AJAX Asynchronous Javascript And XML

API Application Programming Interface

ARINC American Radio Incorporated

ARINC661 American Radio Incorporated standard 661

AST Abstract Syntax Tree

BPMN Business Process Modelling Notation

CERT Centre d'Études et de Recherche de Toulouse

CDS Cockpit Display System

CNUCE Centro Nazionale Universitario di Calcolo Elettronico

COTS Commercial Off The Shelf

CRUD Create, Read, Update, Delete

CSS Cascading Style Sheets

CTL Computation Tree Logic

CTT Concur Task Tree

DAG Directed Acyclic Graph

DF Definition File

DFS Detailed Function Specification

DOM Document Object Model

DSL Domain-Specific Language

DSM Domain-Specific Model

EBNF Extended Backus-Naur Form

ENSEEIHT École Nationale Supérieure d'Électrotechnique, d'Électronique, d'Informatique, d'Hydraulique et des Télécommunications

EOFM Enhanced Operator Function Model

FILL Formal Interaction Logic Language

FP Functional Programming

FRD Functional Requirements Document

FRP Functional Reactive Programming

FSM Finite State Machine

GUI Graphical User Interface

HDMI High-Definition Multimedia Interface

HMI Human-Machine Interface

HCI Human-Computer Interaction

HRD High-level Requirements Document

HTML HyperText Markup Language

HTTP HyperText Transfer Protocol

ICO Interactive Cooperative Object

IDE Integrated Development Environment

IHM Interface Homme-Machine

IP Internet Protocol

IS Interactive Systems

JS Javascript

JSON Javascript Object Notation

JVM Java Virtual Machine

LAF Look And Feel

LAN Local Area Network

LIDL LIDL Interaction Description Language

LOC Lines Of Code

LOTOS Language Of Temporal Ordering Specification

LTL Linear Temporal Logic

MAC Media Access Control

MAL Modal Action Logic

MODEM Modulator Demodulator

MXML Macromedia eXtensible Markup Language

MVC Model View Controller

MVP Model View Presenter

MVVM Model View View-Model

NPM Node Package Manager

OOP Object-Oriented Programming

OSI Open Systems Interconnection

OS Operating System

PAC Presentation Abstraction Control

PEG Parsing Expression Grammar

PM Presentation Model

PHP PHP: Hypertext Preprocessor

QML Qt Modelling Language

REST Representational State Transfer

RP Reactive Programming

SAL Software Analysis Laboratory

SQL Structured Query Language

SSL Secure Sockets Layer

TCP Transmission Control Protocol

TDD Test Driven Development

TRL Technology Readiness Level

UA User Application

UCD User Centered Design

UDP User Datagram Protocol

UI User Interface

UIML User Interface Markup Language

UML Unified Modelling Language

USB Universal Serial Bus

UsiXML User Interface eXtensible Markup Language

VPN Virtual Private Network

W3C World Wide Web Consortium

WCET Worst Case Execution Times

WIMP Windows, Icons, Menu and Pointer interface

WYSIWYG What You See Is What You Get

XAML eXtensible Application Markup Language

XML eXtensible Markup Language

XUL XML User Interface Language

Part I

Version courte en Français

Chapter 1

Des défis à relever

Quelle que soit la complexité du calcul, il vaut mieux aller faire un tour à pied et réfléchir à comment les choses vont s'agencer avant de commencer

Alain Connes

Cette thèse a pour périmètre le développement de logiciel d'IHM embarqué critique. Dans ce chapitre, nous présentons d'abord le contexte technique de ce domaine (Section 1.1). Nous identifions ensuite un problème majeur rencontré dans ce domaine (Section 1.2). Enfin, nous présentons un ensemble d'exigences qui, si elles étaient remplies par une approche de développement, permettraient à cette approche de répondre au problème (Section 1.3).

1.1 Contexte

Le domaine du développement de logiciel d'IHM embarqué critique est à l'intersection de deux disciplines bien différentes: le développement de logiciel embarqué critique d'une part, et le développement d'interfaces utilisateurs d'autre part. Ces deux disciplines sont historiquement, culturellement et scientifiquement bien distinctes.

D'un coté, le développement de logiciel embarqué critique privilégie des approches très cadrées, maitrisées et formelles, afin d'obtenir un niveau de confiance extrêmement élevé dans le produit final. Pour cela, un grand nombre de techniques sont mises en oeuvre, qu'il s'agisse de normes comme DO-178 [1], de méthodologies de développement, d'outils de modélisation et de vérification, ou tout simplement de bonnes pratiques basées sur le retour d'experience. Ces approches très cadrées mènent naturellement à des cycles de développement longs. Le produit final est le résultat d'un processus défini à l'avance, suivant un ensemble d'activités bien ordonné, allant de la définition du besoin à la validation du produit final, en passant par sa spécification, son implémentation et sa validation. Le classique "cycle en V" sert souvent de base aux méthodologies de développement de logiciel critique. Parmi les acteurs concernés, nous trouvons entre autres des développeurs, des ingénieurs systèmes, des experts en méthodes formelles.

D'un autre côté, le développement d'IHM privilégie l'expérimentation rapide, le prototypage et la prise en compte rapide des retours d'utilisateurs. Les techniques mises en oeuvre sont moins formelles et plus empiriques, voir parfois quasiment artistiques. La recherche dans ce domaine s'oriente vers l'expérience utilisateur et les questions d'ergonomie plutôt que sur des questions de fiabilité. Cet état d'esprit mène à des cycles de développement plus courts et itératifs, où la solution émerge à la suite de nombreux aller-retours entre les différents acteurs du processus de développement. Parmi les acteurs

concernés, nous trouvons entre autres des développeurs, des utilisateurs finaux, ou des experts en ergonomie, en facteurs humains.

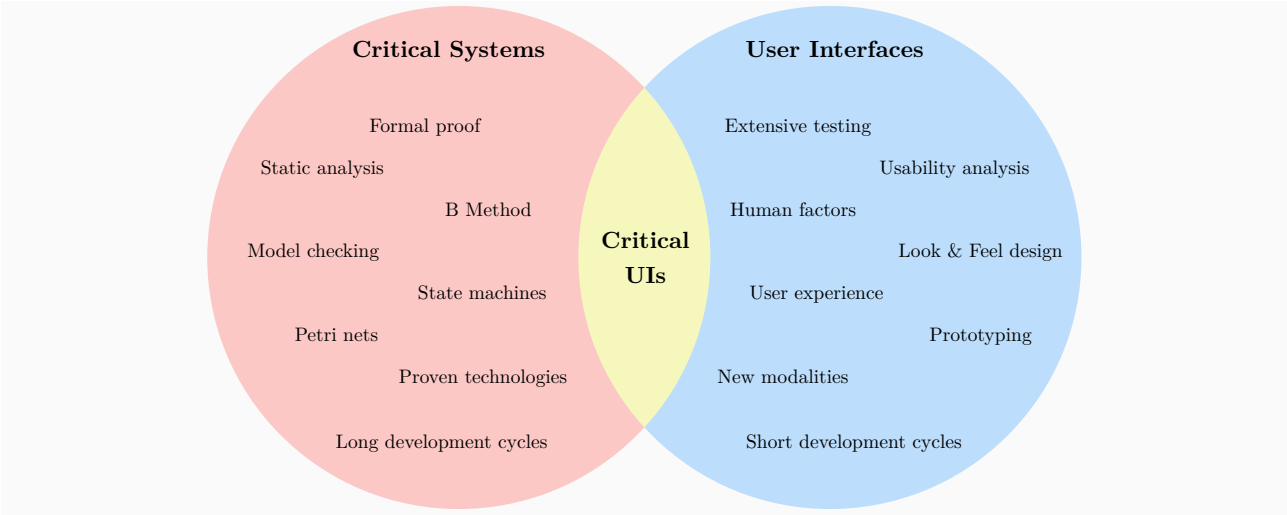


Figure 1.1: Le contexte de cette thèse, à l'intersection de deux disciplines

1.2 Le problème

Nous le voyons, un gouffre sépare les deux disciplines. Cependant, celles-ci sont souvent amenées à collaborer, lorsqu'il s'agit de permettre à un utilisateur (par exemple un pilote d'avion) d'interagir de manière sûre avec un système critique (par exemple les systèmes de contrôle d'un avion). Malheureusement, l'accidentologie aéronautique fait clairement apparaître que le bon fonctionnement des applications interactives critiques est mis en cause de manière récurrente dans une grande partie des accidents. De plus, la multiplication, la complexification et l'automatisation croissante des systèmes critiques a pour effet de faire croître l'importance de la question de l'harmonisation des deux disciplines.

Dans le cadre du développement d'IHMs embarquées critiques, une solution envisageable serait de donner la priorité à l'un des deux aspects, et d'utiliser les méthodes et outils de la discipline prioritaire à la discipline secondaire. On pourrait imaginer appliquer au développement d'IHM les méthodes et outils de développement de logiciel critique, ou vice versa.

L'approche traditionnelle du monde aéronautique a justement été de développer les IHM critiques en donnant la priorité à leur aspect critique plutôt qu'à leur aspect IHM. La justification de ce parti pris est évidente en terme de sécurité. Les méthodologies habituelles de développement de logiciel critique sont donc appliquées au développement de logiciel d'IHM critiques, en essayant d'y intégrer au mieux les spécificités du monde de l'IHM, comme la prise en compte de facteurs humains. L'effet de cette orientation est clair: les IHMs critiques sont en retrait en terme de fonctionnalités lorsqu'on les compare aux IHM non critiques auxquelles le grand public a accès. Par exemple, la Figure 1.2 présente le délai qui sépare l'apparition de technologies grand public et leur application dans le domaine du logiciel critique.

Cet exemple montre que les méthodes et outils utilisés dans ces deux disciplines, prises individuellement, sont la clé de leur succès dans leurs domaines respectifs. C'est grâce à la rigueur mathématique des méthodes de développement de logiciel critique qu'il est aujourd'hui possible d'avoir un taux quasi nul d'accidents aériens dû au logiciel. C'est grâce à la souplesse des méthodes de développement d'IHM qu'il est aujourd'hui possible d'utiliser des systèmes très complexes de manière très intuitive. Il est

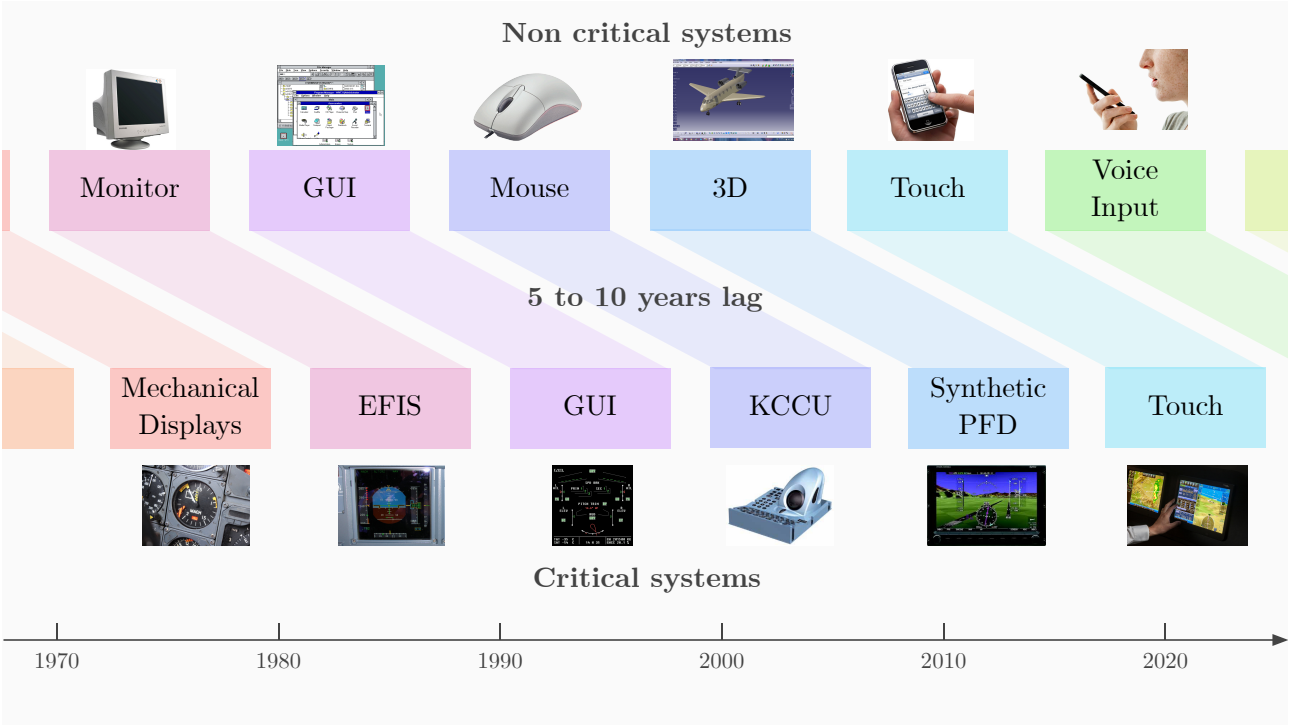


Figure 1.2: Le délai entre l'apparition des technologies et leur application aux systèmes critiques

donc illusoire d'imaginer résoudre complètement le problème de l'harmonisation des deux disciplines en donnant la priorité à l'une d'elles.

1.3 Vers une solution

La méthode à adopter consiste donc à créer des ponts au dessus du gouffre séparant les deux disciplines. Cette thèse a pour objet l'analyse de moyens existants, et la proposition de moyens potentiels permettant de créer ces ponts entre les deux disciplines. L'objectif de ces moyens est de combiner les avantages de chacune des approches, tout en essayant de limiter les inconvénients.

Une analyse du problème nous amène à définir un ensemble de 12 exigences auxquelles une solution au problème devrait répondre:

1. **Formel.** *Offrir une sémantique formelle, claire et non ambigüe*
2. **Collaboratif.** *Permettre le développement collaboratif simultané*
3. **Simple.** *Toutes les parties prenantes doivent être en mesure de comprendre les descriptions créées en utilisant l'approche*
4. **Projection.** *Permettre la projection vers différents langages ayant différentes finalités: prototypage, implémentation, vérification, spécification textuelle*
5. **Traçabilité.** *Permettre une traçabilité entre les différents artefacts*
6. **Composition.** *Permettre de composer des composants existants de manière simple*
7. **Modularité.** *Permettre de définir de nouveaux composants réutilisables*

8. **Abstraction.** *Offrir des moyens de descriptions utilisant les concepts abstraits adaptés au domaine*
9. **Gestion de l'abstraction.** *Offrir des moyens de définir et expliciter différents niveaux d'abstraction*
10. **Prototypage.** *Permettre des cycles de développement courts en utilisant le prototypage*
11. **Verification.** *Permettre la vérification formelle des systèmes décrits en utilisant l'approche*
12. **Souplesse.** *Être souple et ouvert à de potentielles évolutions futures*

Certaines de ces exigences proviennent du domaine du logiciel critique, et d'autres viennent du domaine de l'IHM, enfin, certaines exigences proviennent de l'interaction des deux disciplines. L'ensemble des exigences exprimées ici constitue le fil rouge de cette thèse, et les différentes propositions abordées dans les chapitres suivant se réfèreront à ce cahier des charges.

Chapter 2

État de l'art

Savoir écouter c'est posséder, outre le sien, le cerveau des autres

Léonard de Vinci

2.1 Modèles d'architecture d'IHM

2.2 Modèles d'architecture de réseaux

2.3 Modèles basés sur des interacteurs

2.4 Développement d'IHM à l'aide de langages impératifs

2.5 Langages dédiés (DSLs)

2.6 Langages de balisage

2.7 Bibliothèques basées sur le (DOM)

2.8 Programmation fonctionnelle réactive (FRP)

2.9 Modélisation graphique du comportement des IHM

2.10 Modèles de tâches

2.11 Vérification formelle d'IHM

2.12 Outils commerciaux pour les IHM critiques

Chapter 3

Architecture

Les hérésies jouent un rôle essentiel. Elles tiennent les esprits en état d'alerte
Hubert Reeves

Chapter 4

LIDL

Ce n'est pas en perfectionnant la bougie que l'on a inventé l'électricité
Pierre-Gilles de Gennes

Chapter 5

Implémentation

Il est plus beau d'éclairer que de briller seulement

Thomas d'Aquin

Chapter 6

Application à un cas d'étude

On ne peut montrer le chemin à celui qui ne sait ou aller
Antoine de Saint-Exupéry

Chapter 7

Conclusion

Les portes de l'avenir sont ouvertes à ceux qui savent les pousser

Coluche

Part II

Full version

Chapter 1

Challenges

It always seems impossible until it is done

Nelson Mandela

In this chapter, we present the context of this thesis: safety-critical embedded HMI software development. Then, we proceed to the identification of several problems encountered in this industrial context. Finally, we derive a set of requirements to be fulfilled by potential solutions to the identified problem.

1.1 Context

The main objective of this thesis is to propose a way to design critical interactive software. The first question is: what does "critical interactive software" mean exactly? This expression actually refers to two different fields of engineering: critical software engineering, and interactive software engineering. This observation plants the landscape of this thesis: a field at the confluence of two more general research areas. A great deal of the problematics we tackle come from the particularities of this "niche" research area, and frictions between the two broader fields it is part of.

Figure 1.1 shows these two clashing domains, and the context of this thesis, right in the friction area between them. The following sections details these three areas.

1.1.1 Critical embedded software

Embedded software is software that runs on devices whose primary function is not to be computers, but which often involves interaction with the physical world. Critical systems are systems whose failure would have severe consequences on their users, for example by putting their lives at risk. Critical embedded software combines the characteristics of both: software that runs devices and machines whose failure would be dangerous.

Examples An early example of embedded critical software is the Apollo guidance computer [2], but now such systems are common in many domains. Modern transportation systems such as aircrafts, automobiles, trains or metropolitan transport often depend on critical embedded software. Other

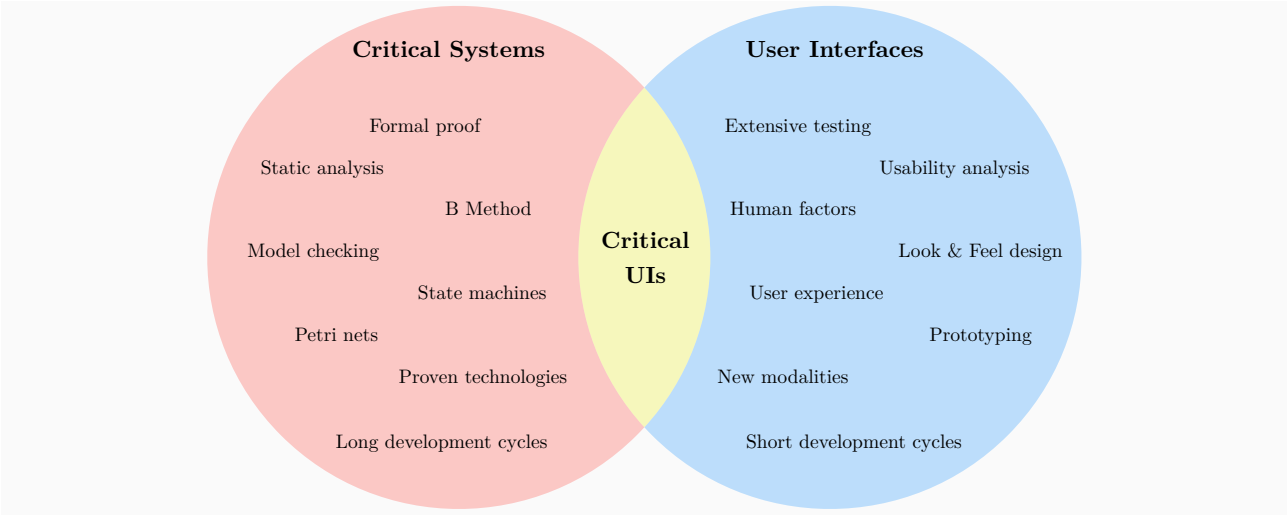


Figure 1.1: The context of this thesis is at the crossroads of broader domains

examples include health equipment such as infusion pumps, monitoring or surgical equipment. Industrial and public equipment also abound with examples: power plants, water treatment facilities and industrial robots of all kinds depend on critical embedded software.

Challenges Critical systems must comply with requirements that are stricter than non-critical systems, while their embedded nature adds additional constraints on the system. Various sources such as [3] and [4] cite several requirements that have to be fulfilled by embedded critical systems:

- Timeliness: Interaction with the physical world means that the time that computations take is important, as the physical world continues evolving at the same speed regardless of speed and delays in computations.
- Concurrency: In the physical world, multiple things can happen simultaneously. Traditional software can pretend that everything happens sequentially and delay the processing of some input. But embedded software must deal with this problem and be able to process several things in parallel if needed.
- Integrity: Critical software must render the service without errors or loss or corruption of data.
- Reliability: Provide failure-free operation over the longest time interval without requiring maintenance
- Availability: Be operational as much of the time as possible, whatever the conditions
- Safety: Operate without catastrophic failure even when unexpected events occur.
- Security: Protect information from a wide range of voluntary and involuntary threats
- Maintainability: Ability to perform corrective and evolutive maintenance of systems
- Performance: Render the service in the best possible elapsed time
- Resilience: Ability to resist unexpected events and control their consequences by returning to the best possible level of operation

Development The development of embedded critical systems is a long and complex process that involves several stakeholders. Particular care has to be taken about the correctness of the program specification and implementation. Figure 1.2 presents a typical development process for such systems. Approaches are often based on the typical V-model: Definition, Specification, Implementation, Verification and Validation. Furthermore, risk assessment loops are often added to the process, in order to reduce the risk of failing to take into account dangerous situations or configurations.

In Figure 1.2, a risk assessment loop is represented on the specification phase: this loop consists in formalising the specification and ensuring that it respects a set of safety properties. Methods used in this phase can be diverse, but we can cite model checking, proof methods, B method... Formalisation can be based on various models such as State machines, Petri nets or many other formalisms. This kind of risk assessment process can also be performed during other phases of the development.

The development cycle for critical embedded systems focuses on delivering a perfectly safe system. As a consequence, the use of proven technologies is often favoured, and long development cycles are the norm.

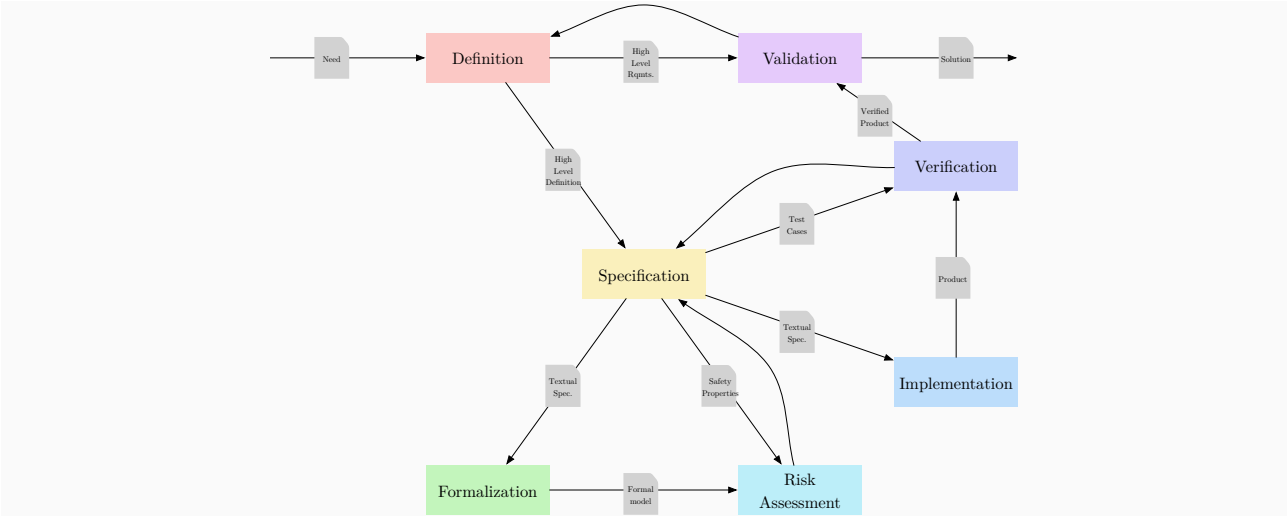


Figure 1.2: A typical development process for critical embedded software

Methods and tools While there is always room for improvement, we can now say that mature engineering techniques, methods, standards and good practices that exist nowadays allow to comply with the constraints of embedded critical software. Thanks to their application, we benefit from the excellent safety records of those systems. Among these different elements, we can find:

- Norms, such as DO-178 [1], which enforce the application of engineering methods and principles.
- Standards, such as ARINC 661 [5], or ARINC 653 which prescribe and detail solutions that are shared by different stakeholders in order to reduce the risk of design error and reduce cost of development.
- Models and languages, that provide appropriate ways to model domains, reducing the possible causes of errors due to accidental complexity.
- Formal methods, such as model checking, proof assistants or static analysis, which are an important way to ensure that critical software meet their requirements [6] [7]. They have been applied with success in different fields such as aerospace [8] or metropolitan transport [9].
- Development methodologies that structure the use of formal methods in a consistent manner, such as B[10], VDM [11]. They have been used successfully in different domains[9].

1.1.2 HMI software

Human-Machine Interface (HMI) software, also known as User Interface (UI) software, is software that runs systems dedicated to the interaction between humans and machines. These kind of systems have become ubiquitous in past few years, and saw a major increase in complexity.

Examples There are numerous examples of HMIs software systems. Consumers are now used to computer and smartphone applications UIs. Many websites can also be considered as UIs that give access to an online service. Transports and industry abounds with examples of HMIs: aircraft cockpits, power plant control rooms, metropolitan supervision rooms, car UIs...

Challenges In order to satisfy users, Human-Computer Interaction (HCI) systems have to verify certain properties, often grouped under the term of usability properties [12]:

- Accessibility: Users should be able to quickly find the information they want, without have to perform too many actions.
- Visibility: Users should be able to visualise the structure of the system and understand its structure and behaviour just by looking at it
- Learnability: Users should be able to accomplish basic tasks the first time they encounter the design
- Efficiency: Once users have learned the design, they should be able to perform tasks quickly
- Memorability: When users return to the design after a period of not using it, they should be able to quickly get used to it again.
- Errors: The design should reduce the rate and gravity of user errors.
- Satisfaction: Users should enjoy using the design

This list demonstrates the particularity of HMI software development: its relation with human users. The human is part of the equation, and this means that feedback from users have to be taken into account in order to design satisfactory products. This makes UI design a specific area or research.

Another difficulty is the increasing diversity of HCI systems. Several qualifiers can be applied to UIs to describe various aspects of them. These qualifiers are not mutually exclusive. Here are some important qualifiers that will be used in this thesis:

- Text-based: An interface where most of the interaction is done by reading text and writing text. For example, early computer terminals and present command-line programs have text-based UIs.
- Graphical User Interface (GUI): An interface where the output to the user is mostly graphical. Most modern computer applications and mobile phones have GUIs.
- Windows, Icons, Menu and Pointer interface (WIMP): A subset of GUIs where the input from the user is done using a pointer (mouse), and which displays windows icons and menus. Most modern mainstream computer Operating Systems (OSs) such as Windows and OS X offer WIMP interfaces, but mobile devices are not WIMP.
- Voice User Interface: Interfaces where some input and/or output is done vocally, either using voice synthesis as an output or voice recognition as an input.
- Touch UI: Interface which contains a device that respond to the touch, such as a touchscreen. Most modern mobile phones have a touch interface.
- Tangible UI: Interface which places a great emphasis on physical interaction with the user or its environment, for example by using several physical objects as an input method.
- Gesture-based: Interface which accepts gestures of the user as an input methods
- Ubiquitous UI: Interface made of several small or hidden computing devices and sensors that communicate in order to make the human machine interaction possible.
- Multimodal UI: Interface which combines varied input and output methods. These methods are called modalities. For example an interface where the user can interact by performing a gesture and saying something at the same time is multimodal

Development Figure 1.3 describes a simplified development process for HMI systems. Due to the challenge expressed above, the opinion of users, or at least other humans, has to be taken into account

as early as possible during the product design phase. This implies phases of evaluations, which becomes almost compulsory for every large systems.

This evaluation can be done early, thanks to prototyping. The prototype can be more or less limited in functionality: some UI prototypes are just mockups or even sketches, but allow to have at least an overall view of what the specified system will look like and how it is organised. These prototypes are evaluated and modified until an acceptable specification is found. When the specification is implemented, the final result is again evaluated. This can cause new iterations if the prototyping phase failed to discover certain issues of the specification, or if the implementation was not able to fully follow the specification.

We see that these design iterations are an important part of the process. This gave rise to techniques such as User Centered Design (UCD), where the user is part of the design process. The increasing complexity of HCI systems makes such techniques become more and more important. In any case, UI development cycles have to be short in order to be able to quickly react to user feedback. I

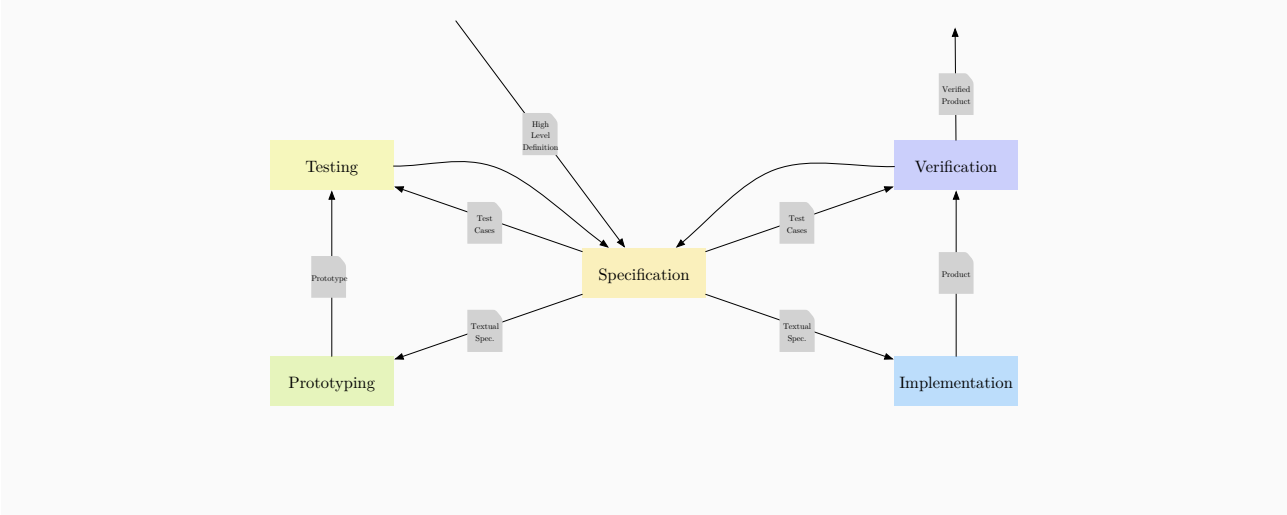


Figure 1.3: A typical development process for HMI software

Methods and tools Various methods and tools are used to support the UI design process: Many tools offer support for prototyping, enabling user feedback early in the design phase. What You See Is What You Get (WYSIWYG) editors allow to create UIs without coding, speeding the development process. Finally, agile methods are a particularly good match for the development of UI, as they enable faster development cycles and more interaction with the end users.

1.1.3 Critical embedded HMI software

Critical embedded HMI software combines the characteristics of both critical embedded software and HMI software. These software systems enable the interaction between Humans and Machines in a context where the loss of this interaction capability could have catastrophic effects.

Examples Critical embedded HCI systems include Human spaceflight vehicles and Aircraft Cockpit Display Systems (CDSs), the UIs of other transportation vehicles such as some trains or self-driving cars. Other examples include medical devices such as infusion pumps, and nuclear power plant control rooms.

Challenges The challenge in the area of critical embedded HMI software design is that critical embedded software design and HMI design have contradictory needs: while the former needs time and a controlled development process, the later needs reactivity and adaptability to user feedback.

The combination and integration of very different methods and tools is also a challenge. How to integrate a specification formalisation activity in a process that also involves frequent modification of the specification due to user feedback ? This combination also involves stakeholders with various cultures, how to make a human factor specialist work productively hand in hand with a model checking specialist ?

Development process Figure 1.4 shows a typical development process of critical, embedded HMI as a combination of the two development processes described in Subsection 1.1.1 and Subsection 1.1.3. Specifications of critical HCI systems must at least include in-depth descriptions of the following aspects of UIs:

- Structure: Static aspects of the HMI as a composition of containers and interactive sub systems.
- Behaviour: Dynamic aspects of the HMI and its interaction with both the machine and the human.
- Appearance: Aesthetic aspect of the HMI as seen by the human user.

All these aspects of the specification have an impact on all subsequent activities of the development process, (Testing, Risk assessment, Implementation...) In the end, we note without surprise that the very challenging sets of constraints coming from two different domains combines at their intersection to form constraints that were impossible to solve until recently. This difficulty results in a very heavy development process, which will hopefully become easier as techniques and methods improves.

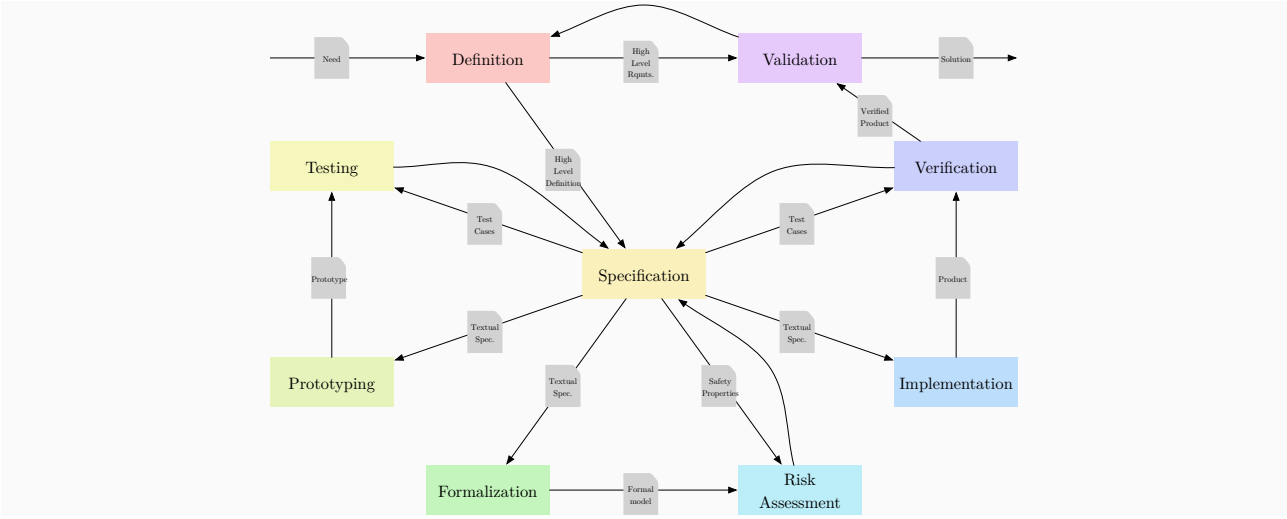


Figure 1.4: A typical development process for critical embedded HMI software

1.2 The problem

In the previous section, we described the field of critical interactive systems. We noted that this field of engineering poses numerous technical issues for designers. In this section, we will explore a bit deeper the challenges encountered by the various stakeholders in this domain.

1.2.1 New HMI techniques time-to-market

The field of HCI is really dynamic at the moment. This is mostly due to the plunging prices of sophisticated hardware technologies that pull the market along two axes.

The first axis is the increase of computational power, according to Moore's law, that states that computational power of state of the art computers is growing exponentially, doubling every 18 months. This first force implies that, as the bandwidth of computers becomes bigger, the bandwidth of human-computer interaction has to increase as well, so new techniques have to be developed in order to cope with this. How useful is an embedded supercomputer without a way to interact with it ?

The second axis is the recent shift in price of hardware and software that enable new interaction modalities, such as pressure sensitive touch screens, voice input, 3D motion tracking, eye tracking, virtual reality... This generates a lot of attention and gave birth to many new interaction modalities that were marginal a few years ago, and are now becoming commonplace.

These two forces brought a whole lot of new ideas in the field of HCI. However, these new technologies take a long time to see application in the field of critical HCI, as presented in Figure 1.5. This is due to the fact that experience is the key to gather enough confidence in HMI systems and develop appropriate development methods. But experience takes time.

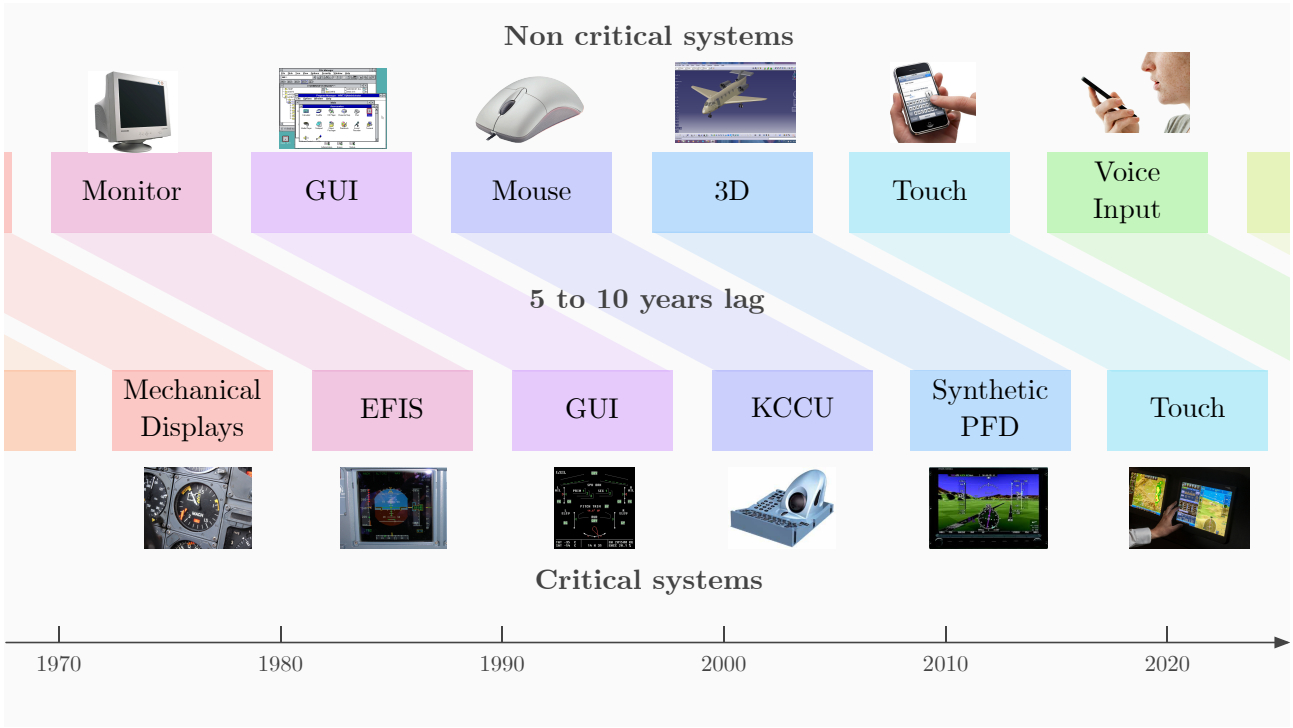


Figure 1.5: The delay before application of new interaction techniques in the field of critical systems

1.2.2 Complexity of the design process

Figure 1.4 shows a typical design process for critical HMI, centered around the specification phase. The classic V-model of critical systems design is embedded in this graph, outlined in purple. Along to this V-model are some activities known as early verification and validation. These activities help ensure that the specification is adequate in regards to the requirements and performs as expected, before the actual implementation is finished. All these artefacts and activities are in use for a reason, they all have a role in the process, either in order to reduce its costs or in order to enhance its quality.

However, the diversity of these artefacts is a big obstacle for the stakeholders. In particular, translation and traceability of these different artefacts into each other represent a large burden during the design of critical HMIs. As an example, consider the following scenario:

- A problem is identified by a human factors specialist during early tests on a prototype of an HMI. This problem is expressed informally using natural language by a human factor specialist.
- A modification of the prototype code is performed in order to quickly try a solution to the problem. This modification is made on code written in the prototyping language (e.g. Java).
- If the modification solves the problem, then it can make it to the specification. A rule would be added in the specification in order to deal with the problem. This modification is made on a specification language (e.g. textual specification), in accordance with code written in the prototyping language.
- Then, in order to ensure that this modification does not create new unexpected problems, it should be formalised and modelled in the language used for model checking (e.g. NuSMV)
- If this modification does not create new unexpected problems, and if the implementation of the system has already started, then this modification should also be applied to the implementation, using the implementation language (e.g. C).

In the end, extra care should be taken to ensure consistency between artefacts written in 4 different languages (e.g. Java, textual specification, NuSMV and C). This implies coordination of 5 different stakeholders (Human factors, Prototype implementer, Specifier, Model checking specialist, Production code implementer). This burden is necessary to ensure the quality of the final product, but causes high costs and delays.

1.2.3 Difficulty of specification

If we focus on the specification phase of the design process explained in Figure **??**, we notice that specification of critical HMIs is a complex task. This process takes time and money. Overall, specification documents for relatively simple critical HMIs often span several hundreds pages, spread between DFS, Look And Feel (LAF), Functional Requirements Document (FRD) and High-level Requirements Document (HRD) of the HMI system. In the absence of other ways to ensure correctness of a specification, these documents are absolutely necessary to ensure a satisfactory specification of these critical systems. Due to the lack of common mathematical models for this domain, most of these specifications are written in informal, natural language.

On the other hand, critical HMIs are still HMIs systems. As such, they benefit from practices such as prototyping, which is in practice almost the only way to specify satisfying HMIs. Specifying an HMIs without prototyping would be like painting on a canvas without looking at the result until it is finished: it can give interesting results, but it is definitely not the easiest way to end up with a satisfactory final result.

From theses points comes a tension: designers have to produce large amounts of detailed specification documents, but at the same time they have to iterate on the design by prototyping. Any modification on the prototype cascades to many modifications in the specification corpus, and all these artefacts have to be kept synchronised and traced to each other.

In the end, critical HMI specification implies many different stakeholders: designers, coders, human factors specialists, system engineers, end users... These stakeholders do not have the same cultures, and communication between them is sometimes perfectible. This is another reason why the specification process is long and complicated.

1.2.4 Difficulty of implementation

Once a specification of the HMIs is done, comes the implementation phase. This phase is also very difficult for implementers for numerous reasons. The first cause is linked with the specification problem. Since critical HMIs are often specified in an informal way, their specification are often ambiguous. The burden and responsibility of picking an appropriate interpretation of the specification, or to come back and ask for more precision is left to the implementer.

Then, even when the meaning of the specification is perfectly clarified and unambiguous, the task of translating it into code is difficult. Standards such as American Radio Incorporated standard 661 (ARINC661) [5] do help in this task, by specifying standard design patterns and architectural elements. Still, interaction patterns described in plain english do not translate trivially at all into code. As an example, consider the specification rules of Listing 1.1.

```
1 • For each Table element, in the 2nd column, when the Inbox Tab Toggle Button is
  ↪ selected, the associated Reference Tab Toggle Button shall be deselected (and
  ↪ reciprocally).
2
3 • If SystemA receives the configuration of the selected SystemB in less than 60s
  ↪ after the display of ScreenB, the HMI:
4   - Shall respectively display the Attributes V, W, X, Y, Z, of SystemB in cells
  ↪ (b), (f), (g) and (h) of the 3rd column of the Table
5   - Shall fill the comboboxes Inbox Tab Combo Box and Reference Tab Combo Box with
  ↪ information contained in the SystemC that are compatible with the selected
  ↪ SystemB.
```

Listing 1.1: An example snippet that can be encountered in an industrial context DFS

Obviously, the code implementing these rules will have very few in common with the text of these rules. This problem is not specific to HMI systems, but it is exacerbated in this field. In other domains of critical systems, the general structure of the specification matches a bit more with the structure of the final code, and the traceability between specification and code is more obvious.For example, in the area of control system, tools such as Scade [13] even allows to partially generate code from specifications that are checked[14]. In the field of interactive systems, the static aspects of user interfaces can often be generated automatically, many tools such as Scade Display[15], Apple storyboards, Swing designer and dozens of others allow this, but the behavioural aspects of interfaces often have to be coded by hand, an error-prone process.

1.2.5 Difficulty of verification

Verification of critical systems is achieved by two complimentary means: testing, and application of formal methods. There are various kinds of testing and various kinds of formal methods.

Testing consists in the execution of sets of various scenarios on parts of the system to verify. Testing individual components of a system independently from each other is called unit testing. This allows to spot errors of implementation. Testing bigger parts of the system made of several components, or the whole system at a time is called integration testing, it allows to spot more complex errors caused by inappropriate and unplanned interactions of components. For most critical systems, testing can be performed automatically, and only the task of specifying test cases or scenarios is left to humans. In fact, tools such as [16] automates this task too, by automatically generating test cases.

In the field of HCI, most testing is still made by humans, a labour-intensive and capital-intensive task, which is prone to human errors. This is caused by the fact that in essence, HCI involves humans, so testing them without humans is a bit more complicated, because it involves modelling human users to some degree. However some tools exists [17] to automatically test HCI systems according to manually defined scenarios. Works such as [18] also proved the possibility of using human-related heuristics to automatically generate test cases.

Formal methods consists in application of mathematical tools to mathematical models of systems in order to raise the level of insurance in the system. These different tools are model checking, theorem proving, static analysis... Such tools have been applied successfully to actual non-HCI critical systems [8], [10], [13], [19]. Verification of small HCI systems has been successful [20]. However formal methods still fail to verify industrial size HCI systems, because of their complexity, which causes problems such as state explosion in model checkers.

It seems that abstraction of the problem is necessary, but the domain of HMI lack appropriate abstractions. No industrial widely used method allows to abstract HMIs the same way as some other domains allow. Approaches to HCI verification often fall in one or the other of the sides. Either they allow to describe systems in much detail, hampering scalability and preventing application to large scale systems, or they describe systems in a more abstract way, missing some small grained details that can prove to be very important, which decrease the value of the analysis. This is due to the structure of the problem of HCI, where small details can prove to be very important.

1.2.6 Reusability difficulties

Because of the difficulties inherent to the task, developing critical HCI takes a long time. In Section 1.2.1, we explained how this is a problem, from the user point of view: Users benefit of new technologies long after they are used in the non-critical world. But there is another problem, from the developers point of view. Product life cycles in the domain of critical systems are so long that the evolution of methods and tools is almost faster than the evolution of the product itself. As a consequence, when comes the time of developing a new product, the methods and tools used back in the time when the previous product generation was developed have become mostly obsolete. As an example, the duration of programs in the field of aerospace are of the order of the decade. But a decade is a very long timeframe in the world of software design tools. As a consequence, reusability is difficult from program to program.

Again, this is not so much a problem in fields which have common, stable abstractions. For example, in the field of control systems, the reusability of previously developed systems is a bit easier, because whatever the tools, the mathematical abstractions remains similar. In the field of HCI however, no common widely accepted abstraction that explains most of the domain issues exist. Methods and tools are based on various abstractions, depending on the time period they were developed in. This is an obstacle to reusability, because new programs cannot reuse what was done during previous programs , since they are not based on the same paradigms.

This problem is probably due to the fact that HCI is a relatively recent field of research, and its theoretical bases did not have time to settle yet. The field of HCI is still subject to trends and fads, several approaches are still competing, and this hampers reusability, sometimes forcing developers to re-invent the wheel in several languages or paradigms.

1.2.7 Integrating the human factor into the process

Human factors are an important aspect when it comes to critical HCI, and they are actively studied and analysed during development of such systems[21]. The world of human factors is a rich field of research, with various approaches and some common notions. On the other hand, the field of critical embedded software is also rich a varied. But these two fields do not have the same culture. This makes sense, as describing humans and their behaviour is very different to describing computers and technical systems. But in the realm of critical HCI development, stakeholder coming from these two different backgrounds have to meet and work together.

From this comes a mismatch which is difficult to overcome. No common language allows to describe these two aspects. Many approaches have proven that the interaction between these two fields is a fruitful research direction [22] [23] [24], but no common language exists to allow to integrate these two aspects seamlessly.

1.3 Requirements for a solution

Now that we stated a set of problems in the field of critical HCI development, we can attempt to find a set of requirements for an attempt to solve them. Fulfilment of all these requirements would help the design process of critical HCI systems by providing ways to tackle the challenges expressed in Section 1.2.

1.3.1 Formal semantics

The goal of this thesis is to ease the design of critical interactive software. We explained in section 1.2.4 that a common problem in this domain is the lack of formalisation of specifications, that leads to ambiguity for the implementers. This is not acceptable in the field of critical systems, because ambiguous specifications can lead to unexpected and dangerous behaviour. All successful approaches in the field of critical system provide formal semantics in order to remove the possibility of misunderstanding between stakeholders and ambiguous behaviour. As a consequence, we can express a first requirement for our solution. This requirement comes from the domain of critical software systems.

Requirement 1 (Formal)**.** *The approach should provide clear and unambiguous formal semantics of human computer interaction*

1.3.2 Team work and collaboration

As expressed in Section 1.2.2, critical HMI design processes implies many stakeholders and artefacts. All these elements have to be linked and synchronised. This represents a large burden which does not directly add value to the final product. This gives two requirements. The first requirement is about synchronisation between the stakeholders. It states that stakeholders should be able to collaborate as a team using this approach. It is a typical requirement for software development tools.

Requirement 2 (Collaborative)**.** *The approach should allow collaborative development*

Collaboration of various stakeholders with various technical skills implies that the approach should provide a common language for them to collaborate. Team members should all be able to understand the artefacts described in this language, in order to be able to discuss issues.

Requirement 3 (Simple)**.** *Descriptions of systems by the mean of the approach should be understandable by all stakeholders*

1.3.3 Projection and Traceability

We stated in the previous section that the approach should act as a link between various stakeholders. As a consequence, it should allow to generate documents for these various stakeholders. These artefacts are as varied as the stakeholders, but the approach should allow to generate at least a skeleton of these various documents, based on the general structure of the specification.

Requirement 4 (Projection)**.** *Descriptions of systems by the mean of the approach should be suitable to projections into other languages with different purposes: prototyping code, implementation code, verification code, textual specification...*

Projecting into various languages without providing traceability between them is useless. Traceability enables to link different aspects on a common frame, it is the glue that holds the various parts together. The approach should allow this.

Requirement 5 (Traceability)**.** *The approach should allow to trace the links between different artefacts*

1.3.4 Modularity

An obvious and well known way to reduce development times and costs is to reuse existing components. All successful approaches in the field of software development encourage reuse by providing ways to design modular systems. An approach that does not encourage reusability is bound for failure. Modularity comes in two parts. The first part is being able to compose elements in order to create complex systems, and the other part is being able to create new reusable elements.

About the first aspect (composition of elements), we can say that allowing composition of elements is not everything. This composition process has to be simple. Many approaches allow to compose elements, but this composition is rarely elegant or easy to use. As we will see in Chapter 2, many approaches require a certain amount of *glue* between components, and this glue ends up being the most complex part of the overall system, defeating the whole purpose of composition.

Requirement 6 (Composition)**.** *The approach should allow to compose existing components in a simple way.*

The second aspect is being able to reuse complex systems by making them modular. Objects described by composing basic objects should be able to be reused as if they were themselves basic objects.

Requirement 7 (Modularity)**.** *The approach should allow to define new reusable components as a composition of other components.*

1.3.5 Abstraction

Technologies come and go, but the basic problems stay the same. Many approaches give really good results when applied to the particular environment they were designed for. But as the technology ecosystem evolve over the years [25], the underlying technologies often die out, and new approaches have to be devised to tackle the same base problems. In order to be useful on the longer term, an approach to enhance UI design would ideally not be linked with any particular toolkit or implementation, but would instead allow to abstract away the implementation specific aspects.

Requirement 8 (Abstraction)**.** *The approach should allow to describe abstract concepts relevant to the field of interactive systems independently of any particular implementation.*

Abstraction is often structured in a complex way, where low-level abstractions have to fit in the model of higher level abstractions, and new problems give birth to new abstractions. In order to be tractable, an approach would have to enable to manage these different abstraction levels.

Requirement 9 (Manage)**.** *The approach should allow to explicit and manage different abstraction layers.*

1.3.6 User Centered Design

This requirement comes from the field of HCI. As explained in Section **??**, UI designers have to iterate quickly on the design of interactive systems in order to adapt the system to the actual needs of the end users. This is a need that is often fulfilled using a range of techniques known as Agile methods. These kind of approaches are necessary to ensure satisfaction of end-users on complex UIs. A UI cannot be designed using a pure V-model, because this would imply knowing *exactly* what end users want, which has proven impossible without some kind of exchange with them.

Requirement 10 (Prototyping)**.** *The approach should allow short design iteration cycles and enable prototyping.*

1.3.7 Support formal verification

Humans engineers, as good as they can be, can not think of every possible failure mode of complex systems, and will fail at it sooner or later. Therefore automatic methods of verification are used in the domain of critical systems as explained in **??**. The approach should allow such techniques to be applied, at least to verify the software aspects.

Requirement 11 (Verification)**.** *The approach should allow to perform formal verification on the systems*

1.3.8 Flexibility

As explained in [25], evolution of software ecosystems has many similarities with evolution of natural ecosystems. Species that can adapt to the environment are the one that survive in the longer term. This is also true for software projects. No approach can be perfect perfect, and even if it was possible, evolution of the technological ecosystem would make it obsolete unless it can adapt to new issues. In order not to fall in the problem expressed in Section 1.2.6, the approach should be able to evolve with its time, while still keeping its roots. To fulfil this requirement, approaches should be based on a set of well-founded basic principles, and allow to adapt new techniques and tools around this founding principles.

Requirement 12 (Flexibility)**.** *The approach should be flexible and allow evolution*

1.3.9 Synopsis of requirements

Table 1.1 presents a synopsis of the requirements we listed. This set of requirements is what we expect from an ideal approach that would solve many problems encountered when developing critical embedded UIs. To our knowledge, no existing approach fulfils all these requirements simultaneously, but some existing solutions solve several of these requirements.

Requirement	Definition
1 Formal	Provide clear and unambiguous formal semantics
2 Collaborative	Allow collaborative development
3 Simple	Be understandable by all stakeholders
4 Projection	Allow projections into other languages
5 Traceability	Allow to trace links between different artefacts
6 Composition	Allow to compose existing components
7 Modularity	Allow to define new custom components
8 Abstraction	Allow to describe abstract concepts relevant to the field of interactive systems
9 Manage	Allow to explicit and manage different abstraction layers.
10 Prototyping	Allow short design iteration cycles and enable prototyping.
11 Verification	Allow to perform formal verification on the systems
12 Flexibility	Be flexible and allow evolution

Table 1.1: Identified requirements

Chapter 2

State of the art

If knowledge can create problems, it is not through ignorance that we can solve them

Isaac Asimov

In this chapter, we present some existing approaches relevant to the field of embedded critical human-machine interfaces development. As stated in Chapter 1, this domain is the intersection of several broader domains which are extremely different, but the set of requirements derived in Chapter 1 is what binds these aspects together. We will present a large spectrum of approaches in this chapter. In order to keep in mind our main goal of finding solutions for the requirements expressed in Section 1.3, we link each presented approach with the requirements it tackles, and to the requirements it does not solve. The various approaches have different characteristics important to keep in mind in order to understand their positioning. These important characteristics include:

- The spanned domains: This can be any combination of critical systems, embedded software and human-machine interaction. Some approaches belong to a particular field, while some other try to tackle the complexity that arise when combining two different domains, such as critical human-machine interaction. For example, the fact that an approach can be applied to critical systems is important.
- The position in the development cycle. Some approaches are relevant in early phases of the development cycle, while others are relevant during implementation, validation, or other phases of the development cycle.
- The maturity level of each approach. Some approaches are experimental and have only been applied to research cases, while some approaches have been in industrial use for a long time already. This aspect is known as Technology Readiness Level (TRL).

The rest of this chapter is sorted by grouping approaches that present similarities on these aspects. Each of the sections of this chapter present a set of techniques that are relatively similar in terms of application domain, maturity level and position in the development cycle. Figure 2.1 presents an overview of the following sections, by positioning them in a space whose dimensions are the membership to the field of HCI, membership to the field of critical systems, and position in the development cycle. As explained in Figure 2.1, we expect to cover the whole spectrum of approaches, but still, many more approaches exist and will not be addressed in this chapter. We merely chose to present approaches that had some impact on the developments presented in the next chapters of this thesis.

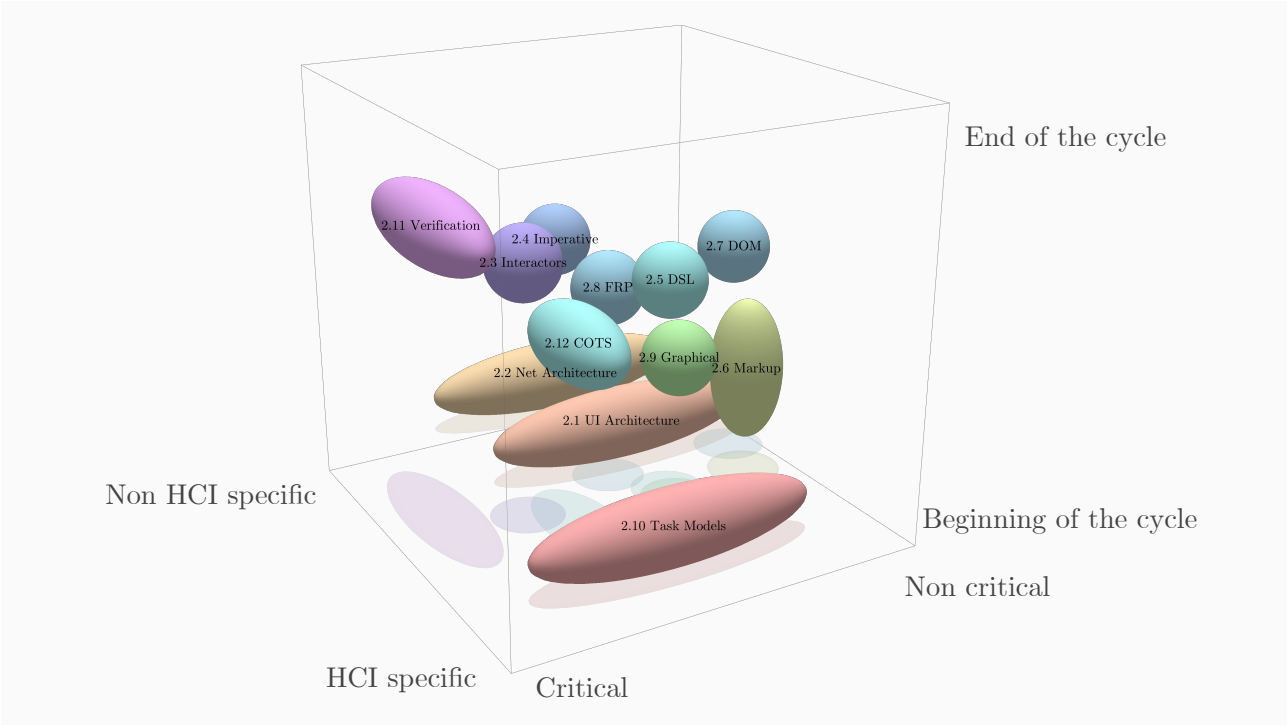


Figure 2.1: Positioning of the groups of approaches presented, along with their section number.

2.1 Human-machine interface architecture

In this section, we will study some existing models of software architecture for HMIs. But first, let us define this notion. According to [26], software architecture is "a set of structures which comprise software components, the externally visible properties of these components and the relationships among them". According to [27], software architecture is "Fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution".

We see that these two definitions use the notion of respectively components and elements. This is not surprising, because when designing complex systems, this problem of composition of multiple components often becomes the main difficulty to tackle, hence our requirement 6, which states that the approaches we need should allow to compose reusable components. This is why, in this section, we present approaches with the perspective of composition of systems. Historically, some approaches tackled other concerns, and did not take into account the notion of composition of components. However, most recent approaches put a central emphasis on how they deal with complex compound systems.

2.1.1 Seeheim model

Proposed in 1983 during a seminar in Seeheim, Germany, this is one of the first models of architecture for UIs. This model describes an architecture made of components which play well-defined roles in the interaction with the user. These roles are based on a metaphor which relates human-machine interaction to a dialog between a human and a machine, using a language whose words are basic actions such as mouse clicks and keystrokes.

- Presentation: This component manages the input and output of the UI. It presents output data to

the user (graphics on screen) and manages input devices (mouse and keyboard). This component represents the lexical aspects of the human-machine dialog.
- Dialog control: This component makes a link between the presentation component and the application interface. It provides a translation between the sequence of user interactions and actions that are significant at the application level. This component represents the syntactical aspects of the human-machine dialog
- Application interface: This component transforms actions into data that can be exploited by the application. It is a part of the application that defines the semantics of the the human-machine dialog.
- Switch: This component is less important but it allows to provide rapid feedback to the user on actions that do not have semantic value for the application, but that have lexical or syntactic signification. For example, a movement of the mouse pointer is rarely significant for the application, but it should be visible to the user: this is what this feedback loop is for.

In Figure 2.2, we present the Seeheim model as it was presented. We see that there are two lines of data flowing through the three main components, one from the human to the machine, and the other in the opposite direction. We see the switch which acts as a shortcut to provide fast feedback. However, the inner workings of elements are not specified. This is why this model is often described as a global or centralised model [28]. Another limitation of the Seeheim model is its emphasis on the dialogue metaphore, which implies that human-machine interaction can be represented as a linear sequence of actions. This does not fit well with multimodality, where several different dialogues can happen in parallel.

Overall, the Seeheim model helps fulfiling Requirement 8 (Abstraction) and Requirement 9 (Manage), however, his weak points are Requirement 7 (Modularity) and Requirement 12 (Flexibility).

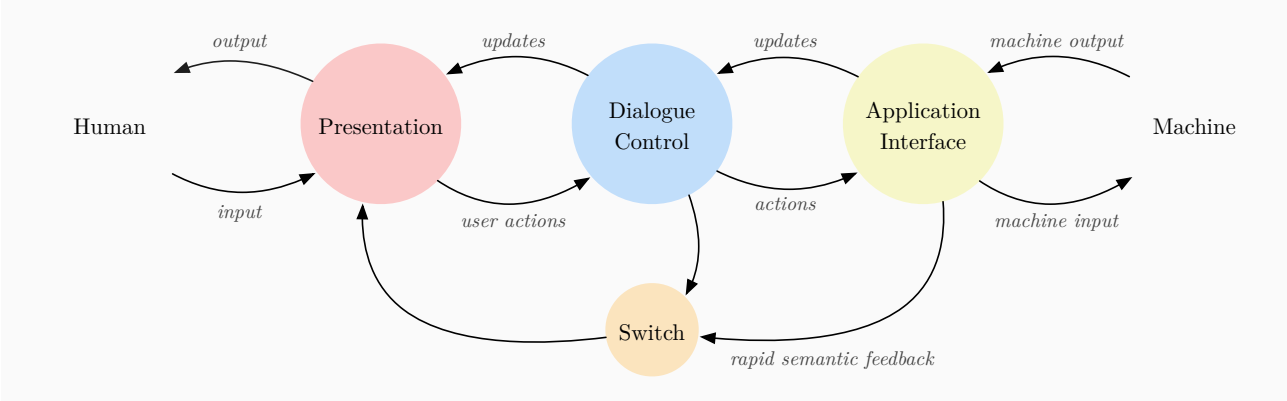


Figure 2.2: The Seeheim Model

2.1.2 Arch model

Proposed in 1991 as an evolution of the Seeheim model in [29], the Arch model adds two intermediate components in the main chain of the Seeheim model. One of these components acts as adapters which enable more flexibility during the evolution of the system, while the other expresses the existence of a functional core for the application. The five components of the Arch model are:

- Interaction Toolkit Component: This is equivalent to the Presentation Component of the Seeheim model. However the name Toolkit emphasises the fact that this component is rarely designed with the applications but merely often part of libraries or OSs, and hence, not customisable by the application programmer.

- The Logical Presentation Component: This component is an adapter between the toolkit and the dialogue component, that is here to deal with discrepancies between an imposed toolkit component and a dialogue component that has its own constraints.
- The Dialogue Component: His role in the arch model is similar to the role of the Dialogue control component in the Seeheim model.
- The Functional Core adapter: This component has a role similar to the Application Interface of the Seeheim model.
- The Functional Core: This component represents the domain-specific structure and behaviour of the application, it is independent of any UI implementation. This component was out of the scope of the Seeheim model, on the "machine" side.

Figure 2.3 shows a depiction of the arch model. Like the Seeheim model, this model is global[28], it does not describe the composition of its elements (Requirement 6 (Composition)), but gives an overall structure which is a good starting point for interactive applications (Requirement 8 (Abstraction)). The main advantage of the arch model as compared to Seeheim is the improvement of the decoupling between components thanks to adapter layers. This helps solving Requirement 9 (Manage).

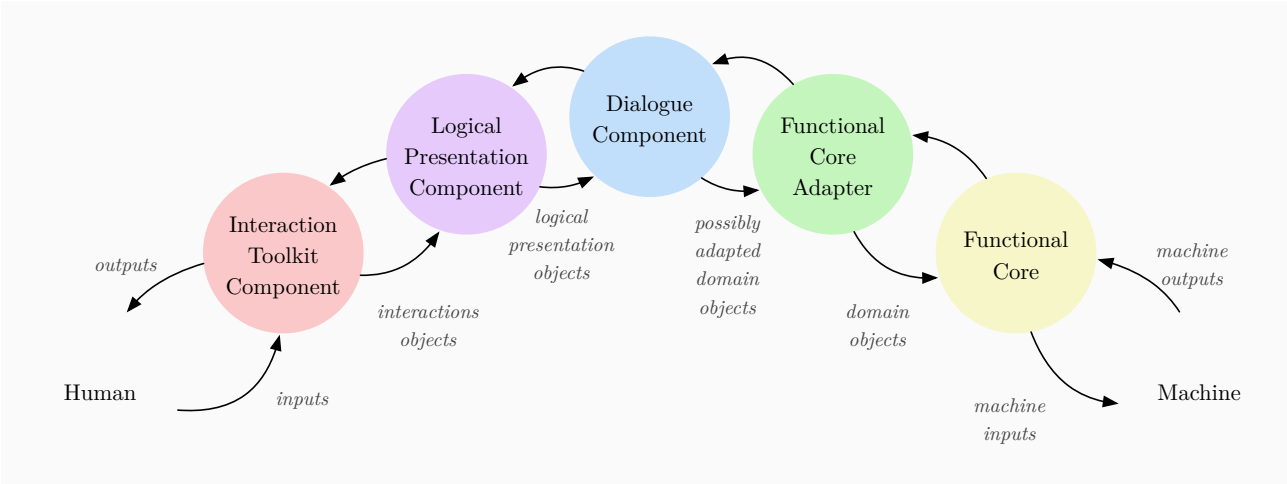


Figure 2.3: The Arch Model

2.1.3 Traditional Model View Controller

Created in 1979 [30], the Model View Controller (MVC) design pattern is one of the first and most successful models of UI architecture. Due to its popularity, it gave birth to numerous other approaches and variations, some of which are explained later in this section. This subsection presents the approach as it was described initially.

As explained in [31], the main idea of MVC is to segregate three different aspects of UIs in order to structure the architecture. These three aspects are :

- Model: The implementation of the application's central structure, that represents the business data and logic. It is independent from the actual UI.
- Views: They deal with the output of the system. At the time, this output was almost exclusively graphical. They request data from their model, and display the data. They contain not only the components needed for displaying but can also contain subviews and be contained within superviews, in a hierarchical way that complies with our requirement 6.
- Controllers: They deal with the input of the system. Controllers contain the interface with the input devices (keyboard, pointing device, time). Controllers are paired with views.

There can be multiple views and controllers, which go together in pairs. MVC relies on the notion of multiple consumers having their needs met from a single shared model object. Each view may be thought of as being closely associated with a controller, each having exactly one model, but a model may have many view/controller pairs. Figure 2.4 gives an overview of a MVC system with one main view made of 2 subviews, their matching controllers and a model. We see that the original MVC separates input techniques (controllers) from output (views), which represent important aspects of UIs. We can say that MVC did help represent UIs in a more abstract way (requirement 8). MVC was one of the first models to express the notion of composition of reusable UI components (requirement 6), but the separation of input and output has its limits and was subject to alterations in subsequent MVC inspired approaches. On this particular point, we can argue that the MVC pattern somehow allowed some flexibility(requirement 12) since it was later adapted and reused in other models.

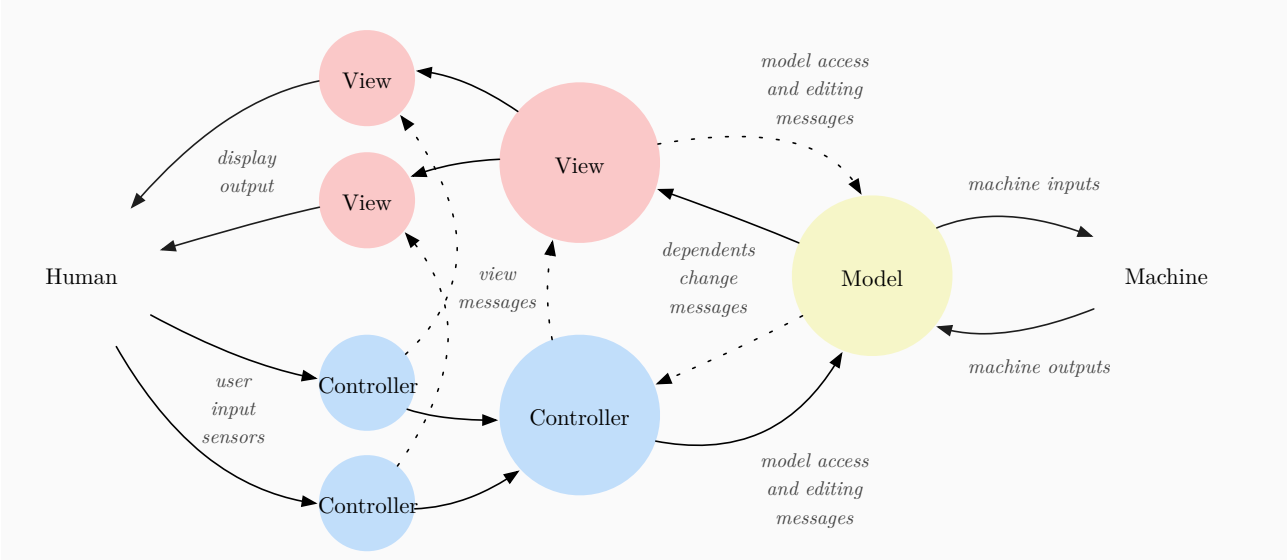


Figure 2.4: The MVC Model as presented in [31]

2.1.4 Presentation Abstraction Control

First presented in 1987 in [32], Presentation Abstraction Control (PAC) is one of the first models to expressly take into account the notion of modularity, the fact that practical UIs are described as a composition of modular components. As a consequence, the different abstraction layers of the human-machine dialogue are not monolithical as in the Seeheim and Arch model, but distributed over a hierarchy of components. These components are each made of three parts:

- Presentation: The concrete syntax of the application, what the user perceives and acts on. For example, in a component that displays a pie chart, the presentation is the actual representation of the chart as a set of coloured shapes and labels.
- Abstraction: The semantics of the application, a more abstract representation that is not dependant on the presentation part. For example, the abstract part of the pie chart component is a list of values.
- Control: The control maintains the consistency between the presentation part and the abstraction part. In the pie chart example, the control part maps elements of the list of values represented in the Abstraction to relative sizes of pie sectors in the graphics represented in the Presentation.

When several PAC components are composed, their Control parts are responsible for the communication between them [33]. As compared to the MVC approach, the Abstraction represents the Model of MVC, the Presentation contains both the View and Controller of MVC, and the Control expresses

dependencies and relationships between the Presentation and Abstraction. PAC also emphasises more on the notion of composition of components, which leads to a Model which is distributed over the hierarchy of components instead of being centralised as in MVC. PAC is one of the first approaches to cleanly solve the problems of composition (requirement 6) and modularity (requirement 7), which are points where it shines.

Several variants of MVC such as Hierarchical MVC are in fact much closer to PAC. The main advantage of PAC is that it describes a modular system which scales well to complex UIs thanks to it abstraction capability (Requirement 8 (Abstraction) and Requirement 9 (Manage)) and offers a great deal of reusability (Requirement 7 (Modularity)).

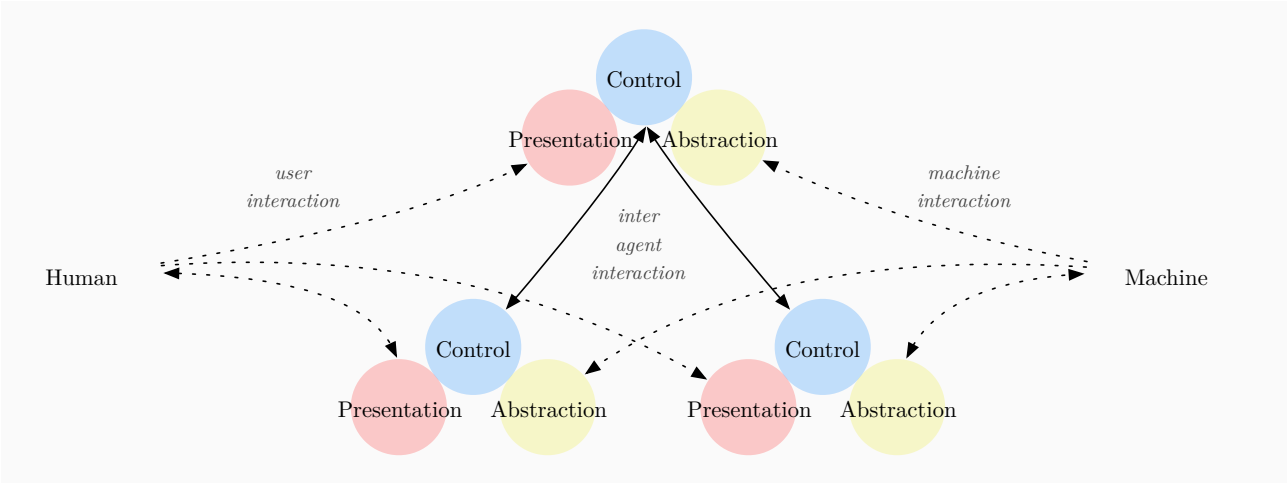


Figure 2.5: The Presentation Abstraction Control Model

2.1.5 PAC-Amodeus

The PAC-Amodeus model[34] is depicted in Figure 2.6. It uses the Arch model as the foundation for the functional partitioning of an interactive system and populates the Dialogue Component with PAC agents. This makes it a really detailed and realistic model that is prone to actual implementation. It was used to develop actual applications [35] while proving its ability to enhance code reuse (Requirement 7 (Modularity)) and hence reduce costs.

2.1.6 Model View Presenter

The Model View Presenter (MVP) architecture as presented in [36], is similar to the PAC architecture. It describes systems made of triplets of components which are similar to PAC components. Models, Views and Presenters respectively stand for Abstractions, Presentation and Control.

As in PAC, MVP prescribes that the machine should interact with components through their Model part (Abstraction part in PAC). The difference between MVP and PAC is that while PAC focuses on describing how triplets can be composed in order to form a complex system made of several PAC triplets whose Control parts communicate, MVP focuses on describing the communication that happens *inside* the triplets. Critically, MVP does not emphasise the notion of combination of several triplets (Requirement 6 (Composition)) , but tends to present applications as a single triplet and emphasise the communication that can happen inside this triplet. MVP describes the interactions between the three components of the triplets, and explains that reusability (Requirement 7 (Modularity)) can be enhanced by adding an interface between them. An interface separates the View from

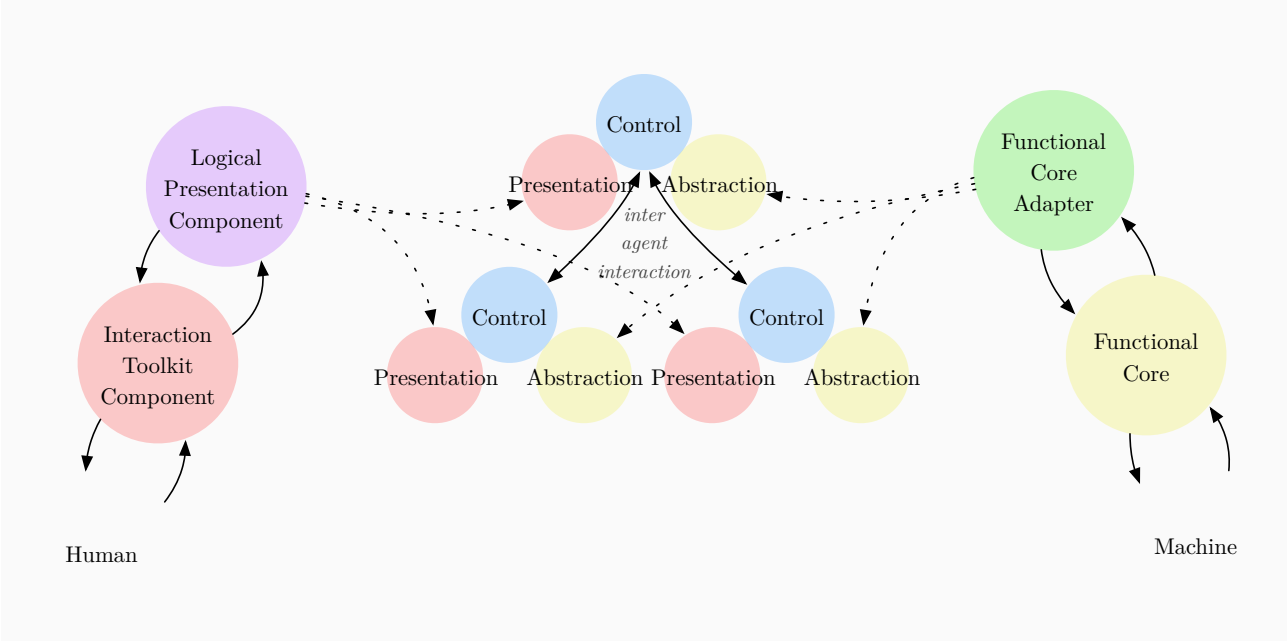


Figure 2.6: The PAC-Amodeus Model

the Presenter, and an adapter separates the Model from the Presenter. These components added to separate components are not without similarity with the components that were added to the Seeheim model (Section 2.1.1) to make the Arch Model (Section 2.1.2).

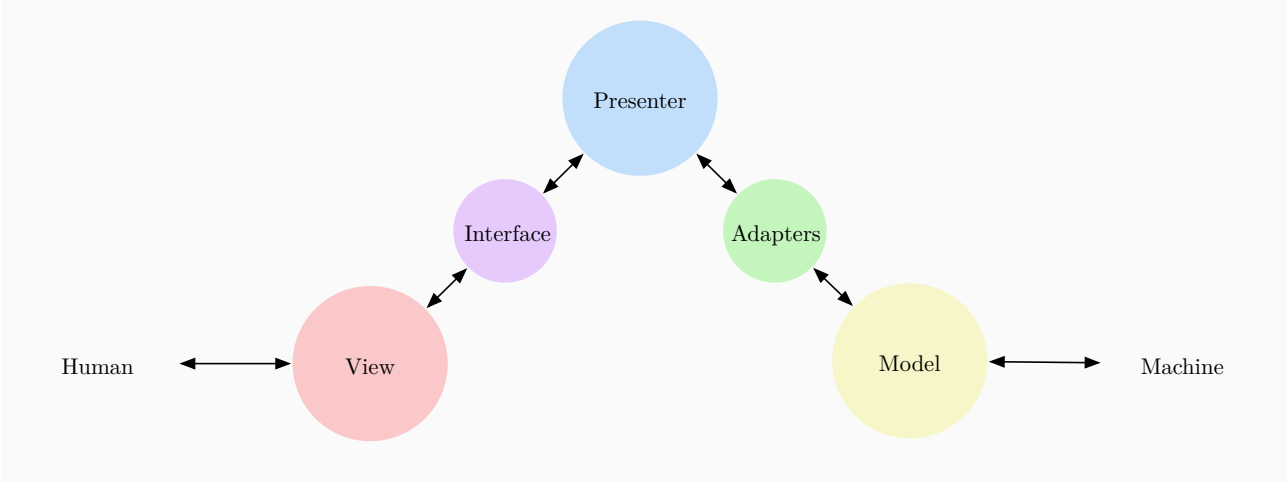


Figure 2.7: The MVP Model

2.1.7 Cocoa version of MVC

MVC gave birth to numerous variants. One of them is the Cocoa version of MVC [37], as used in Apple products. This version is relatively typical of modern MVC architectures, where the dichotomy between views as output components and controllers as input components is abandoned. Instead, views perform both input and output, while controllers play the role of controllers in the PAC model: synchronisation between hierarchical components and between views and models. The cocoa version of MVC specifies different kinds of specialised controller roles. Recent versions allows the composition of view controllers, which makes this approach very similar to PAC.

This approach is an attempt at solving the composition problems caused by the separation of input and output in MVC (Requirement 6 (Composition)), but the various kinds of controllers hamper the simplicity of the model (Requirement 3 (Simple)). However the separation between views and controllers is nicely executed in Cocoa: views are described in a graphical way, without code, and bound to the controllers declaratively. The simplicity of the View-Controller dialogue benefits from this abstraction (Requirement 8 (Abstraction)).

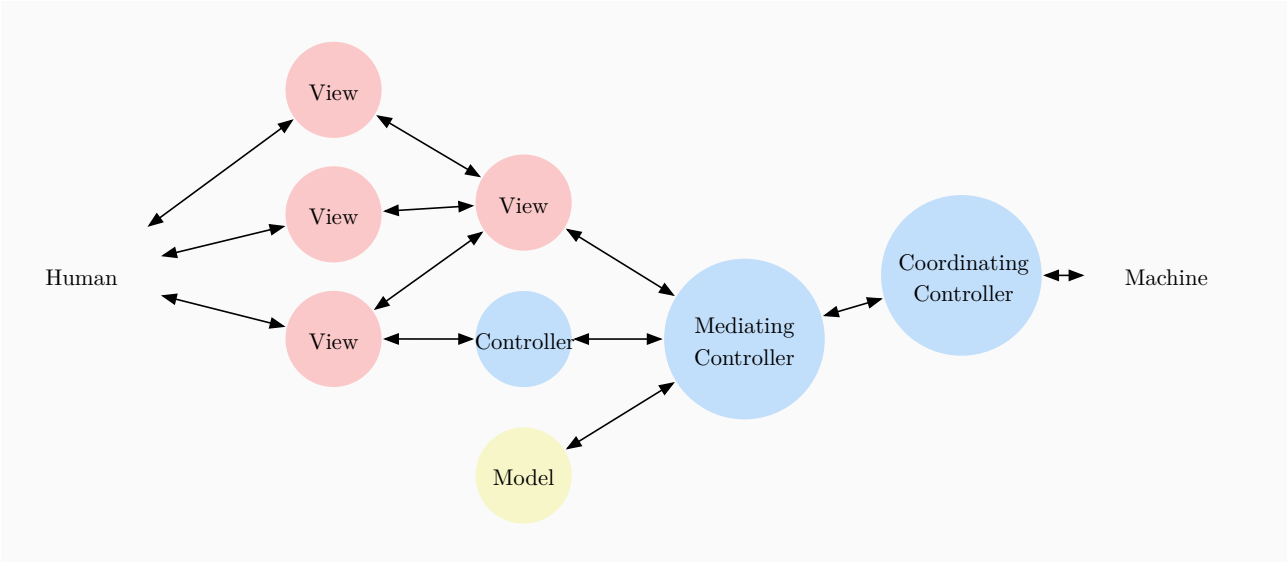


Figure 2.8: The Cocoa version of the MVC Model

2.1.8 Swing version of MVC

Swing is a widely used GUI toolkit for the Java programming language. Swing is based and promotes a certain architecture model. Like many other approaches, this model is based on a modified version of MVC. As explained in [38], the view and controller parts of a MVC component require a tight coupling (for example, it was very difficult to write a generic controller that didn't know specifics about the view). As a consequence, the version of MVC used in Swing collapses these two entities into a single object. This version is called the "separable model architecture" because it separates the Model from the rest. The main entities in this architecture pattern are:

- Models, which represents the abstract aspects of the components. Models are defined for all basic UI elements or widgets: buttons, lists, tables... Models have associated listeners and events. They corresponds to Models of the MVC pattern.
- Components, subclasses of `JComponent`, which contains both the MVC View and Controllers of UI objects. Components are associated to, but separated from, a model. Several different components can use similar models. For example, `JCheckBox` and `JToggleButton` use the same model, because they both represent the same model: a boolean value that can be switched on or off. Most components include a default model that can be overwritten. Models are subordinates of components, and each component as a model.

Figure 2.9 shows and example of architecture of a small UI. Overall, the hierarchical nature of the separable model architecture of Swing really enables to actually compose elements (Requirement 6 (Composition)), allowing a great deal of reusability (Requirement 7 (Modularity)). However, this model is very permissive, and it does not prevent components from listening to changes on the models of other components. When misused, this feature can end up with situations where causal loops exist, and a simple event can trigger a large cascade of asynchronous modifications, in an uncontrolled

manner. The multi-directional data flows that is authorised between various components is a bit too permissive to enable to easily understand complex systems and can lead to overly complex behaviours (hampering Requirement 3 (Simple)), unless the system is designed by following more restrictive rules [39].

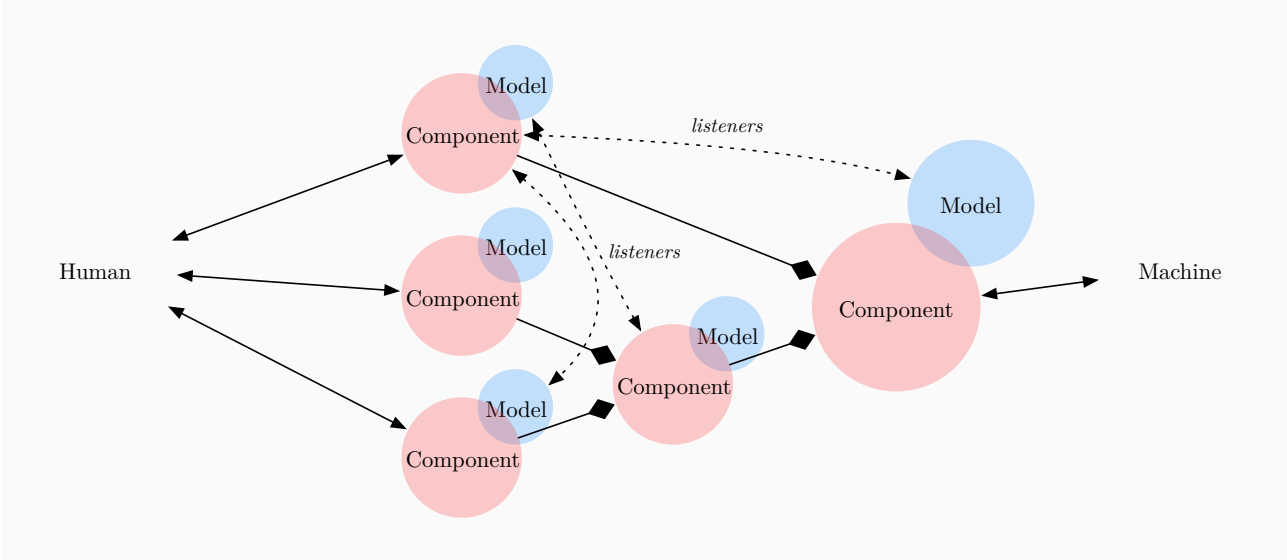


Figure 2.9: The separable model architecture of Swing

2.1.9 Presentation Model and Model View View-Model

Described in 2004 in [40]. The Presentation Model (PM) architecture is a distant variant of MVC. Figure 2.10 presents an overview of this architecture. It is based on 3 different component classes:

- Views: They represent the actual components with which the user interacts. They present graphical representations and they also take input events.
- Model: They represent business logic objects, their description is totally independent of any UI implementation.
- Presentation Models: They represent abstract models of the views, and they interact with the model. A presentation model contains only the abstract elements that are meaningful to the users, abstracting away the appearance or low-level behaviours.

The link between Views and Presentation Models is an important part of this architecture. This link is really simple, its only role is to synchronise states of views with states of views models and in the other way around, transmit input events to the presentation models. This main advantage of this approach is that these links can be managed automatically, because it always follows the same pattern: synchronise views with their view model. This process is called data binding. It provides some traceability (Requirement 5 (Traceability)) between two different aspects: behaviour and appearance. Both the Presentation Models and the Views can be composed hierarchically (Requirement 6)

Microsoft used this pattern in several of their tools, calling it Model View View-Model (MVVM) [41]. The main difference with PM is that MVVM emphasise the fact that data binding is generated automatically and that several reusable view components are provided. As a consequence, in a MVVM architecture, the developer does not have to develop the views anymore, they are abstracted away by the View Models (or Presentation Models). The developer can focus on more abstract aspects of the interaction (Requirement 8 (Abstraction))

This approach presents the advantage of providing a proper abstraction of the views, simplifying the work of the developer. However, data binding is not always as easy or straightforward as it seems.

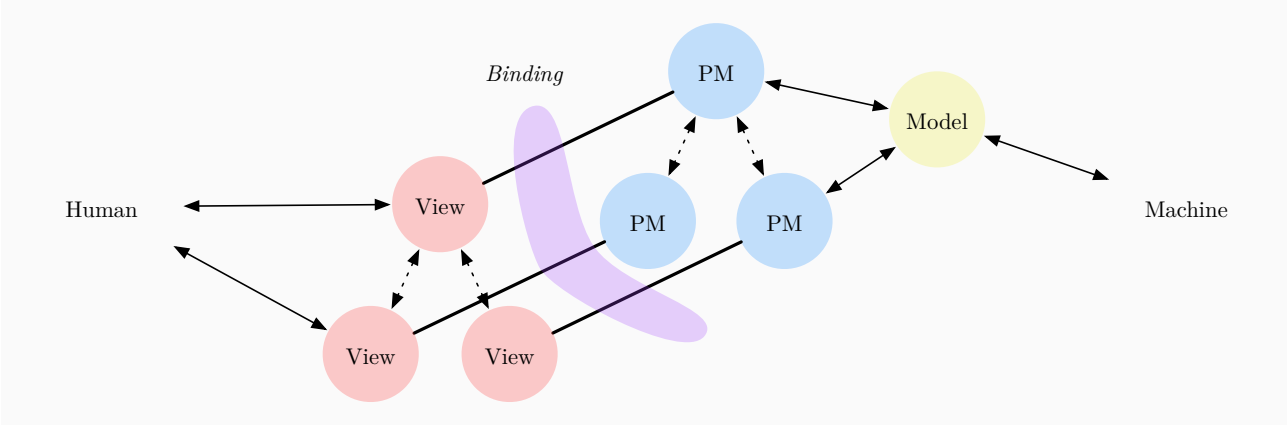


Figure 2.10: The PM and MVVM architecture

2.1.10 Qt signals and slots

Qt[42] is a framework that allows to program cross platform UIs using C++. It does not prescribe any strict architecture pattern. Instead it provides several main notions relevant to the field of UI architecture:

- Components: components can be defined as C++ classes, which can encapsulate several UI components themselves. The composition of components using C++ classes creates a hierarchy of ownership, where high level components owns several lower level components, down to the lowest level components. An important component class is the Widget, which serves as a base for many GUI elements. But other components can exist, such as models or delegates.
- Signals and slots: elements that allows to create peer-to-peer communication channels between components. The notion of signals and slot is orthogonal to the notion of composition. Any element can be connected to any element using signals and slots. The idea is that a component can define signals, which are actually event output ports, and slots, which are event input ports. A parent component can create connections between his descendants

Qt architecture is quite flexible, and the framework itself does not really impose any architecture pattern, but it is nonetheless interesting to study because the notion of signals and slots contributes to clarify the architecture of Qt UIs. Signals and slots attempts to provide an abstraction of useful concepts if the field of UI design (Requirement 8 (Abstraction)), while at the same time offering solutions for Requirement 7 (Modularity) and Requirement 6 (Composition).

As we see in Figure 2.11, while the composition of classes provides a clear hierarchical structure (plain lines), the signals and slots mechanism offers a lot of flexibility and allows connection between any two components of the hierarchy. A downside of this approach is that it might provide too much flexibility, when misused, it can give rise to overly complex networks of components, resulting in so-called “spaghetti code”, hampering Requirement 3 (Simple) and Requirement 5 (Traceability), as we tried to expressed in Figure 2.11.

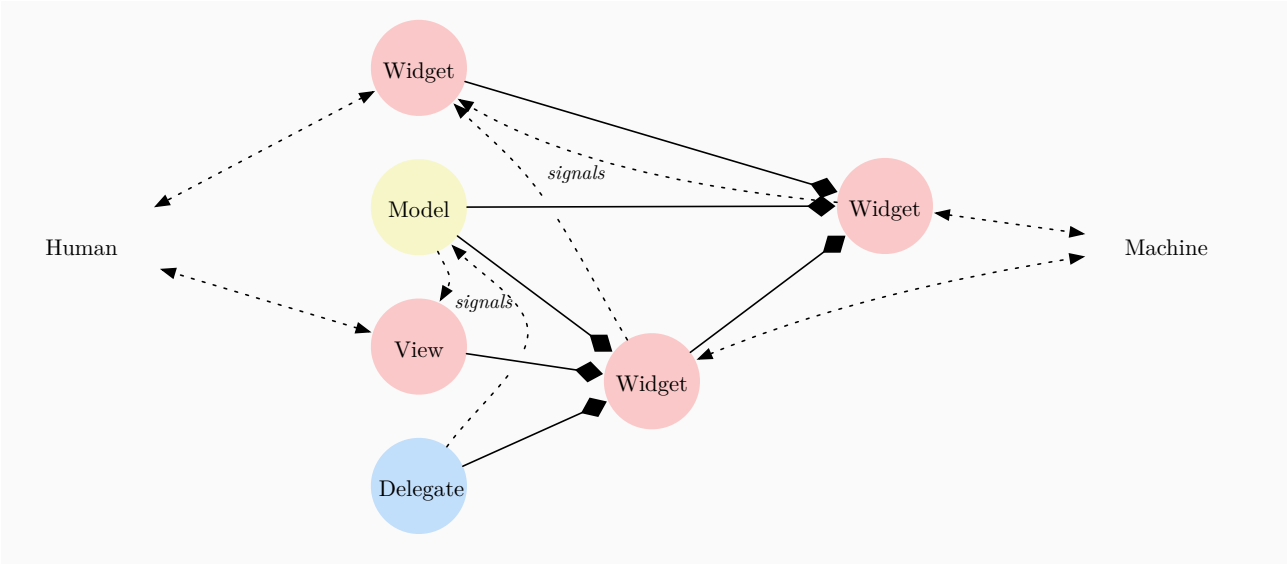


Figure 2.11: The architecture of Qt programs

2.1.11 Flux architecture

In 2012, Facebook unveiled React, a library for building user interfaces. Soon after, a matching architecture called Flux [43] was presented. The main goal of react and flux is to ease the development and maintenance of complex interactive systems such as Facebook websites and apps. The flux architecture is based on four different component types:

- Stores: Their role is similar to the models of the MVC approach. They contain all the data for a logical domain in the system. There can be several stores for several domains. Stores receives all actions that happens in the system and updates their data if necessary, based on the actions.
- Views: They are React components that render the data that comes from the store into elements of a view model. This view model can be the DOM in case of web-apps, but other models exists, such as native view in React-native[44], or 3D or 2D scene graphs. Views can also create actions when they receive user input.
- Action creators: Their role is to create semantic actions from sets of parameters. Actions contains parameters, which can for example be identifier of domain objects or values for parameters to be changed. Actions are send to the dispatcher.
- Dispatcher: There is only one dispatcher instance, it is a singleton, its role is to be a unique and central source of truth in the system. The dispatcher is generic, the same implementation can be used in any Flux application. Its role is to dispatch all incoming actions to all stores.

The main characteristic of flux is the notion of unidirectional data flow. We can see from Figure 2.12, that there is no double-sided arrows. The data flow follows a cycle: $Dispatcher \rightarrow Stores \rightarrow Views \rightarrow ActionCreators \rightarrow Dispatcher$. Authors of Flux argue against bi-directional data flow, where one change can loop back and have cascading effects. The unidirectional data flow presents the nice properties that event propagation is very well controlled and cannot loop. This makes models much easier to understand (requirement 3).

Another thing to notice is that the components called Views in this model are actually View-Models, as expressed in the MVVM approach. The React library performs the task of data binding, which is totally hidden away from the developer, which can think at a higher level of abstraction (Requirement 8 (Abstraction)). This enables an important aspect of the approach: Since the data flow is unidirectional, every time an event happens, *all* the views are re-rendered. This might seem a poor choice in terms of performance, but since the Views are actually View-Models, React performs a diff on the view

model, and actually renders only the views that changed. This diffing algorithm is lightweight and has little to no negative impact on performance. But thanks to this mechanism, development of React-Flux application is simple: Views have little to no behaviour, they just render data, and create actions on input events, the dispatcher is part of the library, actions creators are nothing more than constructors, and the business logic is contained in the stores, which contains business objects. Most of the complex event handling is alleviated. As a consequence, composition require very little glue to perform as expected. This provides excellent composition (Requirement 6 (Composition)) and modularity (Requirement 7 (Modularity)) properties to the approach.

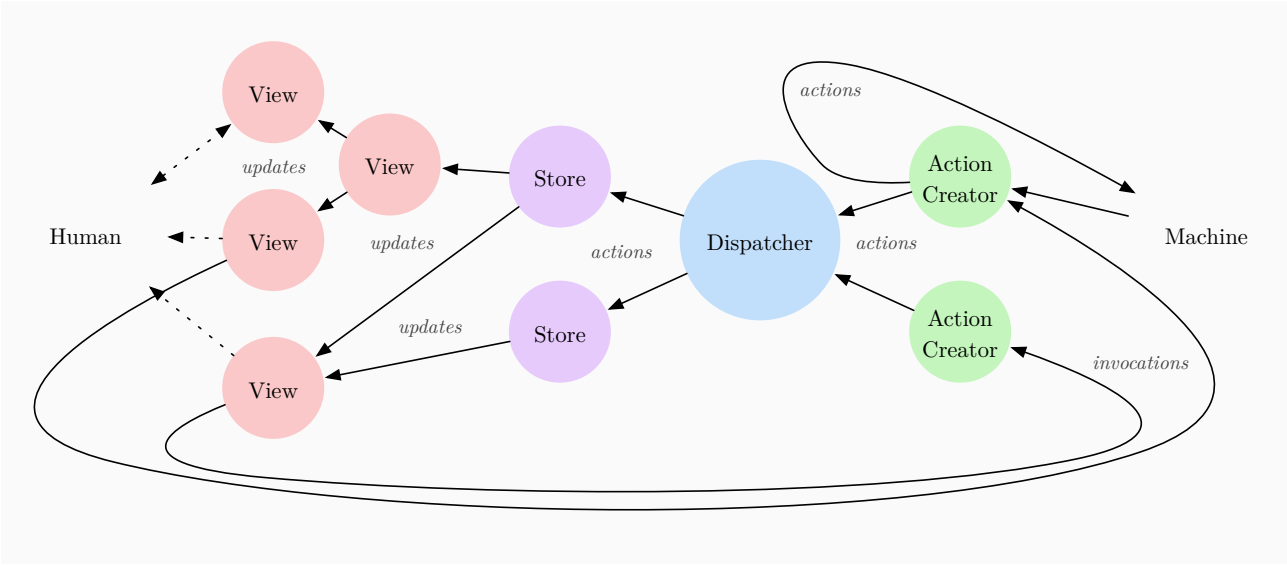


Figure 2.12: The Flux architecture Model

2.1.12 ARINC661 architecture

ARINC661[5] is a standard for CDS interfaces intended for all types of aircraft installations. The main goal of this specification is to offer a standard that increases inter-operability of avionic systems, hence reducing costs of interactivity in cockpits.

This standard introduces a two-tier architecture. It details the interactions between these two physical components:

- Cockpit Display System, which are physical systems and software that display information to users in the cockpit. They also manage interactive events and rendering details. They can autonomously display interfaces that are specified in the form of Definition Files (DFs) and animate their low-level behaviour. For example, they manage cursor movements and widget behaviours (animations when clicked...)
- User Application, which are software application that are executed in other parts of the plane and can communicate with CDS by using the Arinc 661 protocol over aircraft networks systems (AFDX for example). User Applications manage the high level behaviour of interfaces by sending widget parameter changes (visibility of widgets and panels, texts...)

One of the very positive points of ARINC 661 is that it is one of the first standards that allowed interoperability between CDSs by providing adapted abstractions (Requirement 8 (Abstraction)), even if the possibility of custom widgets and behaviours makes this interoperability less than perfect. Another interesting point in that all input and output between the two components is formalised and follows the same pattern: User applications set parameters to display information to the users, while

the CDS send widget events to send information to the User application. This standardisation is a very interesting point and a starting point to formal analysis of the systems (Requirement 1 (Formal)). On the other hand, Arinc 661 allows limited dynamism of interfaces, which is a natural consequence of its orientation towards critical systems.

A point of the standard that needs some improvement is the problem of the rapid feedback, as explained in the Seeheim model (Section 2.1.1). Some user events are provided with fast feedback, when the loop stays inside the CDS, for example when the user clicks on a button, the button displays as "pushed" almost immediately. But some other events triggers loops that have to go all the way to the user application and back in order to provide feedback to the user. For example, if the user asks to see a list, all the elements of the list have to be sent from the user application to the CDS through the aircraft network, which leads to delays which are difficult to reduce. In fact, the positioning of the interface between the User Application (UA) and the CDS as defined in the standard is debatable. A more abstract or more concrete interface might have helped simplifying the standard, making it easier to understand by all stakeholders (Requirement 3 (Simple)).

A good point of the standard is that it provides some kind of traceability between the appearance in the CDS and the behaviour in the UA (Requirement 5 (Traceability)), by identifying each and every widget in a unique way. The ARINC 661 architecture is specifically designed for aerospace HMIs.

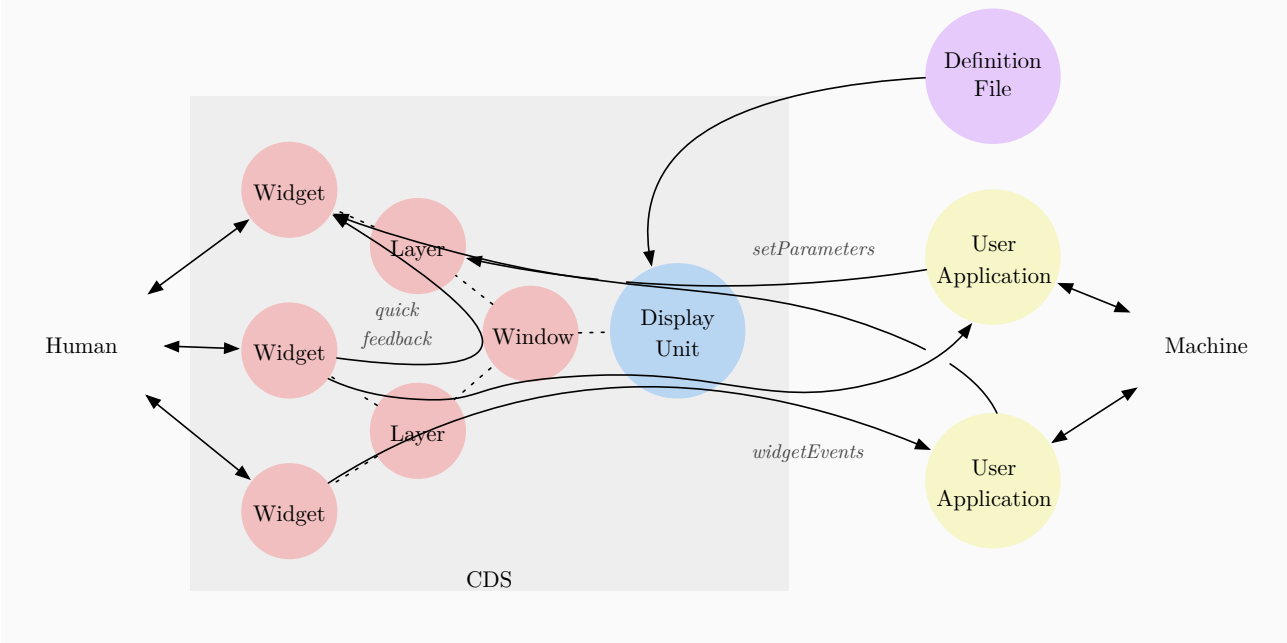


Figure 2.13: The logical architecture model of an ARINC 661 system

2.1.13 Conclusion

As synthesised in Table 2.1, elements described in this section provide several good ideas in order to fulfil several requirements that we identified in Section 1.3.

	1 Formal	2 Collaborative	3 Simple	4 Projection	5 Traceability	6 Composition	7 Modularity	8 Abstraction	9 Manage	10 Prototyping	11 Verification	12 Flexibility
2.1.1 Seeheim							-	+	+			-
2.1.2 Arch						-		+	++			
2.1.3 MVC							+	+	+			
2.1.4 PAC							++					
2.1.5 PAC Amodeus							+					
2.1.6 MVP						-	+					
2.1.7 Cocoa			-			+		+				
2.1.8 Swing			-			++	+					
2.1.9 MVVM					+			++				
2.1.10 Qt			-		-	+	++	+				
2.1.11 Flux						++	+	++				
2.1.12 ARINC	+		-		++			+				

Table 2.1: Requirements vs HMI architecture state of the art

2.2 Models of network architecture

Networks are made of systems that interact by exchanging information. This is a common point with HCI systems. The field of networking systems has been the subject of a lot of research, and achieved great results, the most prominent of which being the internet and mobile networks. This section presents network architecture models that had an influence on the works presented in this thesis.

2.2.1 OSI Model

Most network architecture models are based on the now classical OSI model first presented in [45]. This model is usually depicted with seven layers [46], as described in Figure 2.14. The layers have various roles, that, when combined in a certain order, allow to provide communication between open systems. The architecture of the Internet is based on this model. We find the following layers, described from the lower-level to the higher-level:

- Physical: Transmits and receive data over a physical medium. The Ethernet physical layer is an example of a protocol of this layer.
- Data link: Provides reliable transmission of data frames between two nodes connected by a physical layer. Media Access Control (MAC) is a protocol of this layer.
- Network: Structures and manages multi-node networks, including addressing, routing and traffic control. Internet Protocol (IP) is a protocol of this layer.
- Transport: Provides reliable transmission of data segments between points on a network, including segmentation, acknowledgement and multiplexing. Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) are protocols of this layer.
- Session: Manages the continuous exchange of information in the form of multiple back-and-forth transmissions between two nodes. Secure Sockets Layer (SSL) is a protocol of this layer.
- Presentation: Translates data between a networking service and an application, including character encoding, data compression and encryption/decryption. HTML and other file formats belongs to this layer.
- Application: High-level communication between specialised application for specific purposes. HyperText Transfer Protocol (HTTP) is a protocol of this layer.

We see that this various layers have various roles that each solve a specific problem that has to be tackled to allow large scale communication over networks. In a way, we could say that each layers abstracts a problem away from the layer above. In most cases, agents of these layers perform transformations on the messages to be sent, merging pieces of data in larger messages, breaking them down into smaller messages, encapsulating these messages into messages with a header, encoding messages and so on. Transmission of the messages is performed using protocols, which are specific to each layer.

The OSI model has been successful at describing many aspects of the internet architecture, including the various functions of layers explained above, by providing a convenient abstraction to use as base (Requirement 8 (Abstraction)). But it is now widely agreed that this architecture fails to meet many of society's present and future requirements [47] [48]. In particular, this model represents a fixed number of layers, which fails to model the multiple, customised protocol stacks that navigate on the internet. Most of actual messages that transit on the internet go through much more than 7 layers, especially since the advent of new technologies such as mobile networks, video streaming or Virtual Private Networks (VPNs).

Another limitation of the OSI model is that it does not describe the common situation where multiple protocols exist for the same layer. There are actually dozens of various protocols in wide use over

the internet, and the OSI model does not explicitly tackle the issue of managing this complexity (Requirement 9 (Manage)).

There is a striking parallel between the OSI model and the Seeheim and Arch models for HCI architecture, described in the previous section. These models were devised at around the same time. Their models are strongly layered, with a fixed number of layers which are strictly ordered. In both cases, the result is a reference model structured subsequent approaches for many years, but which is becoming limited when it comes to tackling new challenges, whether they are multimodal and new interaction modalities, or new internet technologies, in other words, more diversified architectures. These models had to be very structuring, being the first approaches to structure complex fields, but this hampered their flexibility (Requirement 12 (Flexibility)).

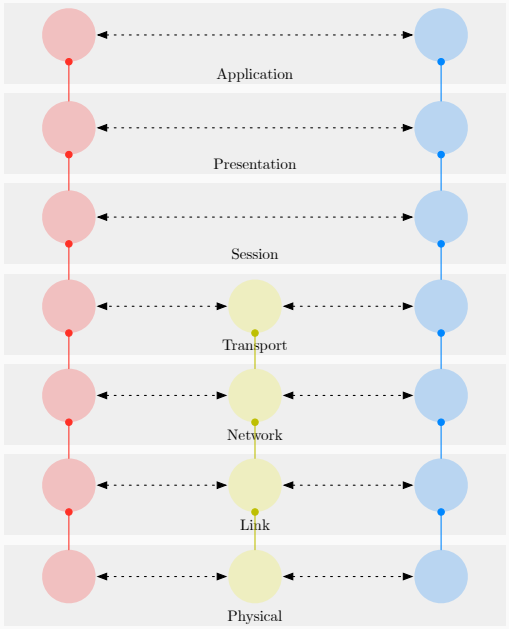


Figure 2.14: The classical OSI Model

2.2.2 Geomorphic view of networking

In [49], a more flexible and detailed model of the architecture of the internet is described. This model is called geomorphic because its possible arrangements of layers resemble the complex arrangements of layers in the earth's crust. In the OSI architectures, each layer has global scope and there is exactly one layer at each level of the hierarchy. But in the geomorphic view, each layer instance corresponds to something real and specific, such as a particular Local Area Network (LAN), as opposed to a generalisation. A layer can have a small scope, and there can be many layers at the same level of the hierarchy. This solves the flexibility problem of the OSI model (Requirement 12 (Flexibility)), and gives a nice way to manage abstraction levels (Requirement 9 (Manage)).

Architecture models described using the geomorphic view present various different layers, which appear at different levels of the hierarchy, with different scopes, with different versions of the basic ingredients, and for different purposes. Figure 2.15 shows the geomorphic view of the classic Internet architecture, with many LAN layers at the bottom level. In each LAN layer, the data structures, algorithms, and protocols are precisely those of the particular LAN technology being used. This is in sharp contrast to the idea of a generic, global link layer expressed in the OSI model, which cannot be made precise because it is a generalisation of a large number of different technologies.

The beauty of the geomorphic view is that any lesson we learn about layers in general can be used many times over. The geomorphic model allows to notice common patterns that can happen at various levels of the network hierarchy. For example, its authors focused on the description of strategies that allow mobility, and found that these solutions are all based on two different mobility patterns, proving the abstraction and formalisation capabilities of this model (Requirement 8 (Abstraction) and (Requirement 1 (Formal))). However the model does not say much about the behaviour of agents of each layer.

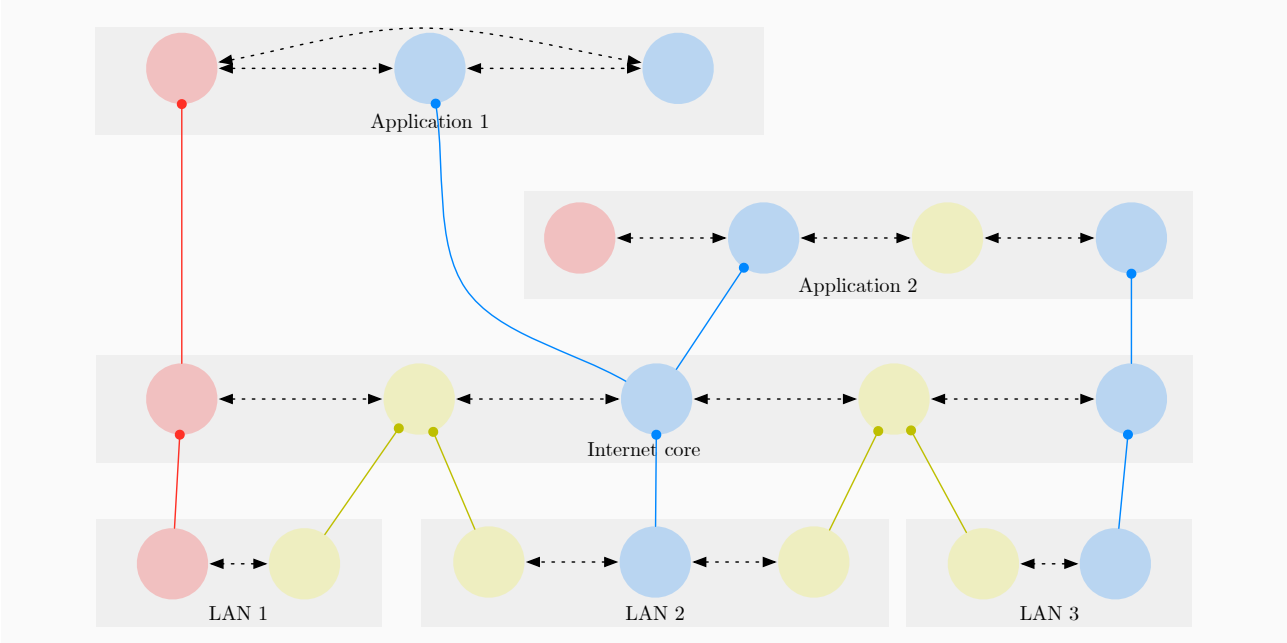


Figure 2.15: Geomorphic view of the classic Internet architecture.

2.2.3 Conclusion

As synthesised in Table 2.2, elements described in this section provide several good ideas in order to fulfil several requirements that we identified in Section 1.3. In particular, the geomorphic model shows us a powerful way to generalise an initially rigid model (the OSI model) of a complex system (the Internet) that has outgrown this rigid model.

	1 Formal	2 Collaborative	3 Simple	4 Projection	5 Traceability	6 Composition	7 Modularity	8 Abstraction	9 Manage	10 Prototyping	11 Verification	12 Flexibility
2.2.1 OSI								+	-			–
2.2.2 Geomorphic	+							+	++			++

Table 2.2: Requirements vs network architecture models state of the art

2.3 Interactors

The notion of interactor is a useful concept in the field of critical HCI where formal descriptions of interactive systems are needed. The idea comes from a need to abstract away some low level problems and describe generic agents that perform basic interactions and can be composed in order to bring out complex interactive systems. There are various interactor-based approaches, but they present some common points:

- Interactors are described as stateful agents that can interact and be combined in order to describe interactive systems.
- A formal representation of interactors is provided and can be used to check properties of interactive systems at an abstract level.
- Present an architecture model of UIs as a composition of interactors whose internal architecture is itself defined, and which presents two sides: the user side and the machine side.
- Present the behaviour of UIs as a property that emerges from certain compositions of interactors.

2.3.1 CNUCE Interactors

Presented in [50] and originating in Centro Nazionale Universitario di Calcolo Elettronico (CNUCE), this approach provides an architecture model with two levels of abstraction. At a higher level, interactors are seen as black boxes, and their composition is studied. As explained in [51], interactors have two sides: the machine side and the user side. On each side is an input port, an output port, and a trigger port, which triggers the output of information on the other side of the interactor. CNUCE interactors are event based, their input and output are events. As described in Figure 2.16, at a lower level, the approach prescribes interactors with an internal structure made of four different components:

- Collection: Represents an abstraction of the external appearance of the interactor. It is similar to the View-Model component of the subsequent MVVM model (Section 2.1.9).
- Abstraction: Transforms user actions into events that are meaningful to the machine. It is somehow similar to the Model or Model adapter in other architecture models.
- Presentation: Update the various elements visible to the user depending on events coming from the collection. This is similar to the Views of the original MVC, at a more abstract level however.
- Measure: Receives and measure the user inputs and transforms them into abstract actions that are sent to the abstraction. This is similar to the Controllers of the original MVC.

Together, the Presentation and Measure components provide a functionality similar to the View components of MVVM. The internal architecture of interactors is not symmetric. We see in particular that the presentation can use events from the measure component, something that exists in the MVC model in the form of messages from low-level controllers to views, and which is named rapid feedback in the Seeheim model (Section 2.1.1). We also see that the measure component can use the signal coming from the collection in order to elaborate its output. This asymmetry of the model somehow hampers modularity of the model (Requirement 7 (Modularity)).

On a higher level, a problem of the CNUCE interactor architecture is that it provides composition in the form of juxtaposition of interactors and linking of their ports, but it does not provide ways to abstract a complex composition of interactors in the form of a higher level interactor which could be reused as a black box (Requirement 7 (Modularity)). However, CNUCE interactors were one of the first formalisation of UI using base components, and paved an interesting way for other approaches based on interactors. This formalisation is based on process algebras, fulfilling Requirement 1 (Formal).

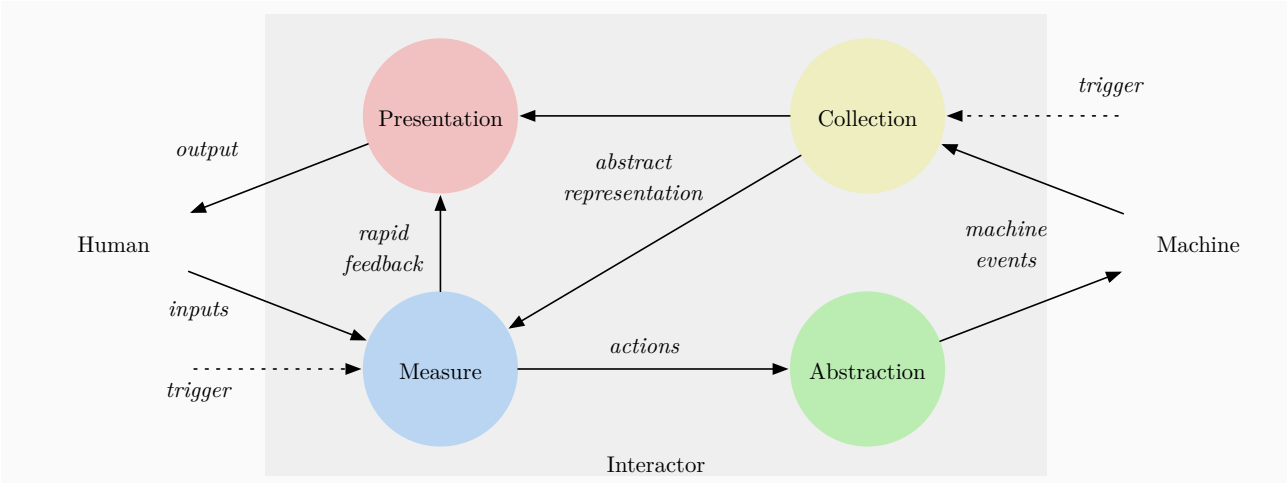


Figure 2.16: CNUCE interactors internal architecture

2.3.2 York Interactors

The idea of modelling UIs using interactors was developed further in [52] at the University of York. In this approach, the idea of interactor is generalised by depicting them as systems that encapsulate a state, input events to manipulate this state and output events generated by the state changes, and a way to present states to human users. Figure 2.17 presents an example of architecture based on york interactors. In this architecture, interactors contain:

- A state, which represents the abstract information necessary to operate the interactor. This state is manipulated using input and output events that allow communication between interactors.
- A presentation, which represents what the user actually sees and acts on, the actual graphical representation.
- A rendering relationship, which describe how state history (or traces) map to presentations. The presentation and rendering relationship are new concepts introduced in York interactors, as compared to the CNUCE model that omits these aspects.

Like CNUCE interactors, York interactors do not provide a way to encapsulate interactors within higher level interactors (Requirement 7 (Modularity)). However, the York approach says more about composition of interactors: they are described by connecting output events of some interactors to input events of other interactors (Requirement 6 (Composition)). York interactors were originally described using the Z notation [53] (Requirement 1 (Formal)). An event-based evolution of the model was proposed in [54], and a unification of York interactors with CNUCE interactors was proposed in [55].

Like CNUCE interactors, York interactors have an asymmetric structure: the User and the Machine are treated differently. This is justified because Human users and Machines are very different. But as a consequence, the architecture is intrinsically dedicated to the description of GUI with one user and one machine. This makes it difficult to generalise these interactors to more diverse UIs such as multimodal and multi-user interfaces (Requirement 12 (Flexibility)).

2.3.3 CERT Interactors

In [56], an approach inspired by York interactors is described by researchers from Centre d'Études et de Recherche de Toulouse (CERT). It generalises again the notion of interactor by describing them

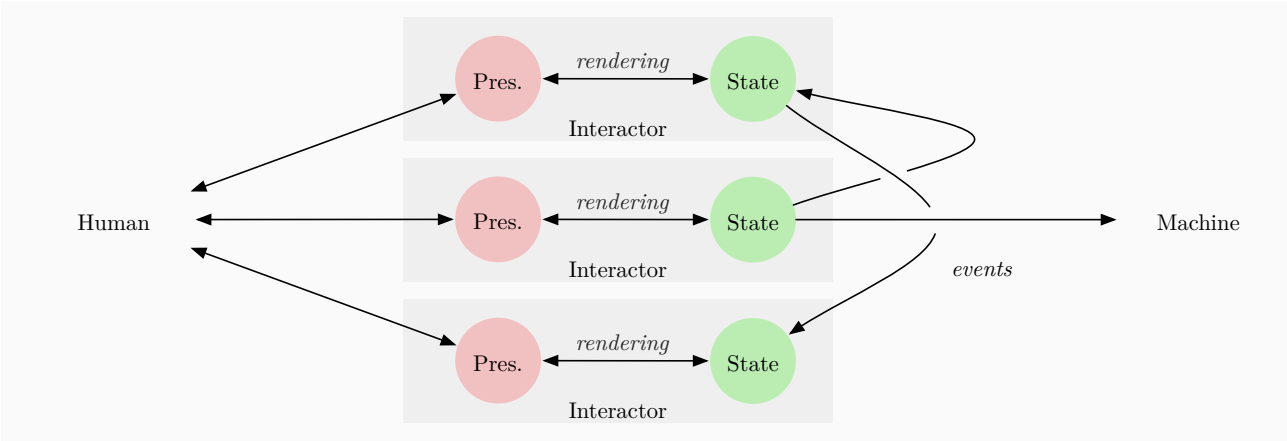


Figure 2.17: York interactors internal architecture

as abstract elements that take several input data flows and output several output data flows (Requirement 8 (Abstraction)). The internal behaviour of interactors and their composition is formally expressed in LUSTRE [57] (Requirement 1). Each interactor is described as a LUSTRE node (Requirement 7 (Modularity)). The equation system of each interactor deterministically maps input data flows to output data flows which are send to the user, the machine or other interactors. Interactors can have internal states. Composition of interactors is done by using LUSTRE node composition facilities (Requirement 6 (Composition)).

The simple semantics of LUSTRE, in particular the fact that signals are synchronous data flows, makes the CERT interactors relatively simple to understand for programmers (Requirement 3 (Simple)), but yet very general (Requirement 12 (Flexibility)).

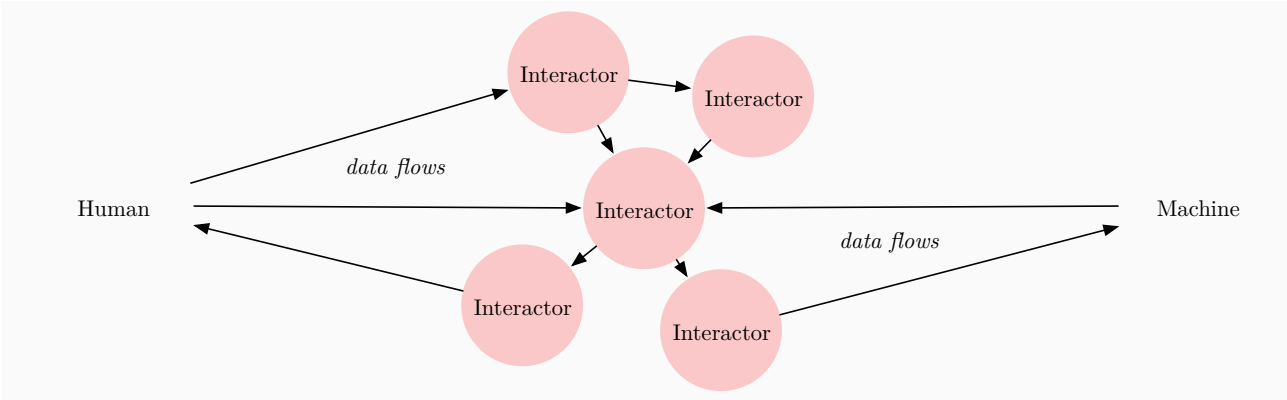


Figure 2.18: An architecture based on CERT interactors

2.3.4 Conclusion

UI models based on interactors provide an excellent way to formalise the behaviour of UIs in a modular way. Several iterations on the concept improved the solution, enabling interesting results. However, all these solutions present the inconvenient of being understandable only by qualified people. Table 2.3 presents an overview of the section.

	1 Formal	2 Collaborative	3 Simple	4 Projection	5 Traceability	6 Composition	7 Modularity	8 Abstraction	9 Manage	10 Prototyping	11 Verification	12 Flexibility
2.3.1 CNUCE	+		-				-					
2.3.2 York	++		-			+	-					-
2.3.3 CERT	++		-			+	++	+				+

Table 2.3: Requirements vs interactor based models state of the art

2.4 Imperative and object-oriented programming of UI behaviour

This section presents approaches that use general purpose imperative programming languages in order to specify the behaviour and appearance of UIs. These approaches are used in the vast majority of industrial and commercial UIs, and represent the traditional way of building UIs on windows, linux, OS X, iOS, Android... The common points of these approaches are:

- A general purpose, object oriented programming language.
- A library that is the dominant standard for UIs in the given language or operating system
- UI objects are instantiated dynamically at run-time
- A graphical WYSIWYG editor that generates code to construct static aspects of UIs
- Pairing with an architecture model. These architecture models are often a modified version of MVC, sometimes closer to PAC. Some of these architecture models are described in Section 2.1

The great success of these approaches shows that they are a convenient and general way of programming UIs. However some suffer from several drawbacks. For example, the object oriented nature of these approaches means that composition of elements always necessitates a lot of glue code, which goes against Requirement 6 (Composition). These approaches are made for software programmers, they were not intended to fulfil Requirement 3 (Simple), but they allow to use the whole panoply of tools for collaborative software development, such as version control systems, satisfying Requirement 2 (Collaborative).

We will present these approaches by applying them to model a simple application that takes a number (*theNumber*) as an input, and outputs this number incremented by one (*theResult*). This will allow us to see the advantages and drawbacks of these approaches.

2.4.1 C++, Qt and Qt creator

Qt is one of the major cross-platform UI framework for the C++ language. Since it is based on C++, we can say that it is definitely intended for programmers and not for the rest of the stakeholders. Requirement 3 (Simple) is not the goal of this approach. We explained the architecture model associated with this approach in Subsection 2.1.10, especially the notion of signals and slots.

Listing 2.1 presents the simplified code of a simple Qt application that gets a number as input (*theNumber*) and outputs this number incremented by one (*theResult*). We see that the code is not easily understandable. However we note that it has a structure:

- Lines 1-13: The structure of the component is expressed: It contains a `LineEdit` widget (Line 11), a `Label` (Line 12). It has a slot, which is the denomination for an input port (Line 7), and a signal, which is the denomination for an output port (Line 9).
- Lines 15-18: The initial behaviour of the UI is expressed: initially `theNumber` is 0, and `theResult` displays 1.
- Lines 20-22: Some low level instructions define the positioning of widgets.
- Lines 24-30: Two connections between components are expressed using signals and slots. The first connection states that the signal `textChanged` of `theNumber` is sent to the slot `theNumberChanged` of the application. The second connection states that the signal `changeTheResult` of the application is sent to the slot `setText` of `theResult`.
- Lines 34-36: The behaviour which is triggered when the slot `theNumberChanged` receives a message is expressed. It states that the application should then send a value on its signal `changeTheResult`.

We see that QT allows components are dynamically instantiated at run time. This is difficult to model formally, which makes this approach difficult to formalise easily (Requirement 1 (Formal)).

```
class Window : public QWidget
{
  Q_OBJECT
  public:
    explicit Window(QWidget *parent = 0);
  public slots:
    void theNumberChanged(QString value);
  signals:
    void changeTheResult(QString value);
  private:
    QLineEdit *theNumber;
    QLabel *theResult;
};

Window::Window(QWidget *parent) : QWidget(parent)
{
  theNumber = new QLineEdit("0", this);
  theResult = new QLabel("1", this);

  setFixedSize(100, 50);
  theNumber->setGeometry(0, 0, 100, 25);
  theResult->setGeometry(0, 25, 100, 25);

  connect(
    theNumber,  SIGNAL (textChanged(QString)),
    this,       SLOT (theNumberChanged(QString))
  );
  connect(
    this,       SIGNAL (changeTheResult(QString)),
    theResult,  SLOT (setText(QString))
  );
}

void Window::theNumberChanged(QString value) {
  emit changeTheResult( QString::number(value.toInt() + 1)  );
}
```

Listing 2.1: The code of a simple QT application

Qt Creator is a tool that allows to graphically describe the structure of UIs, which alleviates a lot of work for the programmers, and also allows other stakeholder to get a grasp of the UI structure without having to understand the code. (Requirement 3 (Simple))

2.4.2 Java, Swing and Swing designer

As explained in Subsection 2.1.8, Java Swing provides an interesting architecture model that provides a nice way model to compose UIs elements (Requirement 6 (Composition)). However, being a general purpose programming language, Java itself is not the best language to describe UIs. The syntax is relatively verbose, and the object-orientation does not map really well to the description of UIs. Java applications often necessitate a lot of boilerplate code [58], code not related to actual matter of UI.

```java
public class Hello extends JFrame {

  private JPanel contentPane;
  private JTextField theNumber;
  private JLabel theResult;

  public Hello() {
    contentPane = new JPanel();
    setContentPane(contentPane);

    theNumber = new JTextField();
    theNumber.setText("0");
    theNumber.setColumns(10);
    theNumber.addActionListener(new ActionListener() {
      @Override
      public void actionPerformed(ActionEvent e) {
        theNumberChanged(Integer.decode(((JTextField)e.getSource()).getText()));
      }
    });
    contentPane.add(theNumber);

    theResult = new JLabel("1");
    contentPane.add(theResult);

    setVisible(true);
  }

  public void theNumberChanged(Integer newValue) {
    theResult.setText(Integer.toString(newValue + 1));
  }
}
```

Listing 2.2: Code of a simple Java-swing application

Listing 2.2 presents the code for a simple Java Swing application that takes a Number and output the same Number plus one. The example is around 30 Lines Of Code (LOC) long. Out of these 30 lines, around 10 are effectively important to specify the actual UI structure at a higher level. Five lines are useful to describe the structure of the interface:

- Line 8: there is a Panel called `contentPane`
- Line 11: there is a Text input called `theNumber`
- Line 22: there is a Label called `theResult`
- Line 20: `contentPane` contains `theNumber`
- Line 23: `contentPane` contains `theResult`

And five lines are actually useful to describe the behaviour of the interface:

- Line 12: initially, `theNumber` is 0.
- Line 22: initially, `theResult` displays 1.
- Line 17: when `theNumber` is changed, it triggers a behaviour called `theNumberChanged`
- Line 29: `theNumberChanged` takes a number, adds one to it, and displays it on `theResult`

As far as User Interface (UI) designers are concerned, the rest of the code is either boilerplate code (lines 1 - 7), lower level code (lines 13, 15, 25), or code to glue together the various components (line 14-16, line 28). Another thing we can notice is that the code mixes several aspects: the structure of the UI and its behaviour are scattered on various lines across the source file, which makes the source file unreadable for non-programmers (Requirement 3 (Simple)).

The most problematic aspect is the notion of listener (line 14-19). It is a common issue with all event-based object-oriented approaches. However it is particularly striking in Java because of its strict object-orientation. Listeners represent the way to link various components together, by making them "listen" to events coming from other components. Listeners causes a lot of accidental complexity[59]. In particular,[60] cites the "Six plagues of listeners":

- Unpredictable order: In a complex network of listeners, the order in which events are received can depend on the order listeners where registered in.
- Missed first event: It can be difficult to guarantee that all listeners are registered before the first event is sent.
- Messy state: The state of the system is distributed across various components and is not guaranteed to be consistent.
- Threading issues: Attempting to make listener thread safe can lead to deadlocks.
- Leaking callbacks: If listeners are not deregistered properly, programs might leak memory.
- Accidental recursion: Listeners can create callback loops which never end.

There are many positive points to programming in Java Swing however: Thanks to Java facilities, Modularity and reusability of the code is satisfactory (Requirement 7 (Modularity)). Being based on a general purpose language, the approach is very flexible (Requirement 12 (Flexibility)). Several tools such as swing designer also allow to create simple prototypes quickly, (Requirement 10 (Prototyping)). The clear and unambiguous semantics of Java also allows to perform verification on UI programs [39] (Requirement 11 (Verification)).

2.4.3 Cocoa, Swift and Interface Builder

Swift is a relatively recent programming language presented by Apple Inc. in June 2014. Swift was designed from the ground up to work with the Cocoa UI framework. This makes the language very appropriate to describe UIs behaviours. Swift is associated with a tool called Interface Builder, which allows to describe the structural and graphical aspects of interfaces. As a consequence, the two aspects (structure and behaviour) of UIs are clearly segregated, which helps collaborative work (Requirement 2 (Collaborative)) and makes the artefacts simpler (Requirement 3 (Simple)).

Listing 2.3 presents the code for the same simple application we described before. It takes a number as an input, and outputs the given value plus one. We see that the code that describes this behaviour

is relatively clear, and can be understood relatively quickly.

- Line 5: There is a `UITextField` called `theNumber`
- Line 6: There is a `UILabel` called `theResult`
- Line 10: Initially, `theNumber` is 0
- Line 11: Initially, `theResult` displays 1
- Line 14: There is an action `theNumberChanged` which can be triggered by a Ui object
- Line 15: `theNumberChanged` takes the value of `theNumber`, adds one to it, and sends the result to `theResult`

```
1  import UIKit
2
3  class ViewController: UIViewController {
4
5      @IBOutlet weak var theNumber: UITextField!
6      @IBOutlet weak var theResult: UILabel!
7
8      override func viewDidLoad() {
9          super.viewDidLoad()
10         theNumber.text = "0";
11         theResult.text = "1";
12     }
13
14     @IBAction func theNumberChanged(sender: AnyObject) {
15         theResult.text = String( Int(theNumber.text!)??0 + 1);
16     }
17
18 }
```

Listing 2.3: Code of a simple Swift Cocoa application

The structural and graphical aspects are described separately in Interface Builder.

Swift provides built-in abstractions which are very relevant to the field of UI. For example, the optional types and associated operators are first class citizens in the language. As an example the `??` operator allows to give default values to optional variables: `myInput??defaultValue` will take the default value if there is no input. This behaviour is formally defined (Requirement 1 (Formal)), and eases the understanding of the behaviour of the UI in edge cases (Requirement 3 (Simple)). Other interesting concepts built in the language are the notion of user actions (`@IBAction`) and outlets (`@IBOutlet`). Overall, we can say that Swift provides convenient UI abstractions, fulfilling Requirement 8 (Abstraction).

However, Requirement 7 (Modularity) is not perfectly fulfilled by Swift and Cocoa. While it allows to compose components in a very convenient way, defining new reusable components is a bit more complex and involves more programming, especially if these new components are to be integrated using Interface Builder.

2.4.4 Conclusion

Comparing the actual code for different imperative and object oriented languages gives a first set of insights about what makes a UI programming language easy to understand, and which abstractions can be useful. Table 2.4 synthesises the conclusions of this section.

	1 Formal	2 Collaborative	3 Simple	4 Projection	5 Traceability	6 Composition	7 Modularity	8 Abstraction	9 Manage	10 Prototyping	11 Verification	12 Flexibility
2.4.1 Qt	-		-				+					
2.4.2 Swing			-			+	+			+	+	+
2.4.3 Cocoa	+	+	++				+	+				

Table 2.4: Requirements vs imperative programming of UIs state of the art

2.5 Textual Domain-Specific Languages for UIs

Domain-Specific Languages (DSLs) [61] are programming languages specialised for a certain application domain [61]. DSLs have recently became more popular due to the rise of Domain-Specific Models (DSMs). DSLs can be based on underlying DSMs, or they can be a subset of an underlying general purpose programming language, in which case they are called Embedded DSLs. Several DSLs for UI exist, and we will describe some of them in this section. As compared to general purpose languages, the goal of DSLs is to provide built-in abstractions relevant to the domain, which would help comply with Requirement 8 (Abstraction). By striping off the programming aspects, they should also make artefacts easier to understand (Requirement 3 (Simple)).

Some approaches allow to automatically derive visual DSMs from meta models [62], which is a good sign of flexibility (Requirement 12 (Flexibility)).The common points to the DSLs presented in this section are:

- Textual DSL, which allow to use traditional version control systems (Requirement 2 (Collaborative))
- A mix of declarative syntax to express the structure of UIs and of high level imperative syntax to describe UIs behaviour.
- A link with a general purpose programming language in order to enable complex behaviours that cannot be expressed within the DSL (Requirement 12 (Flexibility))
- An important reduction in the amount of glue code between components, which can be composed syntactically instead of being juxtaposed and linked (as we saw with general purpose languages, Section 2.4). The composition is done at a syntactical level, by actually nesting components definitions. This proved to be an excellent solution for easy composition of components (Requirement 6 (Composition))

2.5.1 Qt Modelling Language

We saw in Section 2.4.1 that Qt is a framework to create UIs using C++, a general purpose programming language. In 2009, Qt started offering an alternative way to program UIs: Qt Modelling Language (QML) [63]. QML is a DSL to create Qt applications. It is losely based on JS and Javascript Object Notation (JSON) [64]. It provides a very simple way to define the structures of UIs, without any boilerplate code. Listing 2.4 shows the code of our example application that takes a number and outputs this number incremented by one, expressed in QML. We observe that the code is simple and easy to understand:

- Line 1-4: The UI is made of an `ApplicationWindow`, with a given size.
- Line 5: The contents of the window are laid out as a column, one above each other.
- Line 5-9: The window contains a `TextField` called `theNumber`, which initially contains the text "0".
- Line 10-13: The window contains a `Label` called `theResult`, which displays the value contained in `theNumber` plus one.

As observed, QML is an easy to understand notation (Requirement 3 (Simple)), composition of elements is trivial (Requirement 6 (Composition)), and creating new reusable components is also easy to do (Requirement 7 (Modularity)). The notation is very terse, which allows to quickly develop prototypes (Requirement 10 (Prototyping)). QML is also prone to evolution, new base components can be developed (Requirement 12 (Flexibility)). QML proves to be an excellent solution to design UIs, which is the whole purpose of designing a DSL.

The strong orientation of QML towards UI modelling has a consequence however. The clash between this domain and the domain of critical software (Section **??**) is not solved by this approach: QML

```
1  ApplicationWindow {
2      visible: true
3      width: 100
4      height: 50
5      ColumnLayout{
6          TextField {
7              id: theNumber
8              text: "0"
9          }
10         Label {
11             id: theResult
12             text: parseInt(theNumber.text) + 1 ;
13         }
14     }
15 }
```

Listing 2.4: The code of a simple application using QML

programs lack formal semantics that could allow them to be used for critical UIs development. The terseness of the language comes at a price: some important properties of the system are implicitly defined. For example, in Listing 2.4, we see that two seemingly similar lines have various meanings:

- Line 8: `text:"0"` means that the text contained in the `TextField` is equal to "0" *initially*. This equation is enforced only during the initialisation of the UI
- Line 12: `text:parseInt(theNumber.text) + 1 ;` is enforced *all the time* during execution of the UI.

Other aspects of the UI behaviour are left implicit. For example on Line 12, the result of the addition is a number, but the Label displays a text. How is this conversion performed ? What happens if the user enters something which is not a number in the `TextField` ? We see, several aspects are treated implicitly. This is a good thing for non-critical UI development, because it prevents run-time crashes when non-nominal situations are not taken into account by the designer. But the lack of specificity of the behaviour is not a good thing in the field of critical systems, where everything should be under control (Requirement 1 (Formal)).

2.5.2 JavaFX DSLs

JavaFX [65] is a software platform for creating UIs in the Java programming language and other languages based on the Java Virtual Machine (JVM). It used to include a DSL called JavaFX Script that allowed to declaratively create UIs structure. As JavaFX Script was abandoned, various attempts at reviving it using facilities offered by other general-purpose JVM languages have been tried. These languages are Embedded DSL, they are subsets of general-purpose languages, backed by appropriate libraries, sitting on top of Java FX. In this collection, we can cite the following DSLs:

- ScalaFX [66]: Based on the Scala language, which is an multi-paradigm (functional and object-oriented) programming language.
- GroovyFX [67]: Based on Groovy, which is a optionally typed programming language, which makes embedded DSL creation relatively easy.
- ClojureFX [68]: Based on the Clojure language, which is a Lisp dialect, and hence a strongly functional programming language.

These various DSLs are based on the JavaFX API, so they share common genes. Since they are not full blown DSLs but subset of programming languages, they are meant to be understood by programmers and not by other stakeholders (Requirement 3 (Simple)). They provide satisfactory abstractions (Requirement 8 (Abstraction)) for composition (Requirement 6 (Composition)), offering ways to describe the composition of containers as easily as composition of functions. However behaviours have to be specified using the underlying programming language, without any abstraction other than the notion of event.

```
import static groovyx.javafx.GroovyFX.start
start {
  stage(title: 'GroovyFX Hello World', visible: true) {
    scene(fill: BLACK, width: 500, height: 250) {
      hbox(padding: 60) {
        text(text: 'Groovy', font: '80pt sanserif') {
          fill linearGradient(endX: 0, stops: [PALEGREEN, SEAGREEN])
        }
        text(text: 'FX', font: '80pt sanserif') {
          fill linearGradient(endX: 0, stops: [CYAN, DODGERBLUE])
          effect dropShadow(color: DODGERBLUE, radius: 25, spread: 0.25)
        }
      }
    }
  }
}
```

Listing 2.5: The code of a simple GroovyFX UI

2.5.3 Conclusion

Studying existing UI DSLs is an excellent way to identify the powerful constructions that enable easy to understand languages. The limitation of most UI DSLs however is that in their quest for simplicity, they tend to leave out the formalism, and this makes possible to write programs whose behaviour is not obvious, and forget corner cases when programming. Table 2.5 synthesises the conclusion of this section.

	1 Formal	2 Collaborative	3 Simple	4 Projection	5 Traceability	6 Composition	7 Modularity	8 Abstraction	9 Manage	10 Prototyping	11 Verification	12 Flexibility
2.5.1 QML	-	+	++			++	+	+		+		+
2.5.2 JavaFX DSLs	-	+	+			+		+				+

Table 2.5: Requirements vs textual UIs DSLs state of the art

2.6 User Interface markup languages

The family of markup languages, centered around XML [69], gave rise to dozens of dialects to describe the structure (and more rarely some behaviour) of UIs. One of these dialects is HTML, it has a special role because it existed before XML and it has been formalised as a XML dialect since. HTML served as a prototype for many other UI markup languages, most of these languages are used to describe UIs of specific products, and are hence tied to specific vendors, but share the same concepts. One particular markup language is actually in use in the field of critical UIs, so we will explain this one with more details. Overall, all these approaches share many common points:

- Being XML based, they are not really human friendly, as a consequence, they are often associated with graphical editors.
- They describe mostly the structural aspects of UI (appearance and interface with systems), and not the high-level behaviour of UIs.
- A link with a programming or scripting language, which is used to program the behaviour of UIs.
- A run-time engine renders the UI and links it with the program, effectively using the MVVM (Section 2.1.9) or MVP (Section 2.1.6) architecture models.

2.6.1 HyperText Markup Language (HTML)

HTML is the most used markup languages for UI since it is used to describe all websites, at least on the client side. The rendering engine of HTML files is contained in web browsers, and the high-level behaviour of UIs is programmed using the Javascript language, which is executed within web browsers. Hundreds of libraries and frameworks exist to ease the development of HTML + JS applications, some of which will be explored in Section 2.7. But it is possible to use HTML and JS to describe simple applications. Listing 2.6 shows the code of a simple UI description in HTML and JS. As we see, the code is structured in two parts: HTML (static structure) and JS (dynamic behaviour):

- HTML: The body of the HTML document presents what the user actually sees.
- HTML Line 2: The document contains an `<input>` object called `"theNumber"`, which allows the user to enter text, and whose initial value is 0. When this text is changed by the user, a behaviour called `theNumberChanged` is called.
- HTML Line 3: The document contains a `<span>` object called `"theResult"`. This kind of object allows to display text.
- JS Lines 1-4: The behaviour called `theNumberChanged` has the following effect when triggered: Get the value of `theNumber`, add one to it, and set it as the value displayed in `theResult`.
- JS Line 5: Initially, the behaviour called `theNumberChanged` is triggered to set the initial value of `theResult`.

As we see from the example, HTML provides high level abstraction of the structure of UI objects and events (Requirement 8 (Abstraction)). However the language JS itself is a general purpose language, and it provides only low level facilities and no abstraction adapted to the high level specification of UI behaviour. For example, Line 3, referencing the input called `theNumber` involves a long API call. The necessary abstractions are often provided by third-party JS frameworks.

HTML provides separation of concerns: UI structure is described using HTML, UI behaviour is described using JS, and UI Look And Feel (LAF) is described using Cascading Style Sheets (CSS). As a consequence the approach provides good properties to fulfil Requirement 7 (Modularity) and Requirement 6 (Composition) as well as Requirement 2 (Collaborative). Being targeted at non-critical UIs, HTML does not provide the required tools to fulfil Requirement 1 (Formal) or Requirement 11 (Verification). However this provides it with an impressive flexibility (Requirement 12 (Flexibility)).

HTML

```html
<body>
  <input id="theNumber" type="text" value="0" onchange="theNumberChanged()"/>
  <span id="theResult"/>
</body>
```

JS

```js
function theNumberChanged(e) {
  document.getElementById("theResult").innerHTML =
  parseInt(document.getElementById("theNumber").value) + 1;
}
theNumberChanged();
```

Listing 2.6: A simple UI described in HTML and JS

HTML provides abstractions for several kind of UI structures: Containers using , Text and paragraph structuring (using ,<p>, <h1>...), forms and usual widgets (using <form>,<input>), and 2D graphics using (<svg>). Note however that these elements exclusively represent aspects of GUIs, and other kinds of UIs are not supported by HTML directly and have to be programmed using JS when possible.

2.6.2 XML based UI definition

Numerous developers created notations inspired by HTML and based on XML [69]. The notations can be based on different underlying languages, different set of base widgets or controls, or more generally a different environment than web browsers. Among these notations we find:

- User Interface Markup Language (UIML) [70]
- User Interface eXtensible Markup Language (UsiXML) [71]
- eXtensible Application Markup Language (XAML) developed by Microsoft [72]
- XML User Interface Language (XUL) developed by the Mozilla foundation [73]
- Macromedia eXtensible Markup Language (MXML) developed by Macromedia [74]

Dozens of other XML based UI definition language exist, most of them were created during a wave that started in the early 2000s and ended in the early 2010s when many of these different approaches converged back into HTML5. There were variations between these approaches, with some trying to specify only the structural aspects, while others tried to also tackle behaviour or appearance aspects at the same time [75], resulting in different sets of abstractions (Requirement 8 (Abstraction)).

2.6.3 ARINC661 XML Definition File format

A notable XML based UI definition language is the Definition File (DF) format described in ARINC 661 [5], because it is successfully implemented in critical systems. This XML dialect is used to describe the structural aspects and appearance of the UIs presented in Aircraft CDSs. This file format is one of the aspects of the ARINC 661 Standard, and it fits with the associated architecture model described in Section 2.1.12.

ARINC DFs specify precisely all the objects that are displayed in compliant CDSs. A consequence of this precision is that the files are relatively large. The advantage is that even though there are no formal semantics for the language, ambiguity is almost eliminated, which is a good point when

dealing with critical systems (Requirement 1 (Formal)). ARINC also does not allow dynamic instantiation of widgets, which hampers Requirement 12 (Flexibility) but greatly eases potential verification (Requirement 11 (Verification)).

Listing 2.7 presents an ARINC 661 DF for our example application that contains a text input and a text output. We see that the code describes the structure and appearance of the UI at a relatively high level of abstraction, but does not say anything about the behaviour of the UI, which is under the responsibility of the User Application (UA), as explained in Section 2.1.12. We note that the hierarchical decomposition of the UI structure as an application that contains layers that contain widgets is respected. The XML code itself is verbose and difficult to understand, but WYSIWYG tools allows to work on DFs graphically, making it easy to understand by all stakeholders (Requirement 3 (Simple)), while also allowing relatively easy prototyping (Requirement 10 (Prototyping)).

2.6.4 Conclusion

Table 2.6 synthesises the conclusion of this section.

```
<a661_df name="Default" library_version="0" supp_version="5">
  <model>
     <prop name="ApplicationId" value="1" />
  </model>
  <a661_layer name="Default" >
    <model>
      <prop name="LayerId" value="1" />
      <prop name="ContextNumber" value="0" />
      <prop name="Height" value="10000" />
      <prop name="Width" value="10000" />
    </model>
    <a661_widget name="theNumber" type="A661_EDIT_BOX_TEXT">
      <model>
        <prop name="WidgetIdent" value="1" />
        <prop name="Enable" value="A661_TRUE" />
        <prop name="Visible" value="A661_TRUE" />
        <prop name="PosX" value="2073" />
        <prop name="PosY" value="5460" />
        <prop name="SizeX" value="3000" />
        <prop name="SizeY" value="1000" />
        <prop name="StyleSet" value="0" />
        <prop name="NextFocusedWidget" value="0" />
        <prop name="StartCursorPos" value="0" />
        <prop name="MaxStringLength" value="20" />
        <prop name="AutomaticFocusMotion" value="A661_FALSE" />
        <prop name="ReportAllChanges" value="A661_EDB_CHANGE_CONFIRMED" />
        <prop name="Alignment" value="A661_CENTER" />
        <prop name="LabelString" value="0" />
      </model>
    </a661_widget>
    <a661_widget name="theResult" type="A661_LABEL">
      <model>
        <prop name="WidgetIdent" value="2" />
        <prop name="Anonymous" value="A661_FALSE" />
        <prop name="Visible" value="A661_TRUE" />
        <prop name="PosX" value="2575" />
        <prop name="PosY" value="4475" />
        <prop name="SizeX" value="2000" />
        <prop name="SizeY" value="1000" />
        <prop name="RotationAngle" value="0.0" />
        <prop name="StyleSet" value="0" />
        <prop name="MaxStringLength" value="20" />
        <prop name="MotionAllowed" value="A661_TRUE" />
        <prop name="Font" value="default" />
        <prop name="ColorIndex" value="DEFAULT" />
        <prop name="Alignment" value="A661_CENTER" />
        <prop name="LabelString" value="1" />
      </model>
    </a661_widget>
  </a661_layer>
</a661_df>
```

Listing 2.7: An ARINC 661 XML definition file describing a simple application

	1 Formal	2 Collaborative	3 Simple	4 Projection	5 Traceability	6 Composition	7 Modularity	8 Abstraction	9 Manage	10 Prototyping	11 Verification	12 Flexibility
2.6.1 HTML	-	++				+	+	+		++	-	++
2.6.2 XML DSLs		+				+	+	+			-	
2.6.3 ARINC DF	++		+							+	+	–

Table 2.6: Requirements vs UIs markup languages state of the art

2.7 Document Object Model (DOM) based frameworks

As explained in Section 2.6.1, HTML provides satisfactory abstractions for the specification of structural aspects of web-based UIs, but the behavioural aspects are left to JS, which is a general purpose language and, therefore, not adapted to easily define UI behaviour. This gave rise to numerous approaches to ease development of web applications using JS. Detailing even a fraction of the approaches that exist would be impossible. Instead, we focus on a few high profile approaches that cover the spectrum of solutions. We focus on client-side applications, that run autonomously in internet browsers. The common points of these approaches are:

- They are based on JS, the only programming language that can be executed in browsers.
- Interactions happen through manipulation of the DOM [76]. The DOMs is the internal representation of websites inside the browser. It is the View that the user interact with (See Section 2.1.9). These approaches offer ways to manipulate this View and/or bind it to a View-Model.

2.7.1 jQuery: imperative DOM manipulation

jQuery [77] is a library that provides helpers to interact with the DOM, using imperative and event-based programming. The idea is to load a HTML web page, which initialises the DOM in a given state. From there, whenever events are received from the user or the server, callback functions are called. These functions perform mutations on the DOM. When the DOM is modified, the browser automatically modifies what the user sees and the controls he can interact with. Comparison of Listing 2.8 with 2.6 shows that, for the same result, jQuery helps making the imperative JS code that manages web pages a bit simpler as compared to Plain JS, However the architecture and general idea remains the same.

HTML

```
<body>
  <input id="theNumber" type="text" value="0"/>
  <span id="theResult">1</span>
</body>
```

JS

```
$('#theNumber').on('change',function(e){
  $('#theResult').html(parseInt($('#theNumber').value) + 1);
});
```

Listing 2.8: A simple UI described in HTML and using jQuery

The jQuery approach represents the first generation of frameworks that allowed to develop client-side logic for internet applications. This generation of web frameworks is known as Asynchronous Javascript And XML (AJAX) frameworks, and started emerging around 2006 before slowly being taken over by more scalable approaches. While the approach allowed satisfactory amounts of flexibility (Requirement 12 (Flexibility)), it still lacked powerful abstractions (Requirement 8 (Abstraction)) and therefore was not prone to easy composition (Requirement 6 (Composition)). The way of specifying the behaviour of applications imperatively is also not easy to understand for non-programmers (Requirement 3 (Simple)).

2.7.2 AngularJS: Templating

Templating [78] is a common approach in the field of web development, whether it happens on the server side (PHP, Django...) or more recently on the client side. The idea of templating is to specify the View using a static and declarative language (HTML in this case). However, this static description is decorated with placeholders for dynamic data. During execution, the placeholders are filled with data coming from a model, and the output is a view which is displayed to the user.

AngularJS approach [79] is based on client-side templating. Instead of initialising the DOM with a given HTML structure and then modifying it in response to events, as done with jQuery (Section 2.7.1), AngularJS uses a HTML file as a template. Placeholders in the HTML file are represented with double braces: `{{myData}}`. At run-time, these placeholders are filed with the appropriate data, which is provided by Controllers programmed using the AngularJS library. These controllers have a scope, which is made of a set of state variables and a set of functions that alter these state variables. These functions are called by AngularJS in response to user events or server events.

Listing 2.9 shows the code of an example program using AngularJS. The code is split in two parts HTML for the template, and JS for the controller:

- HTML: we see that the structure of the HTML template is very similar to the structure found in the plain HTML approach of Listing 2.6.
- HTML Line 1: The whole HTML document is controlled by a controller named `ResultCtrl`.
- HTML Line 2: The input is bound to a state variable called `theNumber`, and when the user changes the text, a function called `theNumberChanged` should be called.
- HTML Line 3: A placeholder is used instead of an actual value. This placeholder states that the `<span>` element should display the state variable called `theResult`.
- JS Lines 2-8: The controller named `ResultCtrl` is defined.
- JS Line 3: The state variable called `theNumber` is initially 0
- JS Line 4: The state variable called `theResult` is initially 1
- JS Lines 5-7: The behaviour called `theNumberChanged` is defined
- JS Line 6: This behaviour has the effect of setting the variable `theResult` to `theNumber` plus one.

HTML

```
1 <body ng-controller="ResultCtrl">
2   <input ng-model="theNumber" ng-change="theNumberChanged()"/>
3   <span>{{theResult}}</span>
4 </body>
```

JS

```
1 var myApp = angular.module('myApp', []);
2 myApp.controller('ResultCtrl', function ($scope) {
3   $scope.theNumber=0;
4   $scope.theResult = 1;
5   $scope.theNumberChanged = function(newval) {
6     $scope.theResult = parseInt($scope.theNumber) + 1;
7   };
8 });
```

Listing 2.9: Code of a simple application with Angular JS

We see in Listing 2.9 that AngularJS provides higher level abstractions when compared to previous approaches (Requirement 8 (Abstraction)). The definition of controllers as reusable modules also provides satisfactory composition and modularity properties (Requirement 6 (Composition) and Requirement 7 (Modularity)). The templating approach also inherently separates the concern of UI

appearance from the concern of UI behaviour, providing a clear separation line that eases the organisation of collaborative work (Requirement 2 (Collaborative)).

However, a major issue with the templating approach in general is that pure logic-less templates would be very limited in terms of expressive power. For example, they would not able to express the notion of iteration on lists of variable length. As a consequence, most templating engines add some kind of control structures to the templating language. For example, angularJS offers structures called "directives" to express UI behaviour logic in the template language. Angular directives include `ng-repeat` to allow instantiating a part of the template several times in order to render lists, or `ng-if` for conditionals. These control structures may become relatively complex, and this ends up in a situation where the templates (supposed to describe the static structure of UIs) also contains some description of the UI behaviour logic (supposed to be kept separated from the structure). As a consequence, the logic of the UI is spread over two different artefacts: the HTML template and the JS code. These two artefacts are written in different languages, which really hampers the simplicity of the program (Requirement 3 (Simple)). Traceability (Requirement 5 (Traceability)) is also made difficult, because it is not always trivial to know whether some behaviour aspects are implemented in the template or in the JS code. However, good practices and correct methodology may allow to somehow reduce these issues in some cases.

2.7.3 React: Unidirectional data flow

React[80] is a JS library to create UIs. React fits well within the Flux architecture explained in Section 2.1.11. React helps creating the Views of the Flux architecture model. Views are organised hierarchically, with a root view. The approach is called unidirectional because the data flows in one direction: data is passed as properties of the root view, the root view passes some of these properties to its subviews, and so on, until the data reaches the leaves of the view hierarchy, which are concrete HTML elements.

HTML

```html
<div id="container"/>
```

JS (ES7)

```
class SimpleApp extends React.Component {
  state =     { theNumber:0 };
  render()    { return (
    <div>
      <input onChange={this.theNumberChanged}/>
      <span>{this.state.theNumber + 1}</span>
    </div>
  )}
  theNumberChanged(e) { return (
    this.setState({theNumber:parseInt(e.target.value)});
  )}
}
ReactDOM.render( <SimpleApp/>, document.getElementById('container'));
```

Listing 2.10: The code of a simple React Application

Listing 2.10 shows the code of our example application implemented using react. We see that the HTML part is reduced to the minimum. The JS part is a bit more complicated however, but still fairly simple conceptually:

- Line 1: We define a new component called `SimpleApp`.

- Line 2: The internal state of this component consists of one variable, called `theNumber`, initially equal to 0.
- Lines 3-8: The Render method for this component defines how it will look
- Lines 4-7: The component is a container (`<div>` in HTML)
- Line 5: The component contains an element of type `<input>`, which is the HTML name for a textBox. When the user inputs text in the box, the behaviour called `theNumberChanged` is called.
- Line 6: The component contains a label (`<span>` in HTML) which displays a number equal to `theNumber` plus one. Note the similarity between this notation and the templating notation seen in Section 2.7.2. This similarity is only syntactic, because the semantics is different. In this case, the `<span>` denotes a function call, this function takes an argument (`this.state.number + 1`) and returns a UI component (in this case, a `<span>`).
- Lines 9-11: The behaviour `theNumberChanged` consists in changing the state of the component: the value of the state variable `theNumber` becomes the value sent by the `<input>` element.
- Line 13: This imperative function call is necessary, it renders an instance of the Component `SimpleApp` that we defined in the `'container'` defined in the HTML part. It is the only link between the JS file and the HTML file.

As explained in Section 2.1.11, re-rendering the whole app on every change might seem heavy, but appropriate algorithms allow to obtain excellent performance this way.

As we see in the example, both UI structure and logic are implemented purely in JS: this single language is used to express all the aspects of the UI, whether they are structural, static, or dynamic. Actually, even the appearance can be defined in JS, while it is usually defined using a third language called CSS. The fact that a single language is used everywhere is a positive point for Requirement 3 (Simple). It also allows to take advantage of the facilities that are built in this language, enhancing modularity (Requirement 7 (Modularity)) and flexibility (Requirement 12 (Flexibility)) by delegating these aspects to JS. The unidirectional data flow orientation allows excellent composition properties and often does not need any glue code (Requirement 6 (Composition)). These composition properties allow to abstract low level aspects and create components at higher levels of abstraction (Requirement 8 (Abstraction) and Requirement 9 (Manage)).

2.7.4 Conclusion

The flexibility of the HTML + JS couple gave rise to many innovative ideas, some of which being actually inspiring when it comes to developing approaches for embedded critical interactive systems. Table 2.7 summarises the conclusion of this section.

	1 Formal	2 Collaborative	3 Simple	4 Projection	5 Traceability	6 Composition	7 Modularity	8 Abstraction	9 Manage	10 Prototyping	11 Verification	12 Flexibility
2.7.1 jQuery			-			-		-				+
2.7.2 AngularJS		+	-		-	+	+	+				
2.7.3 React			+			++	+	+	+			+

Table 2.7: Requirements vs DOM based frameworks state of the art

2.8 Functional Reactive Programming

Functional Reactive Programming (FRP) [81] is a declarative programming paradigm based on Functional Programming (FP), with added notations for values that change over time, known as Reactive Programming (RP) [82]. In FRP, time-varying values are called signals. Signals represent any value that change with time. There are different variations around FRP, such as Classical, Event-Driven or Arrowised FRP. The main idea is that FRP is based on the functional programming paradigm, which allows to describe programs in a declarative way. But instead of representing one computation, that happens at one point in time, FRP programs represent a computation that happens at each point in time, and instead of dealing only with single values, FRP can also deal with signals, which are infinite sequences of values. FRP involves facilities for dealing with signals, for example *lifting* functors that allows to transform functions that acts on values to functions that acts on signals. FRP based approaches have several common points:

- Being based on functional programming, they are declarative, and composition of interactions is expressed simply, in terms of composition of functions (Requirement 6 (Composition)), which reduces the amount of glue code and might simplify the programs (Requirement 3 (Simple))
- Unidirectional data flow. Functions are objects that take several parameters (whether it is through currying in languages like Haskell [83] [84] or by taking several arguments in languages like C) and output a single value. This means that programs describing UIs are functions that takes the UI input as arguments and return the UI output as a result.

2.8.1 Reactive banana

Reactive Banana[85] is a library to create UIs using Haskell and the FRP paradigm. It is one of the most used FRP libraries for the Haskell programming language. Being based on Haskell, its syntax is not straightforward for non-programmers, and even for programmers not used to functional programming. As a consequence, it does not give a satisfying answer to Requirement 3 (Simple). However, it is based on relatively simple abstractions adapted to the description of UIs: events, behaviours (continuous flows of data), and combinators (higher order functions that allows to work on events and behaviours).

Reactive banana programs simply represent a network (Directed Acyclic Graph (DAG)) of functions, with user and machine input as source nodes, and user and machine output as sink nodes. However some additional code is needed to plug actual widgets to the inputs and outputs of this network, which increases the complexity of the code. Listing 2.11 shows the code for our simple application example, which takes an input (*theNumber*), and outputs this number incremented by one (*theResult*).

- Lines 3-4: Structural aspects of the UI are described. It is made of a `frame` whose title is `"MyApp"`. This frame contains an `entry` widget called `theNumber`, which initially contains the text `"0"`. The frame also contains a `staticText` widget called `theResult`
- Lines 6-17: Specification of the network graph that represents the behaviour of the UI as a composition of nodes.
- Line 8: The node `btheNumber` receives the text entered by the user in the text box `theNumber`.
- Line 12: The result is obtained by applying the function `(+)` to two incoming signals: the first one is `(readNumber x)`, which transforms the input text coming from `btheNumber` into a a number. The second signal for the sum is `(Just 1)`, which represents a number that is constantly defined and equal to 1.
- Line 15: If the result number cannot be computed (possibly because the user entered a text which is not a valid number), then the result text defaults to `"N/A"`.

- Line 17: Finally, the DAG has a sink, which takes the result value and sets it as the text displayer by the widget `theResult`.
- Lines 19 - 20: Once the network is defined, it is compiled and executed.

```
1  main = start $ do
2      window       <- frame                [text := "MyApp"]
3      theNumber    <- entry       window   [text := "0"]
4      theResult    <- staticText  window   []
5
6      let
7          networkDescription =  do
8              btheNumber  <- behaviorText theNumber ""
9              let
10                 result = f <$> btheNumber
11                     where
12                     f x = liftA2 (+) (readNumber x) (Just 1)
13
14                 readNumber s = listToMaybe [x | (x,"") <- reads s]
15                 showNumber   = maybe "N/A" show
16
17             sink theResult [text :== showNumber <$> result]
18
19     network <- compile networkDescription
20     actuate network
```

Listing 2.11: The code of a simple application using the Reactive banana Haskell library

As we note in Listing 2.11, Haskell and Reactive Banana have a powerful typing system, which forces UI developers to explicitly specify all the aspects of the UI behaviour, including the non-nominal situations. This makes the approach very formal (Requirement 1 (Formal)), and some level of verification (Requirement 11 (Verification)) is actually performed by the type system itself: the program will not compile if some behaviours are not specified correctly. However, this strong typing means that all aspects of UI behaviour have to be described precisely before the UI can even compile. This is not a good point for prototyping (Requirement 10 (Prototyping)), since it prevents testing UIs which are not totally polished.

A good point is that the interaction logic of the UIs is defined using the simple concept of DAG, which could simplify its analysis. This approach also benefits from the interesting composition properties provided by FP: behaviours can be composed as simply as functions, a good point for Requirement 6 (Composition) and Requirement 7 (Modularity).

2.8.2 Elm

Elm [86] is a concurrent Functional Reactive Programming (FRP) language for creating web applications. The react compiler generates JS code that runs in browsers. Elm provides only a single notion for values that change over time, and they are called Signals. All input and output of interactive applications are represented as signals. Elm prescribes an architecture similar to the Flux architecture model (Subsection 2.1.11) [87].

Listing 2.12 Shows the code of a simple Elm application, which is divided in three parts: a View function, a Model (in the MVC sense), and an Update function:

```
1  model =
2    "0"
3
4  view address model =
5    div [] [
6      input [ on "input" targetValue (message address) ] [],
7      span  [] [ text ( toString (map ((+) 1) (toInt model))) ]
8    ]
9
10 update action model =
11   action
12
13 main = App.start { model = model, view = view, update = update }
```

Listing 2.12: The code of a simple application using elm

- Line 1-2: The Model for this application is simply a string. It is initially the string `"0"`.
- Line 4-8: The View is defined as a function that takes two parameters: an `address` to which events should be sent, and a `model`. The view outputs a signal whose values are HTML elements to be displayed to the user.
- Line 5: The view is made of a container (HTML `div`)
- Line 6: The container contains an `input` element, which sends messages when changed by the user. The messages contain the value input by the user, and are sent to the `address`.
- Line 7: The container also contains a `span` element, which displays the result to the user. The result is computed by adding one to the number represented in the model.
- Line 10-11: The Update function here is really simple. When an action is sent by the view, the update function is called and should output a new model. In our case, there is only one possible action: the user changed the input value. This action has a simple consequence: the model then becomes the new value input by the user.
- Line 14: The application is created and started by calling the function start.

As we saw with Haskell in Subsection 2.8.1, Elm is strongly typed and the typing system forces developers to explicit all aspects of the application behaviour, including non-nominal cases. This is enforced using data types such as `Maybe` and `Result` which allow to explicitly represent computations that could fail or data that is missing. As a consequence, authors of Elm claim that it practically eliminates runtime errors, because all potential errors are detected by the type system during compilation. This demosntrates that the type system of Elm is a step in the right direction for Requirement 1 (Formal) and Requirement 11 (Verification). We also see that Elm presents a simple, clear and concise syntax (Requirement 3 (Simple)), partly thanks to the type inference system. As shown in [87], Elm programs benefit from excellent composition properties thanks to the declarative orientation of functional programming (Requirement 6 (Composition) and Requirement 7 (Modularity)).

2.8.3 Conclusion

FRP provides extremely interesting new ways of thinking about programming interactive systems. However it still lacks simplicity in some cases, but approaches such has Elm are definitely going in the right direction. Table **??** summarises the conclusion of section.

	1 Formal	2 Collaborative	3 Simple	4 Projection	5 Traceability	6 Composition	7 Modularity	8 Abstraction	9 Manage	10 Prototyping	11 Verification	12 Flexibility
2.8.1 Reactive banana	+		-			++	+			-	+	
2.8.2 Elm	+		++			++	+			+	+	

Table 2.8: Requirements vs FRP state of the art

2.9 Graphical modelling of Interactive Systems behaviour

Many approaches allow to graphically model the structure of Graphical User Interface (GUI). Such tools are common place, and exist for many programming languages, as explained in Section 2.4. Representing UI behaviour graphically on the other hand is much less common, but present some interesting advantages. This section presents visual languages that allow the specification of the behaviour of Interactive Systems (IS). These approaches are interesting because the graphical view often allows to have an overview of the general interaction structure at a first glance. Interestingly, most graphical approaches for UI specification present similarities:

- Representation in the form of graph-like structures, with nodes and edges.
- Basic nodes whose meaning are defined in external programming or scripting languages
- Additional coding is often needed for functional non-UI aspects.
- Regarding the semantics of the diagrams, there are two possibilities:
  - Similarity with Petri nets diagrams, where objects circulate asynchronously on the edges between nodes. Nodes represent basic elements, and their connections represent the glue described in Requirement 6.
  - Similarity with State machines diagrams, where nodes represents states of the systems, and edges represents transitions.

2.9.1 Interactive Cooperative Objects

Presented in [88], this approach is based on objects called Interactive Cooperative Objects (ICOs). These objects are made of four components:

- Data structure: A set of attributes, each having a name and a type. Attributes can have simple types or class types.
- Operations: A set of methods, each having a name, input and output parameters, and code expressed in any algorithmic language.
- Control structure: A representation of the behaviour of objects, in the form of Object Petri nets.
- Presentation: A hierarchy of widgets which are the graphical representation of the ICOs. It is what the user can see and interact with.

Figure 2.19 shows an example of an ICO control structure expressed in the form of a Petri Net. The graphical notation for the behaviour of ICOs allows to have an overview of the system behaviour with a quick look at the diagram, even though getting into the detail is more difficult (Requirement 3 (Simple)). Relying on Petri nets allows this approach to link properties of ICOs to mathematical properties of Petri nets, which are a well studied mathematical objects, providing a sound formalisation (Requirement 1 (Formal)). This allows to verify some properties of ICOs such as Liveness, boundedness, or mutual exclusion (Requirement 11 (Verification)).

On the other hand, ICOs are inherently non-deterministic, which may make certain properties such as liveness or Worst Case Execution Times (WCET) difficult to ensure. Another limitation is that Petri nets used to describe the behaviour of ICOs can become relatively complex, which makes complex systems difficult to comprehend by non-programmers (Requirement 3 (Simple)). ICOs Provide interesting abstractions for the field of UI development (Requirement 8 (Abstraction)), but all these abstractions have to be based on, or to be embeddable in, Petri nets. This limits the flexibility of the approach (Requirement 12 (Flexibility)). However, works such as [89] managed to describe various kinds of UI modalities using this formalism, and ICOs have proven over the years to be a manageable way of describing UI behaviour [90].

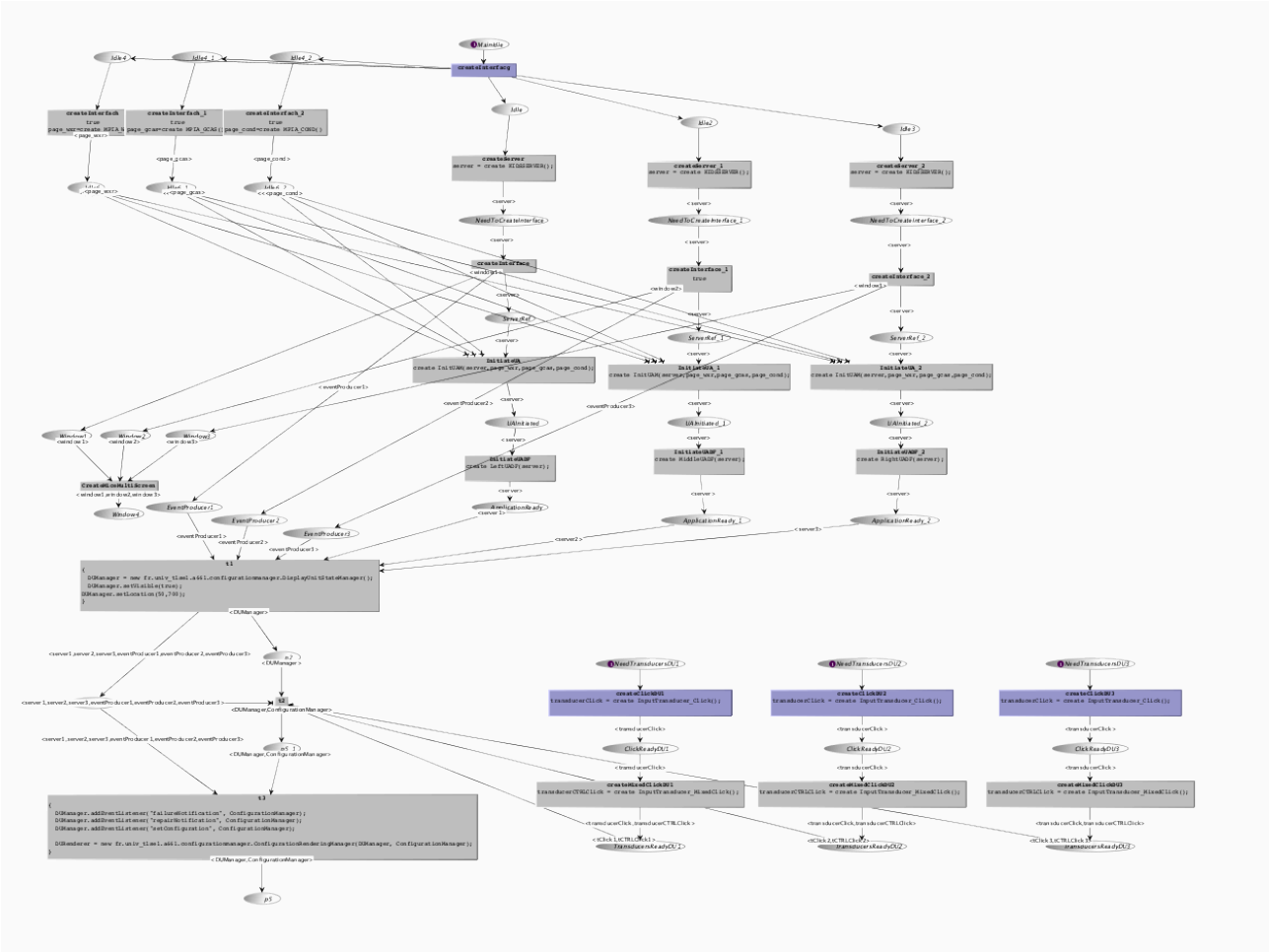


Figure 2.19: A system described using ICOs

2.9.2 Formal Interaction Logic Language

Formal Interaction Logic Language (FILL)[91] is a formalism to describe the behaviour of UIs using a graphical notation which integrates three main families of nodes:

- Operation nodes, which represent either system operations (interaction with the machine), interaction logic operations (behaviour of the UI), or input and output operations (interaction with the user). These nodes are represented as rectangles which can have various numbers of input and output ports.
- Business Process Modelling Notation (BPMN) nodes, which rely on the BPMN notation, and represent control or data flow branching operators. They allow to send data to various paths depending on conditions, or execute several behaviours in parallel.
- Proxy nodes, which represent events coming from widgets, which triggers behaviours in the system.

FILL systems denote Reference Petri nets. Reference Petri nets are similar to coloured Petri nets where colours denote references to objects, which can be external objects (e.g. Java objects) or other elements of the Reference Petri net. All FILL nodes have an underlying representation as Reference Petri Nets. Figure 2.20 shows an example FILL system, with its mapping to a Petri net:

1. The first node “Event” is a Proxy node, it represents an input from a widget.
2. The second node is a System operation node. We see on the right that it denotes an asynchronous function call.

3. The third node is a BPMN node, it denotes a branching operator. Depending on a condition on the data they represent, tokens will go on the left branch or the right branch.
4. The last node is a terminator node, which means that tokens that are sent to it are consumed.

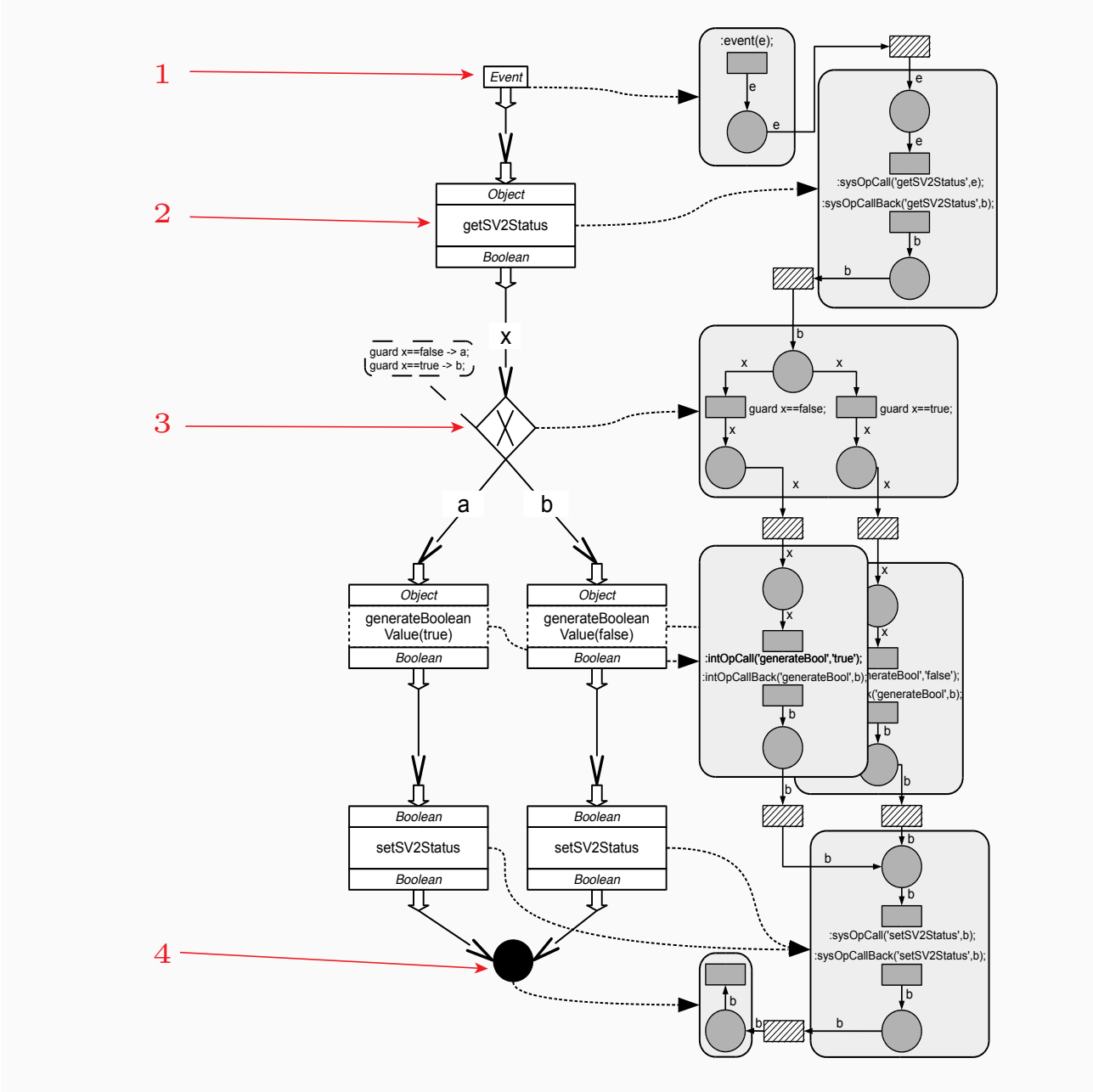


Figure 2.20: A FILL system (left) with its representation as a petri net (right)

FILL provides a formal model of UIs (Requirement 1 (Formal)), since it denotes Reference nets. The graphical notation provides a way to create prototypes relatively easily (Requirement 10 (Prototyping)). It is also highly understandable, borrowing from notations which are intended for the general public, and not only programmers (Requirement 3 (Simple)). At the moment, FILL lacks a way to create new abstractions (Requirement 9 (Manage)), for example by allowing UI designers to create new blocks as a composition of other blocks (Requirement 6 (Composition) and Requirement 7 (Modularity)). However, this limitation is acknowledged by the authors and might be solved in the future.

A limitation of FILL is the same as ICOs: being based on Petri nets and tied to the notion of

GUI, the model lacks some flexibility when it comes to representing new kinds of Interactive Systems (Requirement 12 (Flexibility)). The event-based bias of Petri nets makes it difficult to represent continuous data flows: they are represented using ticker events that continuously trigger updates at fixed time intervals, an approach whose scalability is debatable. Finally, as for ICOs, being based on Petri nets eases some aspects of verification (Requirement 11 (Verification)), but makes other aspects difficult to verify.

2.9.3 Max/MSP

Max/MSP [92] is a commercially available modular and graphical environment mostly dedicated to the creation of digital and interactive art, including music, video and immersive installations. Max/MSP provides a graphical environment in which basic blocks that can be connected in order to generate complex behaviours and processing pipelines. These blocks can communicate in two ways. The first and main mode is called Max, it provides asynchronous communication between blocks and allows complex data types. The other mode is called MSP, it provides synchronous communication at audio sampling rate (several kHz), but only allows basic data types such as integers and floats to be exchanged. The Max mode provides specific functionalities for the creation of UIs:

- The first is a set of basic UI blocks that represents typical UI components such as labels, buttons, panels... These elements are able to communicate asynchronously with each other and with functional non-UI blocks in order to generate complex behaviours.
- The other is the ability to present Max programs in two modes. While the “patching mode” allows to visualise all blocks, including non-UI blocks and their connections, the “presentation mode” displays only UI blocks without clutter. The transition between these two modes is continuous and allows a very clear view of the link between the “code” and the final UI, which are represented in the same way.

The main advantage of Max/MSP is that it allows fast prototyping of UI, by breaking the barrier between the code and the final result, which are represented and mixed in the same model, providing great prototyping capabilities (Requirement 10 (Prototyping)). However, the asynchronous event-based nature of communication between components can give rise to unexpected complex behaviours. Max/MSP allows to create nodes that contains patches, allowing to create new reusable blocks, an elegant answer to Requirement 7 (Modularity). Composition of nodes is easily done by connecting them, which allows to have overview of systems behaviour (Requirement 3 (Simple)). However the amount of connections between components can quickly become problematic if no care is taken. The solution to this is to encapsulates parts of the model within modules.

Max has not been developed with verification in mind, so verification of these models seems a far fetched option (Requirement 11 (Verification)). The execution model is based on a scheduler with several event queues, which can lead to extremely large state spaces. Max/MSP suffers from the same verification problems as petri-net based approaches, but its lack of formality does not allow to verify properties that could be checked with ICO.

However, a tool called VVVV[93] provides an environment similar to Max/MSP, but with synchronous semantics, which forces to explicit and formalise some aspects, such as feedback loops, coming closer to formal models such as Scade (Requirement 1 (Formal)), but still lacking mathematical formalisation.

2.9.4 Apple storyboards

Apple storyboards [94] are a way to describe the high level behaviour of applications by describing the hierarchy of screens encountered by the user. Architecturally, they allow to describe the “coordinating

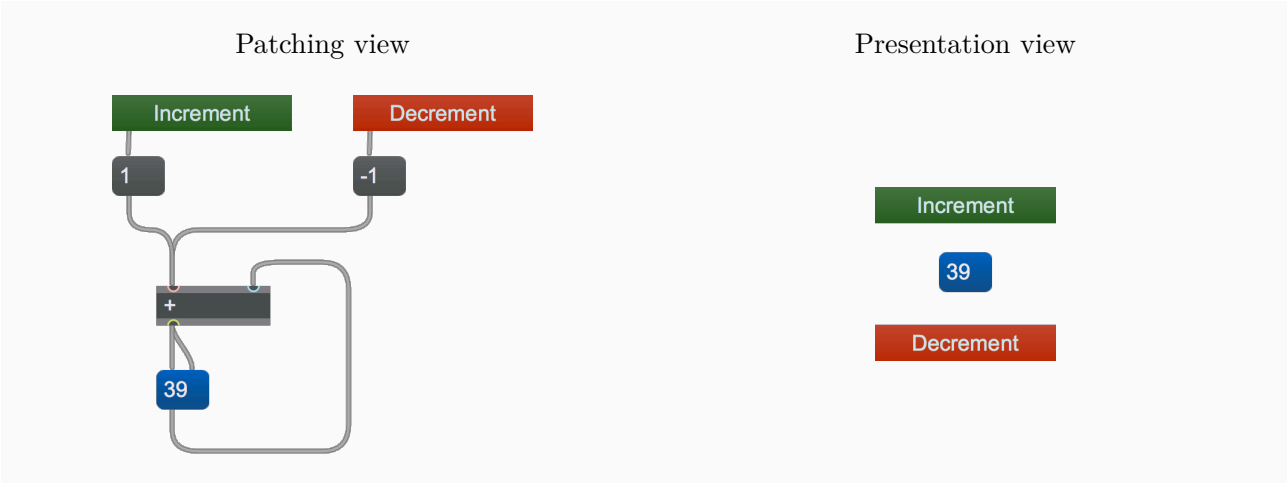


Figure 2.21: An example of Max/MSP program with the "patching view" (left) and "presentation view" (right)

controllers" of the Cocoa MVC variant explained in Section 2.1.7. There is usually one of these controllers per application, and they manage the activation of different screens of the application.

An interesting aspect of storyboards is that they allow to have a high-level graphical overview of the UI appearance and behaviour, as shown in Figure 2.22. This explicit and clear view helps stakeholders to exchange about the UI early during the development cycle (Requirement 2 (Collaborative) and Requirement 3 (Simple)). Storyboards also allow to generate a good part of the code of the "coordinating controllers" automatically. Therefore prototypes of application, containing the different views and their transitions can be generated really quickly, hence a big improvement in development efficiency (Requirement 4 (Projection) and Requirement 10 (Prototyping)).

Limits of storyboards are that they are platform specific, and enforce the use of a standard set of interactions, therefore not encouraging experimentation of innovative interaction patterns (Requirement 12 (Flexibility)). This later limitation is however a good thing in the context of consumer products with thousands of developers, because it ensures a relative homogeneity in the behaviour of applications, even when they come from various developers. This way, end users are not disoriented by unusual behaviours.

2.9.5 Conclusion

Its is difficult to conclude about the understandability of graphical notations. They tend to be very easy to understand for small systems, but they often become too complex to understand when used on large or complex systems. This section provided several very interesting ideas, as summarised inTable 2.9.

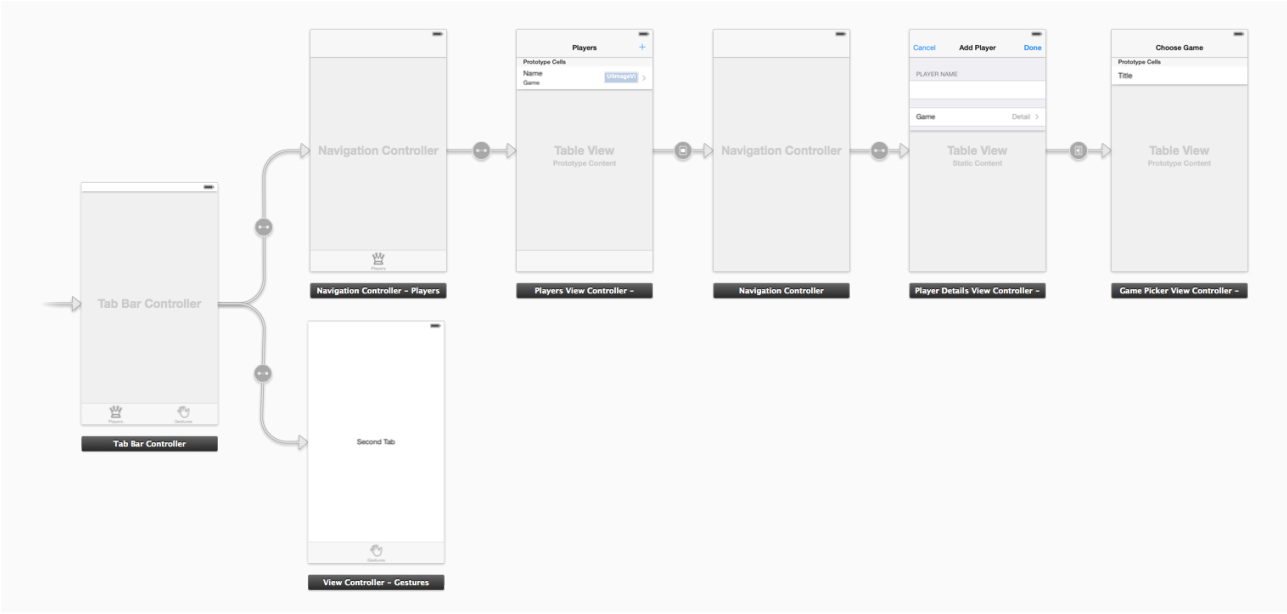


Figure 2.22: An example of Apple storyboard

	1 Formal	2 Collaborative	3 Simple	4 Projection	5 Traceability	6 Composition	7 Modularity	8 Abstraction	9 Manage	10 Prototyping	11 Verification	12 Flexibility
2.9.1 ICOs	+							+			+	-
2.9.2 FILL	+		+				-		-	+	+	-
2.9.3 Max/MSP	)						+			+	–	+
2.9.4 Storyboards		++	++	+						++		–

Table 2.9: Requirements vs graphical modelling of UI behaviour state of the art

2.10 Task models

Task models are models that represent tasks as a composition of basic tasks, by adding various constraints between tasks, such as temporal or dependency constraints. User task modelling is an important mean for developers of HCI systems. The ISO 9241-11:1994 standard defines usability as the extent to which a product can be used by specified users to achieve specified goals effectively, efficiently and with satisfaction in a specified context of use. In this standard, the context of use has four main components: user, tasks, platform and environment.

2.10.1 Concur Task Trees

Concur Task Tree (CTT) is a notation first presented in 1997 in [95]. It is a framework an a graphical notation to describe task models as a hierarchical composition of user tasks, computer tasks and interaction tasks. The main features of CTT are:

- Focus on high level activities, not detailing low-level implementation details.
- Hierarchical task structure, where tasks are decomposed into smaller tasks, until the tasks are basic enough
- Graphical syntax, with tasks models depicted in the form of trees with relationships between sibling elements.
- Temporal operators between siblings that, when combined with the hierarchical view, allow to specify complex task models.

CTT is a good starting point to analyse the task models for a given application, and the graphical view provides an overview of the system that is understandable by all stakeholders (Requirement 3 (Simple)). Each task may be linked with one parent (a more general task, which the current task is part of), several children (sub tasks that compose the current task), and two siblings: the previous task and the next task. Temporal operators are the process algebras operators of Language Of Temporal Ordering Specification (LOTOS), which gives the approach a formal definition (Requirement 1 (Formal)). These operators define temporal ordering properties between consecutive sibling tasks. Temporal operators can only be applied between *consecutive* sibling tasks, which gives a linear, ordered organisation of sub tasks. This linear organisation of tasks can make some configurations more difficult to express usingCTT, when compared to other approaches such as BPMN, which allow a non-linear, more general organisation of relations between tasks. An interesting point of CTT however is that this linear organisation allows to represent several abstraction levels on the same diagram (Requirement 8 (Abstraction) and Requirement 9 (Manage)). Similar frameworks such as [96] or[97] were created, in order to respectively improve tooling support, and support multimodal interfaces; proving that the approach is flexible (Requirement 12 (Flexibility)).

Figure 2.23 presents a CTT model for a simple hotel room reservation application. In this task model, the user can either select a single room or select a double room, and then he can make a reservation: the computer shows him availability of rooms and then he can select a room.

2.10.2 Enhanced Operator Function Model

EOFM[98] is a formalism which allow to hierarchically represent task models. It presents some superficial similarities with CTTs, but it is different in various points:

- EOFM represents only the tasks of the Human operator, while CTT mixes different categories of tasks such as user tasks, machine tasks, interaction tasks. EOFM represent tasks in which the human user is involved: user tasks and interaction tasks.

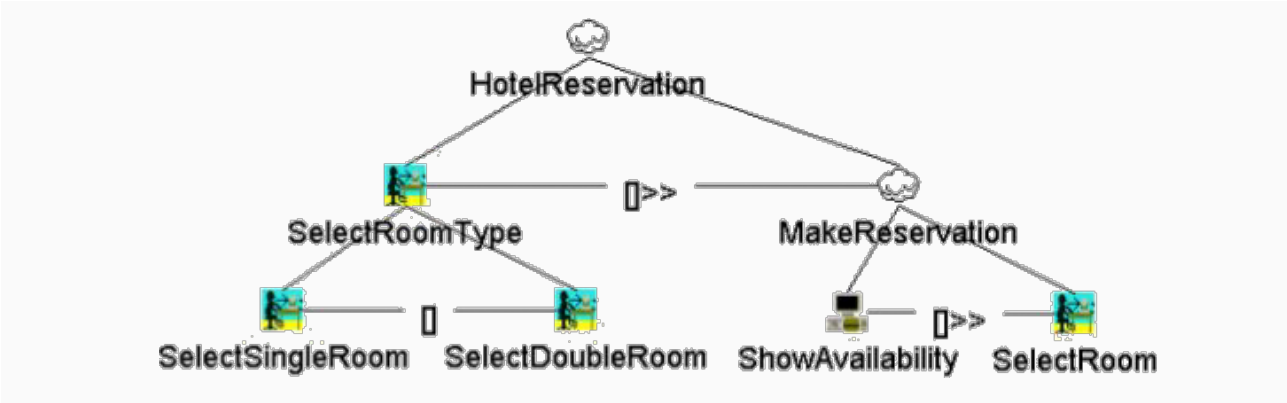


Figure 2.23: A Concur Task Tree (CTT) task model

- EOFM has a different set of operators between nodes. While CTT operators are process algebra operators, EOFM decomposition operators are 9 ad-hoc operators [99]. They include *ord* for ordered sequential execution, *xor* when only one of several tasks must be executed, *or_seq* for unordered sequential execution...
- EOFM includes pre and post conditions that can be linked to tasks. A task can only be executed if its pre condition is true and it stops when its post condition becomes true.

Thanks to the pre and post conditions, EOFM has slightly more expressive power than CTT, while still remaining a simple way to clearly define task models (Requirement 3 (Simple)). EOFM task models are formal (Requirement 1 (Formal)), and were used to generate Software Analysis Laboratory (SAL) code in order to check safety properties of systems composed of a machine and a human operator, [22], including cases were the human operator makes errors [100] (Requirement 11 (Verification)).

Figure 2.24 presents an EOFM diagram describing the task of responding to a traffic light. The task is first decomposed in three different subtasks whose execution is dependent on preconditions: $WaitTillCloser$, $BrakeStop$ and $RollStop$. The $RollStop$ task is further decomposed in two subtasks $InitiateRoll$ and $StopAtIntersection$, which are themselves decomposed further again. Yellow arrows represent pre conditions on each tasks, while Pink arrows represent post conditions.

2.10.3 Petri nets

Several approaches model Task models using Petri nets [101], which are a convenient way to formalise concurrent task models. Some of these approaches are more oriented toward modelling of business workflows [102], some are more oriented toward UI task models[103]. In this section, for example, we will describe the approach presented in [104].

This approach describes task models using Object Petri Nets [105]. A task model is represented by a single Petri net, which is structured by grouping nodes in order to model two hierarchy levels of task decomposition:

- The Cognitive Processing Unit (CPU) represents the main task, the higher level of the task decomposition hierarchy.
- Operational Processing Units (OPU) represent sub tasks. When the CPU arrives on a complex task, it sends a token to the appropriate OPU, and waits for a token to come back from the OPU before it can proceed to the next task.

Figure 2.25 presents a Petri net representing a complex task, described with this approach. The center part is the CPU which represents the main task, while the other parts are OPUs, which represent

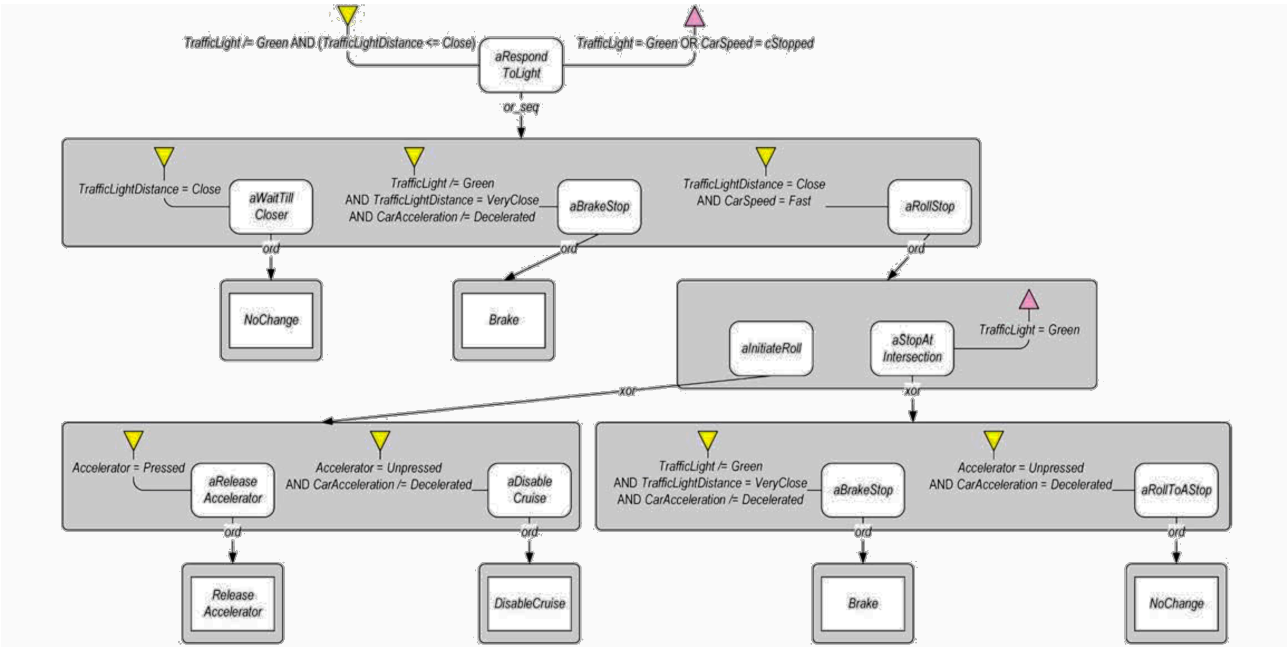


Figure 2.24: A Enhanced Operator Function Model (EOFM) description of a task model

various subtasks. We note that Petri net are more expressive than previous aspects such as CTTs, they allow to express complex task structures.

Petri nets allow to easily have an overview of the structure of the task model (Requirement 3 (Simple)). For example, sequential tasks will "look" different to parallel tasks (horizontal stacking instead of vertical stacking of nodes on the diagram). However, we note that without further decomposition or abstraction facilities, Petri net based approaches can lead to complex Petri nets which can be difficult to comprehend. Authors of [104] acknowledge that there is a need for ways to create abstractions (Requirement 8 (Abstraction)) in order to simplify Petri nets for large applications by allowing modularity (Requirement 7 (Modularity)). In the case of the approach presented in Figure 2.25, we see that the two composition levels are segregated and managed manually by placing nodes in appropriate groups, an approach which cannot scale to larger hierarchy of compositions (Requirement 6 (Composition)).

2.10.4 Conclusion

This section is an occasion to realise that task modelling is not as simple as it seems, and can involve similarities with UI behaviour modelling (For example, Petri nets can be used for both). Table 2.10 summarises the conclusion of this section.

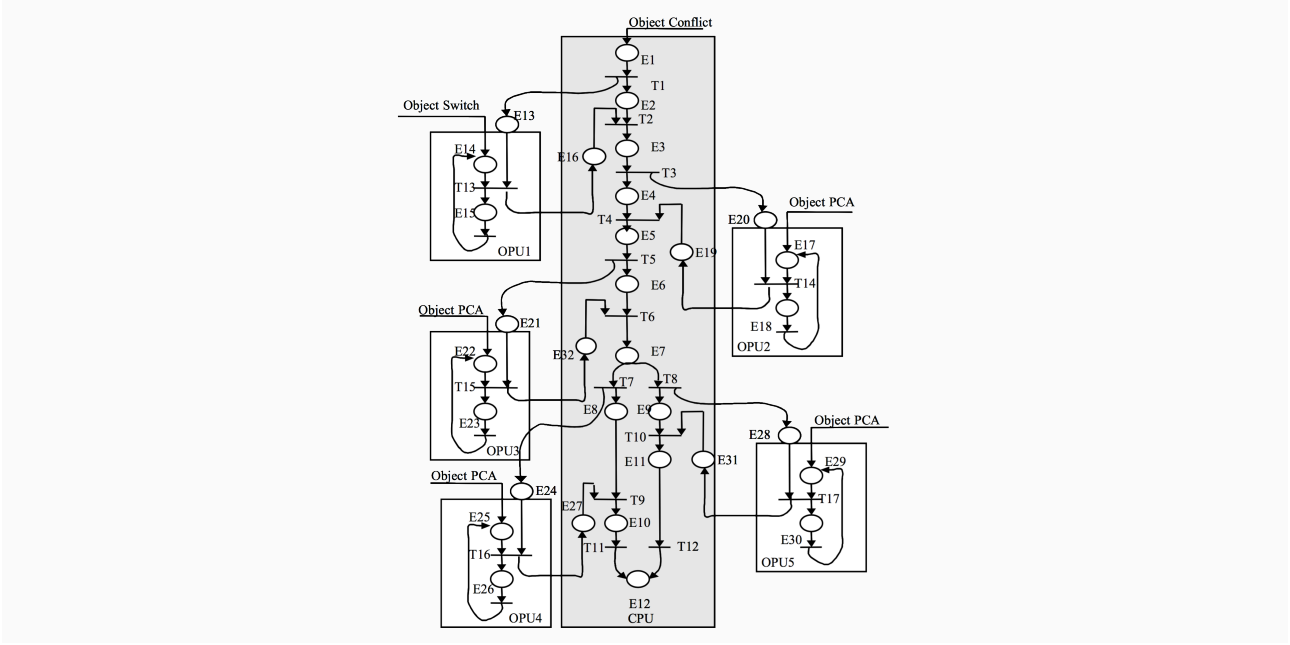


Figure 2.25: Object Petri nets to describe user tasks as in [104]

	1 Formal	2 Collaborative	3 Simple	4 Projection	5 Traceability	6 Composition	7 Modularity	8 Abstraction	9 Manage	10 Prototyping	11 Verification	12 Flexibility
2.10.1 CTT	+		++					+	++			+
2.10.2 EOFM	++		+								++	
2.10.3 Petri nets	+		+			-	-	-				+

Table 2.10: Requirements vs task modelling state of the art

2.11 Verification techniques applied to critical interactive systems

In this section, we will examine a few examples of verification techniques applied to interactive systems. These techniques are answers to Requirement 11 (Verification), and this imply that they are formal (Requirement 1 (Formal)). They are based on the use of general-purpose verification techniques or tools in order to verify critical UI systems. A relatively large number of approaches have approached this problem with some success. In this section, we will only present a small subset of these approaches,

2.11.1 Model checking UIs

These approaches aim at verifying properties of UI systems. As expressed in Figure 2.26, the general architecture of these approach is based on four main activities, two of which are usually automated. These activities are:

- Modelling: From the description of a UI in a potentially non-formal language, an expert models the UI in a formal language (Requirement 1 (Formal)) . This step is often difficult because care has to be taken so that the resulting model is a faithful representation of the UI. The UI model also has to be relatively abstract so it does not lead to combinatorial explosion during model checking.
- Express Properties: From the description of the UI, the expert defines logical properties that have to be verified in order to ensure that the UI verifies required User interaction properties. The difficulty of this step is two fold. First, these logical properties can be complex to express in the chosen formalism. Second, it is difficult to prove that the set of logical properties is sufficient to ensure a correct operation of the UI.
- Transform: This activity is optional, and often automated. From the formal model of the UI, a tool generates code that can be input to a model checker tool.
- Model checking: This activity is performed automatically by a model checker tool, which checks that the logical properties are verified by the modelled UI in all possible states of the UI system. If the UI does not verify the required properties, the result is made of counter examples. These counter examples have to be interpreted manually in order to trace their cause back to the original UI specification.

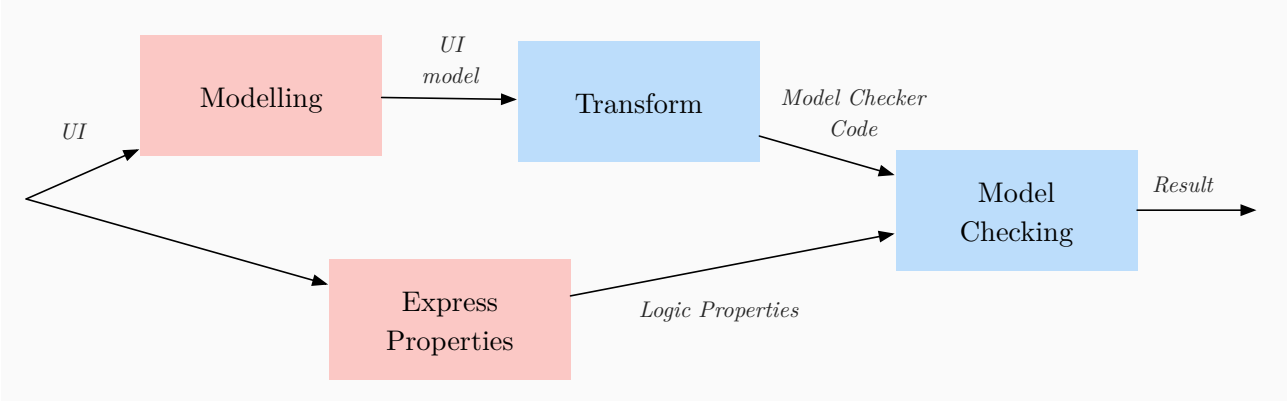


Figure 2.26: General workflow for model checking of UIs

Even though these approaches are difficult and require expert knowledge, they have been successfully applied, using several model checking tools. Approaches used the model checker NuSMV [39], Lesar [106] [107], Spin [108], SAL [22] have been applied with success. Here we are going to detail a successful example of this approach: the IVY workbench, which is based on the NuSMV model checker.

IVY workbench is a tool for developing and verifying models of critical UIs. This approach has

successfully been applied to several real use cases [109] [110], proving that it fulfils Requirement 11 (Verification). In this approach, the UI model is specified in the form of York Interactors (Subsection 2.3.2), using structured Modal Action Logic (MAL)[111] [112]. MAL is a modal logic that incorporate the notion of action. The properties to be checked are expressed using Computation Tree Logic (CTL). The transformation of UI models into model checker code is performed automatically by the IVY workbench. Model checking is performed by the NuSMV model checker [20].

Systems with state spaces as large as 10^9 spaces have been checked using the IVY workbench, but combinatorial explosion is a common problem when dealing with model checking of complex systems. As a consequence, expert knowledge is necessary during the modelling phase in order to deal with actual UI systems, because naive modelling of such systems would lead to excessively huge state spaces. IVY acts as a relatively thin layer above NuSMV, so the abstractions it presents are very relevant for model checking UI, but not as much in the general field of UI. Being easily understandable by non-experts is not the primary objective of the MAL syntax (Requirement 3 (Simple)): this is another reason why modelling has to be performed by experts.

The IVY workbench is interesting because it is probably the most integrated tool to perform model checking on UIs. Extensions to the workbench were created, proving its flexibility (Requirement 12 (Flexibility)). For example, a tool allowing prototyping was presented in [113] (Requirement 10 (Prototyping)).

2.11.2 Model discovery, and theorem discovery

This approach, presented in [114], is based on two automated techniques which allow to verify properties of systems which have a finite state space. As expressed in Figure 2.27, the workflow for this approach consists of a sequence of activities:

- Instrumentation: During this phase, a developer manually adds code to the UI codebase in order to allow model discovery. It is noteworthy that this technique is applied on the actual implementation of the UI and not on a model of it (he goal is to discover the model). This activity results in a program which consists of the UI and an engine that can perform Model discovery.
- Model discovery: It is a dynamic analysis technique which consists in simulating all the possible user actions in order to automatically explore all possible states of the system. Working on the actual implementation of the UI allows to capture situations which could have been missed by a model. Model discovery results in the description of a State Machine whose states are states of the UI system, and transitions represent user actions.
- Theorem discovery is a technique for analysing the discovered state-spaces. It consists in systematically computing and comparing the effects of sequences of user actions.

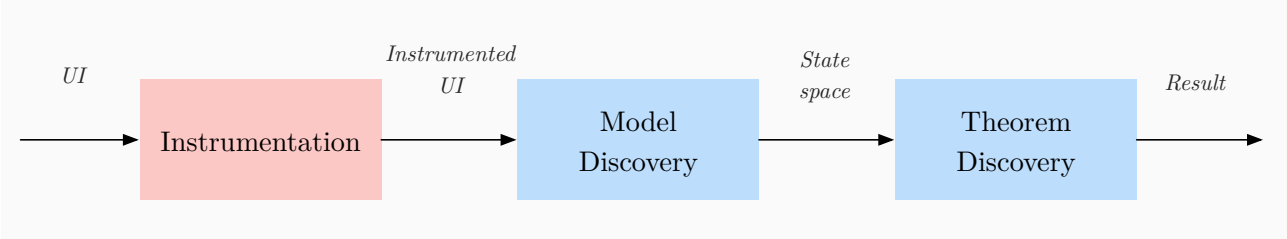


Figure 2.27: Workflow of model discovery and theorem discovery

Insights can arise from theorem analysis: Finding which sequences of user actions are almost equivalent and considering the meaning of sets of such equivalences allows to find interesting properties of the interactive system. In the cases where equivalence classes are found, examples of action sequences

where the equivalence is not respected are an interesting result because they represent situations which might cause problems for users. Here are some examples of some interesting patterns can be discovered:

- Idempotent actions: Actions whose repeated application is equivalent to a single application. For example, turning a device off is idempotent because turning a device off twice does nothing more than turning it off once.
- Action groups: Actions which affect the same variable. For example turning on and turning off a device are actions that belong to the same group, they act on the same state variable.
- Inverse actions: Actions which, if performed one after the other, do not change the state of the system. For example incrementing a value and decrementing a value are inverse actions most of the time.
- Undo action: Action which always reverses the last user action of a certain kind. For example, clicking the undo button in desktop application is a Undo action, but it does not reverse the last cursor movement.
- Safe actions: Action which always leads to the same state. For example, Pressing the AC key on a pocket calculator is a safe action, it leads to a known state in all cases.

A limitation of the presented approach is that it is based on the explicit discovery of the whole state space of FSMs, a form of model checking. As a consequence, it can suffer from combinatorial explosion on large system. Another limitation of this approach is that it only takes into account systems in which only user actions can trigger state changes. In many actual use cases, external events can change the state of the UI without any user action. But the approach seems general and flexible enough (Requirement 12 (Flexibility)) so that taking this aspect into account should be possible.

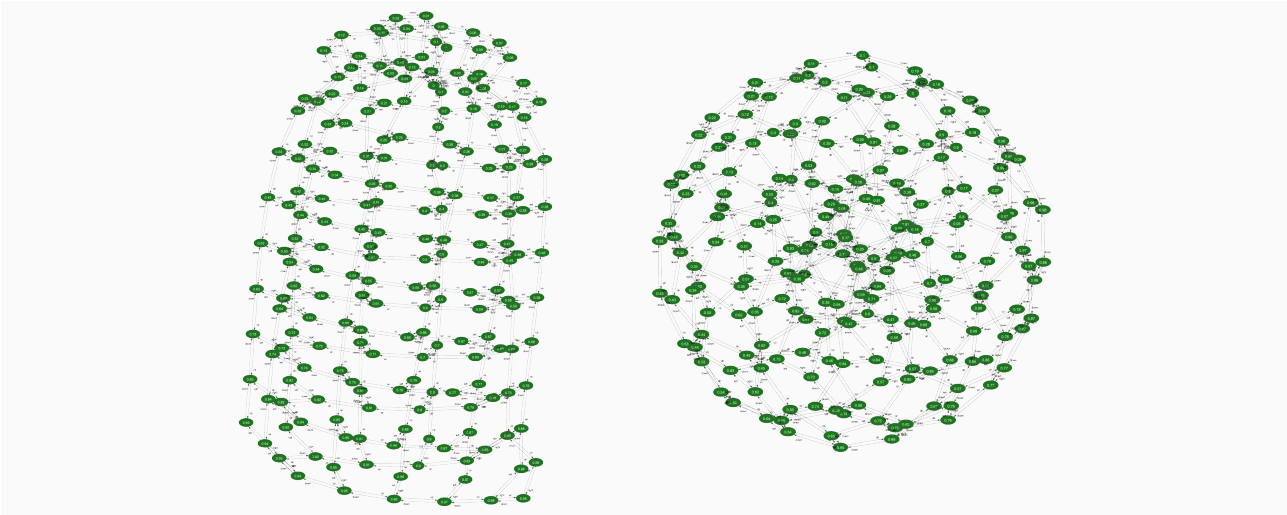


Figure 2.28: State spaces of two different number entry systems described in [114]

2.11.3 Proving the validity of UIs

This technique aims at validating UIs against task models. As explained in Figure 2.29, the idea is to develop a formal model of the UI on one side, a formal task model on the other side, and finally merge these two models in a larger formal model which should verify properties of both the task model and the UI model. Proving the validity of this model gives a formal proof that the UI allows to perform all tasks of the task model (Requirement 1 (Formal) and Requirement 11 (Verification)).

Several approaches demonstrated this technique. For example, in [39], an approach based on the B method [10] is proposed. In this approach, the task model is first modelled using CTTs, and then formalised using the approach described in [115], using several refining steps, in order to obtain a formal task model expressed in Event B[116]. The UI implementation is defined using Java Swing (See Subsection 2.4.2). Static analysis of the Java Swing based program is performed in order to generate a formal model of the UI, expressed in Event B. Then the two Event B models are merged together, using gluing invariants in order to express the links between the task model part and the UI part. This generates proof obligations. Once these proof obligations are solved, the UI is validated: the UI is proved to implement the task model.

The proofs are performed using tools that automate the process as much as possible, but in most cases, manual proofs have to be performed. The number of proof obligations can grow relatively quickly, which makes the approach reasonable only on relatively simple systems. Complex UIs can lead to extremely complex models (Requirement 3 (Simple)).

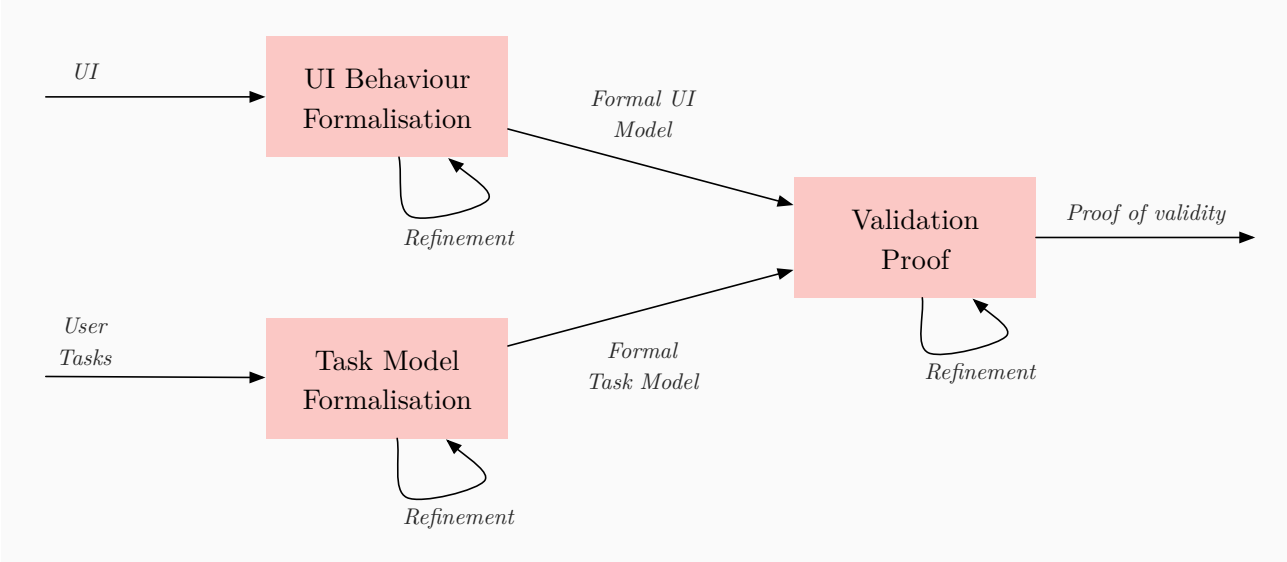


Figure 2.29: Workflow for the validation of UIs using proof methods

2.11.4 Conclusion

This section only scratched the surface of the actual application of formal methods to UI verification. Many varied ideas have been tried in this direction. Table 2.11 shows the conclusion of this section, which is itself only an overly simplified vision of the state of the art in this domain.

	1 Formal	2 Collaborative	3 Simple	4 Projection	5 Traceability	6 Composition	7 Modularity	8 Abstraction	9 Manage	10 Prototyping	11 Verification	12 Flexibility
2.11.1 Model Checking	++		-							+	++	+
2.11.2 Model discovery	+										++	+
2.11.3 Theorem proving	++		–								++	

Table 2.11: Requirements vs formal methods for UIs behaviour state of the art

2.12 Commercial tools for critical HMI

In this section, we detail two Commercial Off The Shelf (COTS) lines of products which are dedicated to the development of critical embedded HMI, which is exactly the scope of this thesis. These tools have been used successfully for the development of the UIs for several aircraft programs. A first scenario of use of these tools is to develop general DO-178B compliant HMI applications. As explained in Figure 2.30, the process involves three tools:

- A WYSIWYG UI editor to specify the graphical appearance and structure of the UI. This tool allows to model the static aspects of HMIs graphically, which makes the static definition of the UI simple to understand by all stakeholders (Requirement 3 (Simple)).
- A tool to help defining the behaviour and dynamic aspects of the UI, using models.
- A code generator that takes both static and dynamics models and generate qualifiable C code that implements the UI and eases the qualification process.

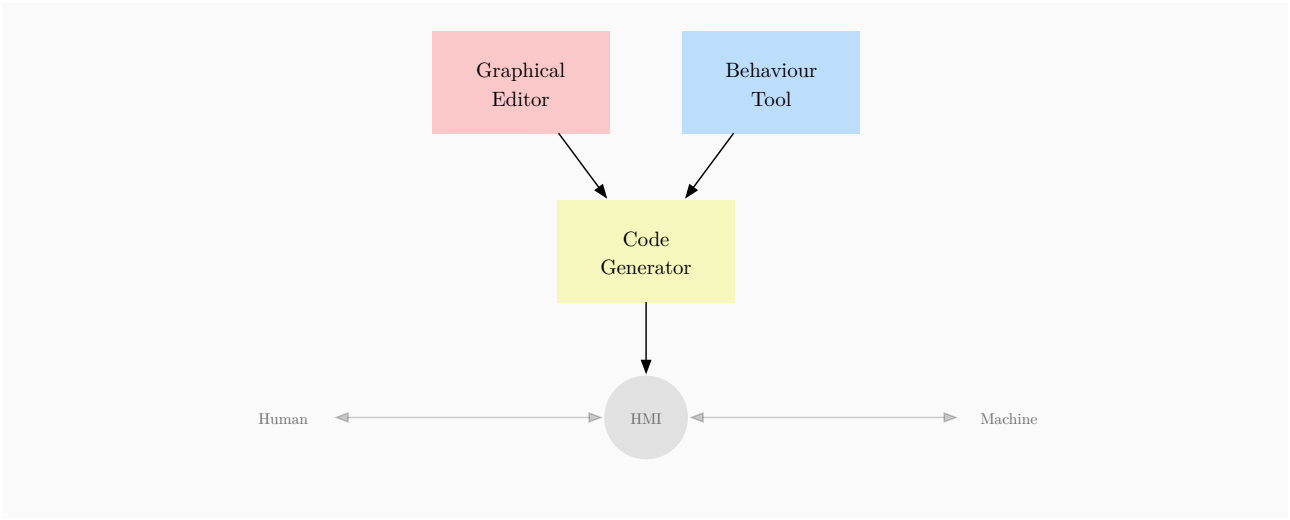


Figure 2.30: Set of COTS tools used to create DO 178B UI Systems

A second scenario of use of these tools is to describe ARINC 661 compliant interfaces. In this case, as explained in Figure 2.31, the dichotomy between Cockpit Display System (CDS) and User Application (UA) enforced by the standard (see Subsection 2.1.12) means that the tools have to be adapted to work on two sides:

- On the CDS side, the tools can be used to create new custom ARINC 661 widgets.
- On the UA side, the tools can be used to develop UAs. In this case, the static aspects of UI are not hard-coded into the program but are defined using DFs, this means that the graphical Editor should have a way to output ARINC 661 definition files. The behaviour of the UA, on the other hand, still has to be compiled in the UA.

2.12.1 Presagis solutions

Presagis proposes a set of tools which help developing DO-178B compliant UI software. Presagis allows to develop software for ARINC 661 compliant applications as well as non ARINC 661 applications. Presagis solutions include:

- VAPS XT as the graphical editor, and it also allows to specify parts of UI behaviour using state charts editors.

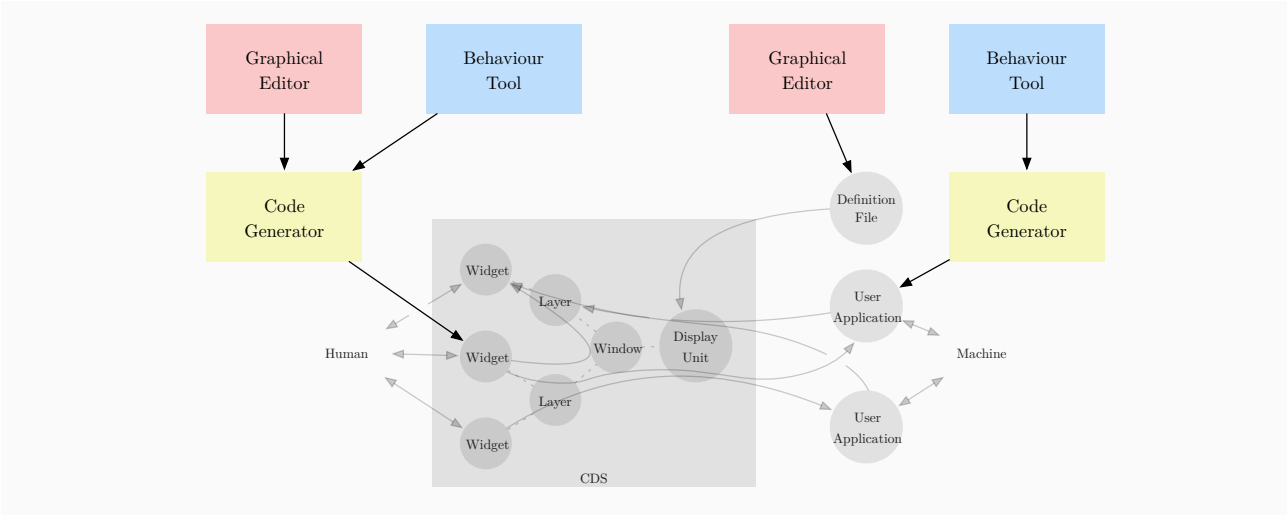


Figure 2.31: The set of Presagis tools in the context of ARINC 661 systems

- VAPS U2A is a tool that helps developing UA code in C/C++ by automating tasks such as traceability between UA code and DF.
- VAPS QCG as the code generator. It allows to generate implementation code for the static and graphical aspects, and also the associated traceability and documentation documents.

The solution provide a comprehensive toolset that allow excellent traceability (Requirement 5 (Traceability)) and a collaborative environment (Requirement 2 (Collaborative)). The WYSIWYG tool VAPS XT in particular helps making HMI design understandable by all stakeholders (Requirement 3 (Simple)) and allows prototyping of the UI graphical appearance (Requirement 10 (Prototyping)).

However, the tools do not integrate any verification capabilities (Requirement 11 (Verification)), even though the state chart tool used to specify some behaviour brings some formalism in the approach(Requirement 1 (Formal)). Another related limitation of the tool is that it does not provide abstractions which would help developing UI at a higher level of abstraction (Requirement 8 (Abstraction)) without having to deal with implementation details.

2.12.2 Esterel technologies solutions

Esterel technologies proposes an interesting toolset which plays the same role. This tool set is made of the following elements:

- Scade Display is the graphical editor. It allows to model the graphical aspect of UI by combining graphical objects. It can be used on the CDSs side to create new widgets and on the UA side to create DFs.
- Scade Suite allows to describe the dynamic behaviour graphically using blocks. This tool has proven valuable to create non-UI systems, and it can be used to describe UI behaviour, on the CDS side for new widgets and on the UA side.
- Scade Suite KCG is the code generator that transforms Scade Suite models into C code.

The main advantages of the solutions proposed by Esterel technologies as compared to its competitor is the Scade Suite tool. This tool allows to have a formal (Requirement 1 (Formal)) yet understandable (Requirement 3 (Simple)) description of the *behaviour* of UIs, thanks to the graphical representation. This tool is also already in use in other domains of aerospace software engineering. The Scade Display tool allows to have an simple to understand description of the static aspects of UI and allows prototyping (Requirement 10 (Prototyping)).

A limitation of this tool is the lack of abstraction and verification capabilities (Requirement 8 (Abstraction) and Requirement 11 (Verification)). The tool allows to verify some static properties of UI but not to analyse the behaviour of UI at higher levels.

2.12.3 Conclusion

Table 2.12 summarises the conclusion of this section. Since the two approaches are commercial products aiming at the same marker, they are relatively similar, with only slight differences between them.

	1 Formal	2 Collaborative	3 Simple	4 Projection	5 Traceability	6 Composition	7 Modularity	8 Abstraction	9 Manage	10 Prototyping	11 Verification	12 Flexibility
2.12.1 Presagis	+	++	+	++	++			-		+	-	
2.12.2 Esterel	++	++	+	++	+			-		+	-	

Table 2.12: Requirements vs state of the art commercial tools

2.13 Conclusion

This chapter described a set of very different groups of approaches that tackle problems related to HCI, critical systems and embedded software. As explained in the introduction and Figure 2.1, we tried to cover a large spectrum of approaches. While exploring these approaches, we tried to note how they could give answers to the several requirements we identified in Chapter 1.

An important conclusion is that no existing approach covers the whole set of requirements. A first reason for this is that only a handful of approaches target the particular domain of critical embedded HCIs. Obviously, approaches that do not target the specific domain do not address some of the requirements that are specific to the domain. Some approaches target this particular domain (For example, Interactors, ICOs, EOFMs, UI Verification techniques, COTS), but were not created with the same set of requirements in mind. As a consequence, even though many works provide interesting solutions to the challenges we face, no approach gave a clear solution that solved at the same time all the challenges that we identified.

Nevertheless, for each of the requirements, there are approaches that provide excellent solutions. When exploring the state of the art, we tried to understand *if* but also *why* some approaches successfully solve certain requirements in order to start getting an idea of how to answer these requirements. Table 2.13 lists the requirements of Section 1.3 and gives for each a short list of approaches that provide interesting solutions. These interesting solutions inspired the works that are developed in the following chapters the this thesis.

	1 Formal	2 Collaborative	3 Simple	4 Projection	5 Traceability	6 Composition	7 Modularity	8 Abstraction	9 Manage	10 Prototyping	11 Verification	12 Flexibility
2.1 HMI architecture					2.1.12 ARINC	2.1.8 Swing, 2.1.11 Flux	2.1.4 PAC 2.1.10 Qt	2.1.9 MVVM 2.1.11 Flux	2.1.2 Arch			
2.2 Network architecture									2.2.2 Geomorphic			
2.3 Interactors	2.3.2 York, 2.3.3 CERT						2.3.3 CERT					
2.4 Imperative languages			2.4.3 Cocoa				2.4.3 Cocoa					
2.5 Textual DSLs			2.5.1 QML			2.5.1 QML						
2.6 UI markup	2.6.3 ARINC DF	2.6.1 HTML								2.6.1 HTML		2.6.1 HTML
2.7 DOM frameworks						2.7.3 React						
2.8 FRP			2.8.2 Elm			2.8.2 Elm, 2.8.1 Reactive banana						
2.9 Graphical		2.9.4 Storyboards	2.9.4 Storyboards							2.9.4 Storyboards		
2.10 Task models	2.10.2 EOFM		2.10.1 CTT						2.10.1 CTT		2.10.2 EOFM	
2.11 UI verification	2.11.1 Model Checking, 2.11.3 Theorem proving										2.11.1 Model Checking, 2.11.3 Theorem proving, 2.11.2 Model discovery	
2.12 Commercial tools	2.12.2 Esterel	2.12.2 Esterel, 2.12.1 Presagis		2.12.2 Esterel, 2.12.1 Presagis	2.12.1 Presagis							

Table 2.13: Requirements vs state of the art

Chapter 3

The geomorphic view of interactive systems

Simplicity is prerequisite for reliability

Edsger Wybe Dijkstra

As said in Subsection 2.2.3, the geomorphic model of networking (exposed in Subsection 2.2.2 and [49]) proved to be a powerful way to expand a rigid model (the OSI model) of a system (the Internet), once the system has evolved and outgrown this rigid model.

It seems that this situation is similar to the situation encountered in the field of UI, where systems (modern UIs) have outgrown their traditional rigid model (the Arch model). The geomorphic view of interactive systems is inspired by the geomorphic view of networking, and is an attempt of application of this idea to the realm of interactive systems.

In this chapter, we present the geomorphic view of interactive systems, a way to informally represent some aspects of the global architecture of interactive systems. In a first part, we express the basic notions of this model. In a second part, we provide examples of systems described using this model.

An important assumption used in this chapter is that Humans, HCI systems and Network systems are a subset of a broader category: interactive systems. From now on, we assume that this is true, and that we can represent these three kinds of entities under the broader concept of "interactive system".

3.1 Introduction

To get an overview of the geomorphic model based on an example, consider the entities depicted in Figure 3.1: a human mind, a software application, a human eye, a computer screen, a human hand and a computer mouse. The question is: what do these entities have in common? Instinctively, one notices that these entities are related in different ways. But it is not always trivial to describe how exactly. The first thing to notice is that they are implicated in the interaction between two interactive systems: a Human and a Computer. Therefore, we call these entities *interactors*. By definition, interactors can interact with other interactors. The geomorphic model presented in this chapter is based on the idea that there are *two* orthogonal kinds of links between interactors.

The first kind of link is called a *channel*. This kind of link denotes that information can be directly exchanged between two interactors. In this example, we can find 3 channels: Computer screen - Human

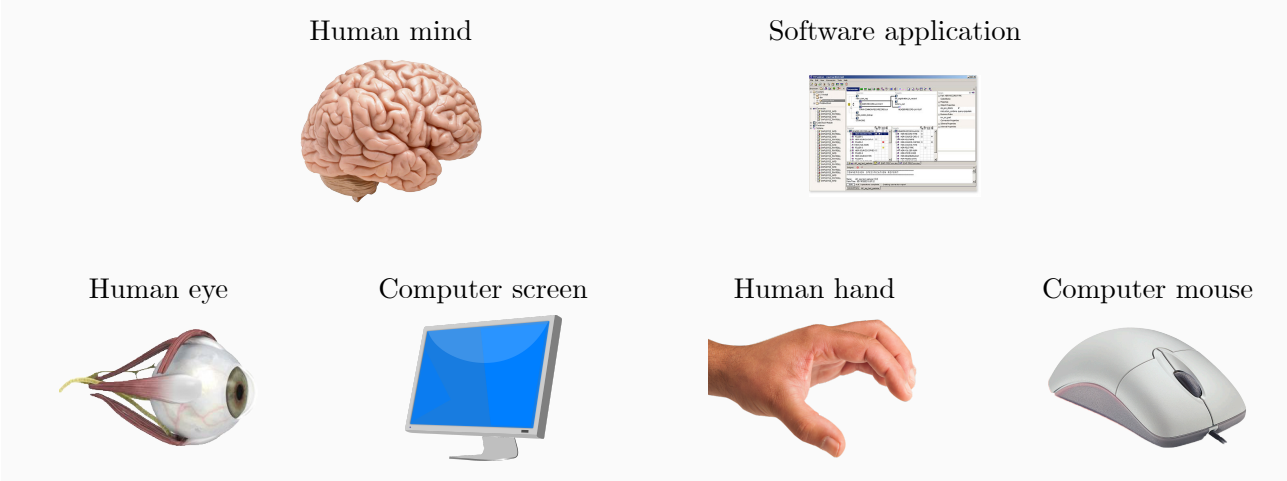


Figure 3.1: Six different entities that can interact

eye, Computer mouse - Human hand, and Software application - Human mind. These channels mean that these couples of entities make sense because information somehow flow between them. From the notion of channel, we can derive the notion of *layer*, which are sets of all entities that can be linked with specifics kinds of channels.

The second kind of link is called an *attachment*. This kind of link denotes that interactors belong to the same bigger entity, with a notion of composition. In this example, we can find 4 attachments: Human eye - Human mind, Human hand - Human mind, Computer mouse - Software application, Computer screen - Software application. The notion of *attachment* involves a notion of abstraction. For example, there is no Human hand - Human eye attachment, because interactions involving human hands are not abstractions of interactions involving human eyes. From the notion of attachment, we can derive the notion of *interactive system*, which are sets of entities that are attached together.

It is important to note that the geomorphic model has a graphical representation. This graphical representation is an important part of the model, as it attempts to help understanding the general architecture of complex systems.

Section 3.2 details the various aspects of the geomorphic model, especially the two different aspects it attempts to model. In Section 3.3, common patterns that arise when using the geomorphic view are explained. Finally, examples of use of the model on real use cases are presented in Section 3.4.

3.2 Presentation of the model

This section presents the various notions of the geomorphic view of interactive systems in detail. Note that the model is relatively simple and contains are only five important notions.

3.2.1 Interactors

An interactor is an entity that can exchange, store and process information. This complies with the general idea of interactors described in Section 2.3. An interactor is an abstract model of a logical entity. The model says nothing about the inner architecture of interactors, they are considered monolithic. This means that the geomorphic model is a global model, as defined in [28]: it does not describe the internal structure of its components.

Interactors perform interactions. The description of interactions will be treated in Chapter 4. Until then, interactors can be considered as black boxes that can interact with each other, and the description of these interactions is out of the scope of the model. Interactors must have a name that allows to uniquely identify them.

Definition 3.1 (Interactor)**.** *An interactor is a named entity that that performs interactions.*

Graphical representation Interactors are the basic building blocks of the geomorphic view of interactive systems. They represent nodes in a graph. Visually, they are represented using labelled discs as shown in Figure 3.2.

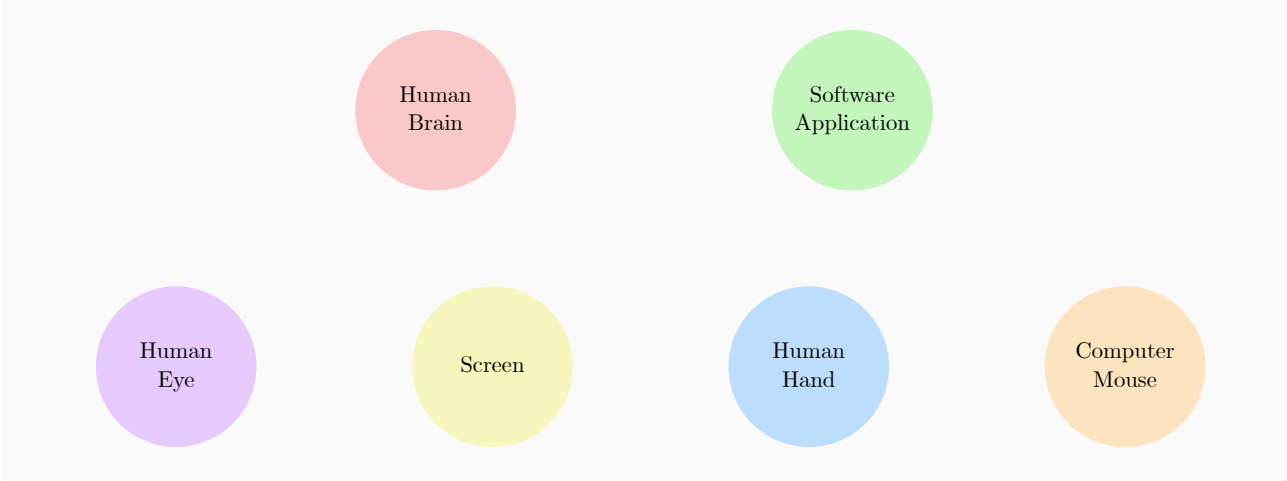


Figure 3.2: Some interactors depitcted using the graphical representation

3.2.2 Layers and channels

For any pair of interactors, there are two different possibilities: Either they can possibly interact in a way, or they cannot. It makes sense that two interactors interact if they are at the same level of abstraction, if they are represented in the same abstraction.

For example, it makes sense to connect an ethernet port to another ethernet port. Another example: it makes sense to connect a Java object to another Java object that implements a compatible interface. On the other hand it does not make sense to connect a Java object to an ethernet port. Those two entities are not at the same level of abstraction. Finally, it does not make sense to connect a Universal Serial Bus (USB) device firmware to a High-Definition Multimedia Interface (HDMI) device firmware, even though one could argue that these two entities are at the same level of abstraction, these two entities are just not in the same abstraction.

This is what the concept of layer denotes. A layer contains several interactors. Interactors in a layer are able to communicate with each other because they live in the same abstraction, the same layer. Interactors that could not possibly interact in any way are not in the same layer. The concept of layer is quite general and encompasses the concept of modality as defined in HMI engineering, as well as the concept of network layer as defined in the OSI Model [46]. It also matches the notion of layered architecture described in[117]. For example, layers can represent protocols, abstraction layer of a layered architecture, modalities of user interaction.

Definition 3.2 (Layer)**.** *A layer is the set of all interactors that can communicate in a certain way, at a certain level of abstraction*

Two interactors in the same layer can eventually be linked by a channel. A channel represents a connection, the fact that two interactors are actually interacting with each other. An interactor can have zero or more channels linking it to other interactors of the same layer. Some interactors can have a limited amount of connections, for example a most USB devices can only be connected to one USB host. But some interactors can have a very high number of connections. For example a interactor in the IP layer can be connected with several thousands other interactors of the same layer at the same time.

Definition 3.3 (Channel)**.** *A channel is an edge joining two interactors within the same layer*

Graphical representation A layer is a subgraph which contains several interactors (nodes). An interactor (node) belongs to one and only one layer (subgraph). A channel is an edge that links two interactors (nodes) which belong to the same layer (subgraph). As a consequence, a channel (edge) also belongs to one and only one layer (subgraph).

The graphical representation of Figure 3.3 shows the different layers and channels in our example. Layers are represented by large grey rectangles, and channels are represented as arrows.

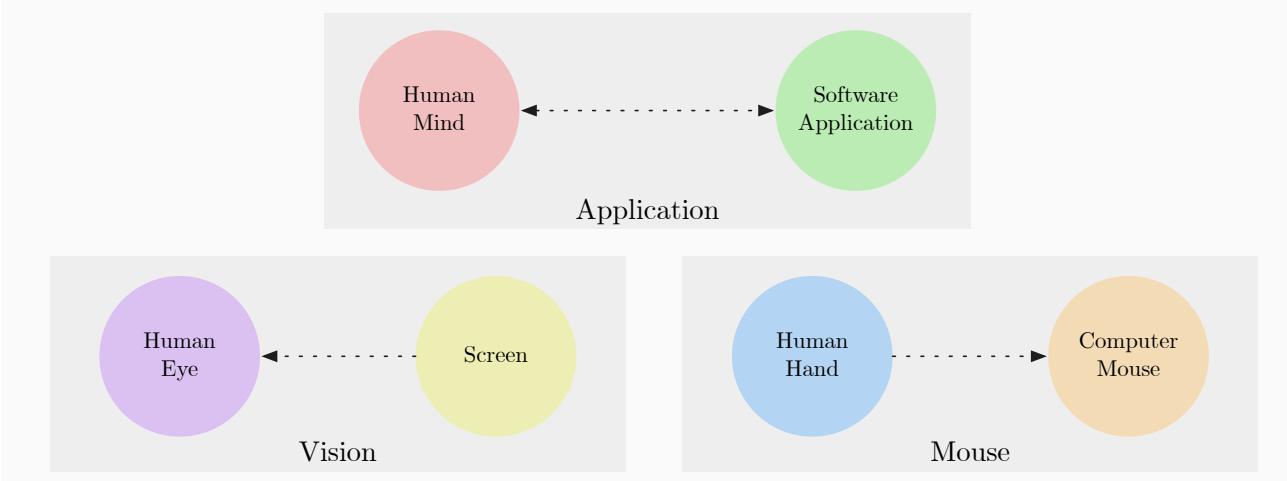


Figure 3.3: Interactors and channels within layers

3.2.3 Interactive systems and attachments

Several interactors can belong to the same system, even if they are at different abstraction levels. For example, a computer mouse, a computer screen and a software application may all belong to the same computer. On the other hand, a human eye and a computer screen do not belong to the same system.

We see that interactors all belong to logical or physical entities that spans several layers. This entity is called an interactive system. An interactive system may span several abstraction levels.

Definition 3.4 (Interactive system)**.** *An interactive system is the set of all interactors that belong to a same physical or logical entity*

An important point to notice is that communication at high levels of abstraction can only happen if it is realised by communication at lower levels of abstraction. For example, consider the Human Mind - Software Application channel. At a certain abstraction level, we can consider that this channel exists, and these two interactors can interact through this channel. But if we dig deeper, we see that this interaction actually needs to go through lower levels of abstractions in order to actually happen. The

Human mind cannot magically interact with the software application, the interaction has to go down to the physical layer so that the physical hand of the user acts on the physical mouse of the computer, and from then, the interaction goes up in the abstraction layers to the software application.

This means that interactors which belongs to the same interactive system interact with each other, even if they are at different abstraction levels. More precisely, some interactors are less abstract than others, and are used to implement the interactions of higher-level interactors. This relationship is what we call an attachment.

Definition 3.5 (Attachment)**.** *An attachment is a directed edge joining two interactors within the same interactive system*

Graphical representation An interactive system is a subgraph that contains several interactors (nodes). An interactor (node) belongs to one and only one interactive system (subgraph). An attachment is an edge that links two interactors (nodes) which belong to the same interactive system (subgraph). As a consequence, an attachment (edge) also belongs to one and only one interactive system (subgraph).

The graphical representation of Figure 3.4 shows the different interactive systems in our example. Interactive systems are represented by coloured rounded shapes, and attachments are represented as coloured lines with a circle on the higher layers.

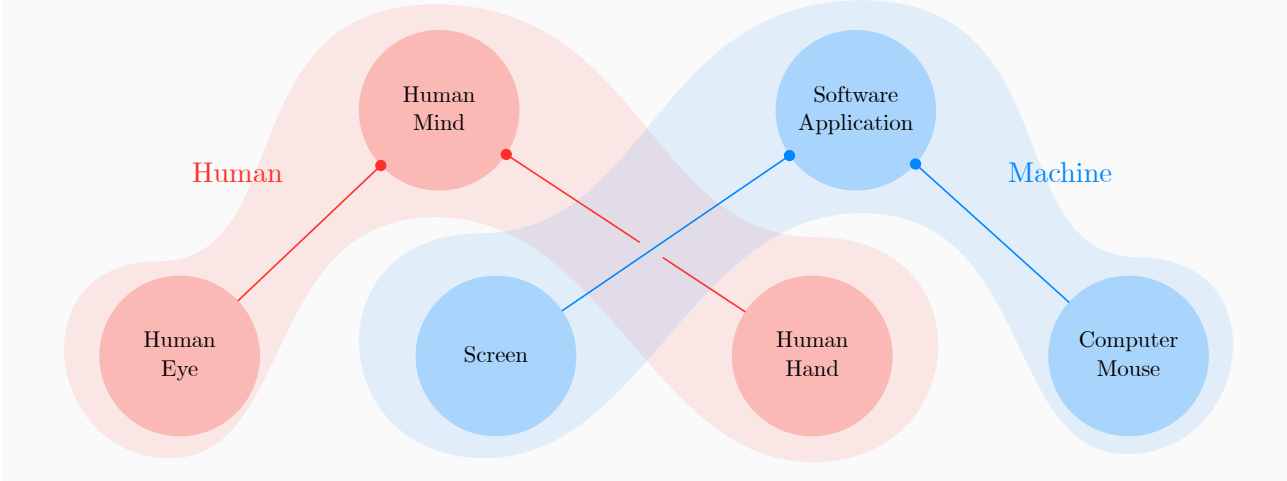


Figure 3.4: Two interactive systems

3.2.4 Overview

The geomorphic model presents one type of entity, the interactors, which are linked with two kinds of relationships: channels and attachments. These two relationships define the notions of Interactive systems and Layers. Figure 3.5 presents a metamodel of the geomorphic view of interactive system.

These two relationships are associated with two graphical representations. These two representations can be combined on the same graphical view in order to give an overall view of the models. The layout of this view can be used to emphasise one dimension of the other. If we emphasise the Channel-Layer dimension, we end up with a layered view, where each layer is an horizontal line and systems are tree-like shapes spanning several layers, as seen on Figure 3.6. This view is called the *vertical view*, because it shows a cut of the vertical stacking of layers. If we emphasise the Attachment-System dimension, we end up with a clustered view, where each system is a cluster of interactors regrouped

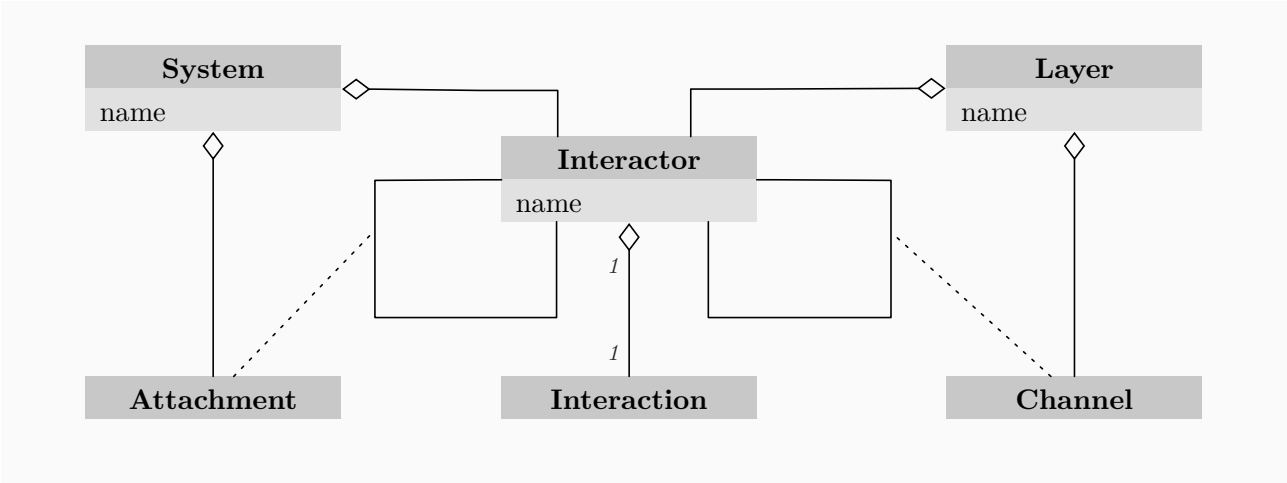


Figure 3.5: Metamodel of the geomorphic view of interactive systems as a UML class diagram

together, and layers are long curved shapes linking systems, as seen in Figure 3.7. This view is called the *horizontal view*.

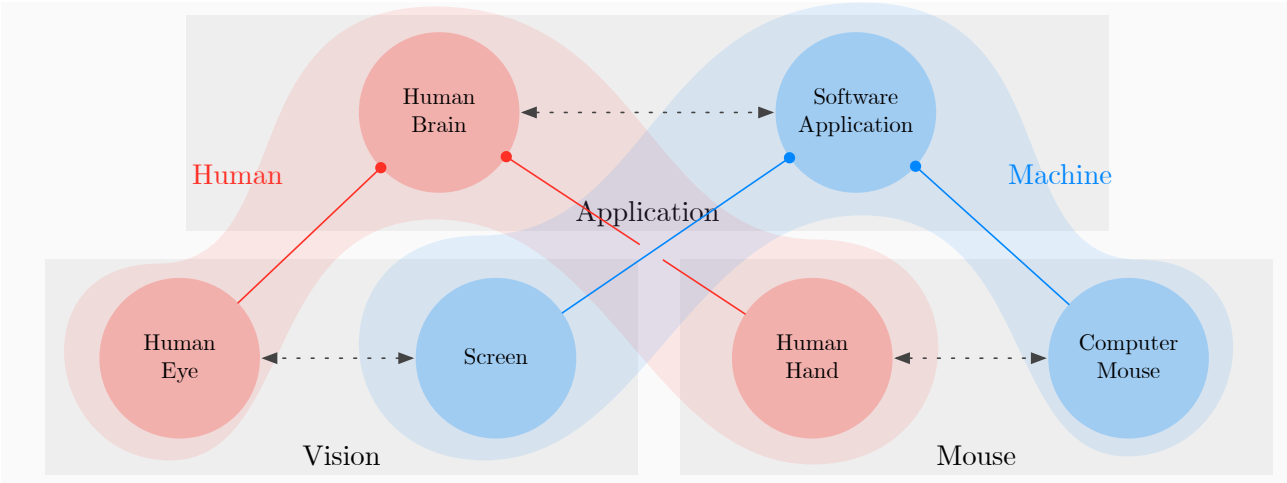


Figure 3.6: A view of a simple model with emphasis on layers

In Figures 3.8 and 3.9, we see that this views can be applied to larger models and allow to have a clear view on them even if they are detailed precisely.

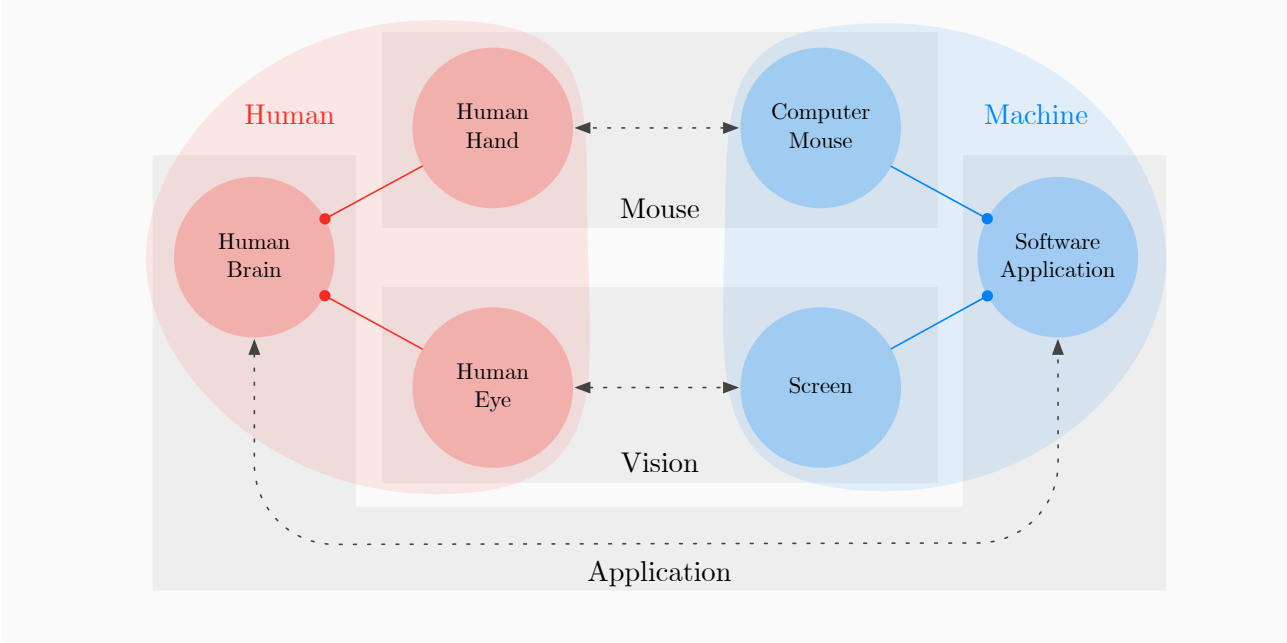


Figure 3.7: A view of a simple model with emphasis on interactive systems

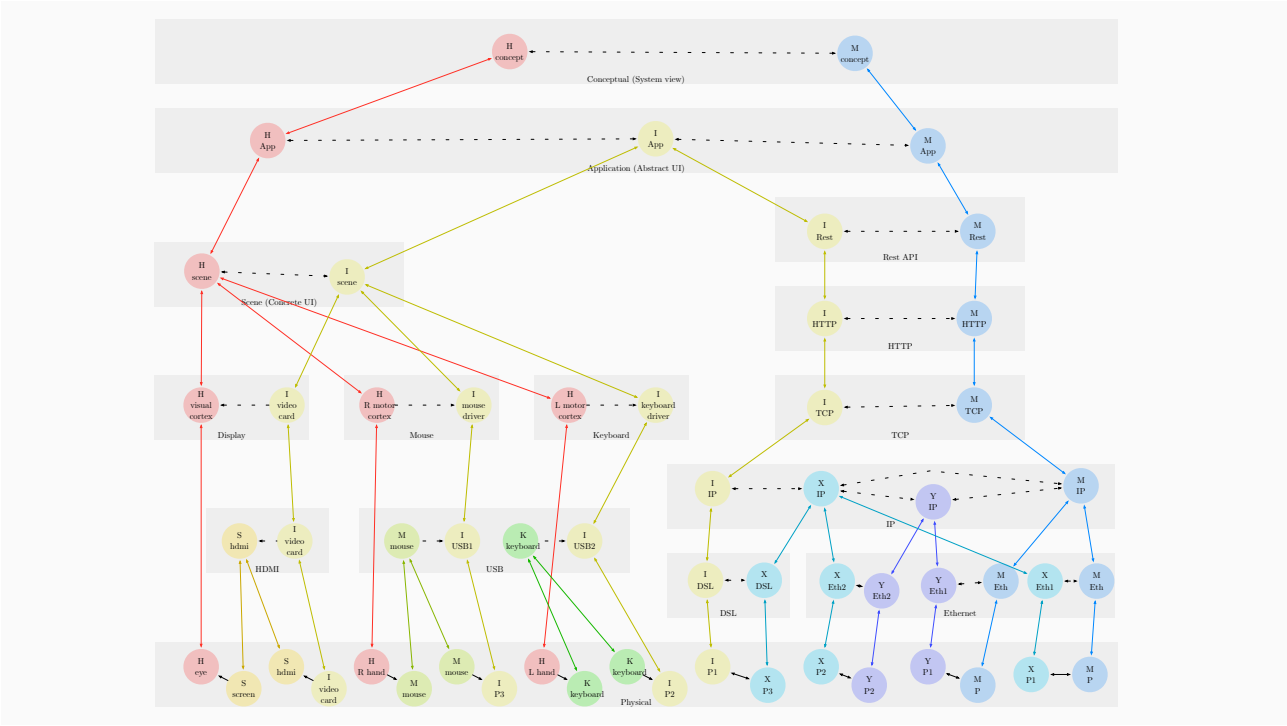


Figure 3.8: A view of a complex model with emphasis on layers (Vertical view)

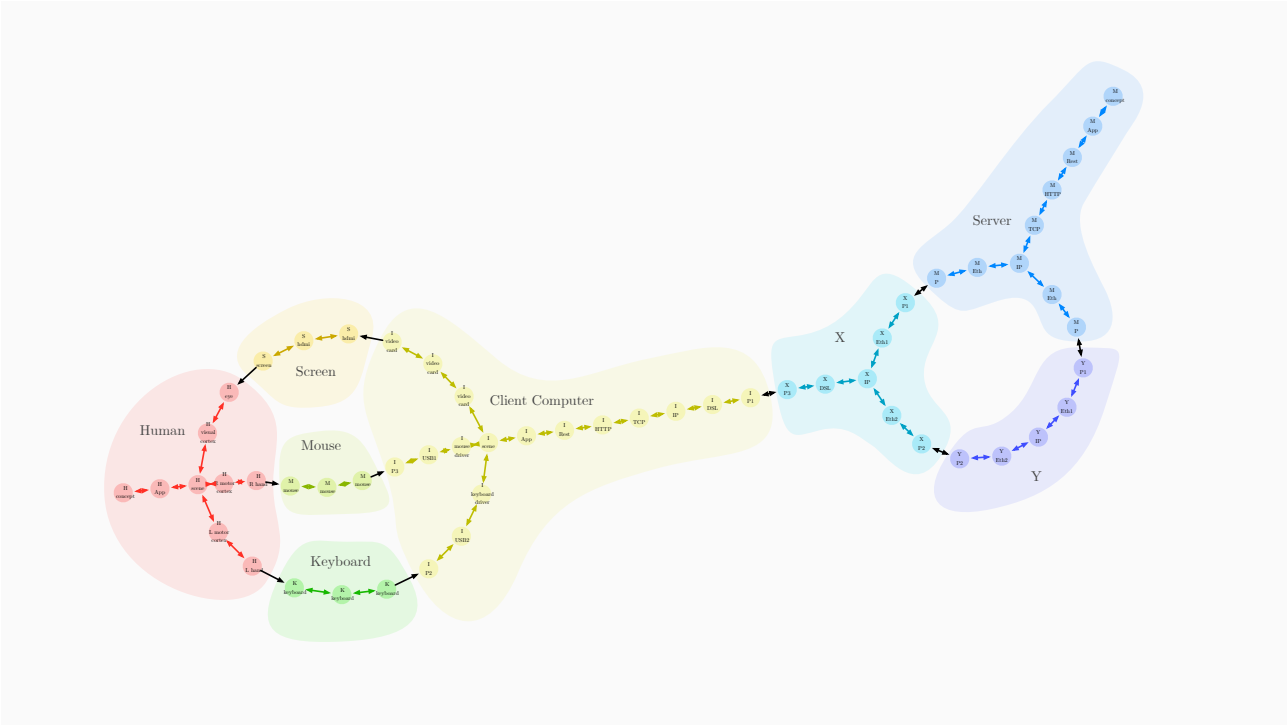


Figure 3.9: A view of a complex model with emphasis on interactive systems (Horizontal view)

3.3 Common patterns

In this section, we will describe common patterns that happen in the field of HCI and in the field of network systems. These patterns have been identified while modelling various systems using the geomorphic view of interactive systems. They represent situations that commonly happens in different parts of models. Most complex geomorphic models can be described as a combination of such patterns.

3.3.1 Rendering

Rendering is a common pattern used in HCI, especially for GUIs, but also for voice-based interaction. It is also used in lower levels of network systems. This patterns describes a unidirectional channel between two interactors of a high level layer (Called “Abstract” in this example). In this layer, the Emitter interactor sends information to the Receiver interactor. Communication in this abstract layer is delegated to a lower level layer (Called “Rendered” in this example). In the lower layer, an interactor called Renderer receives the messages send by the higher level Emitter, transforms it in a given way, and sends on a channel to another low-level interactor called the Interpreter. The Interpreter receives the information and interprets it for the receiver. Here are some examples of use of the rendering pattern:

- 3D scene rendering: The emitter is a 3D application, and it sends a 3D scene to the receiver, a human user. The renderer is the graphics library (Open GL) that generates an image (array of pixels) from the 3D scene. The Interpreter is the human visual cortex that interprets the image and creates a mental model of the 3D scene.
- HTTP: The emitter is a server, and it sends data to a client application. The rendered information is an HTTP response.
- Radio Modulator Demodulator (MODEM): The abstract information is a sequence of bits. The rendered information is a radio waveform. The renderer is a modulator, and the interpreter is a

demodulator.
- Voice input: The emitter is a human user, sending words. The receiver is a computer application receiving those words (maybe to interpret them in an even higher level). The renderer is the laryngeal motor cortex of the human[118], the interpreter is a voice recognition engine.

A limitation of the geomorphic view is that it says nothing about the actual actions performed by these various interactors. For example, the renderer and interpreters can perform very different actions depending on the actual instance of this pattern. For example, the renderer can send information using time division multiplexing, space division multiplexing, or apply coding or any arbitrary transformation on the data before sending it. This lack of detail in the model is what makes it possible to identify common patterns in various situations.

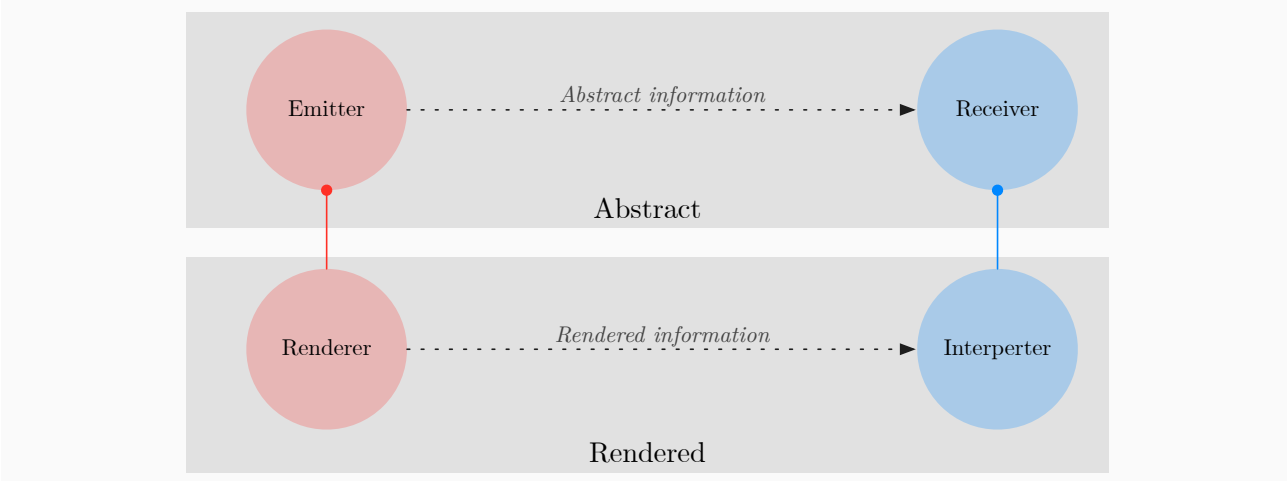


Figure 3.10: Rendering Pattern

3.3.2 Multimodality

Multimodal UIs are UIs with which the user can interact using several various input and output devices. These various modalities can be used together in combination in order to perform complex interactions that make sense.

Figure 3.11 describe the pattern of multimodality. In this model, a channel in an overlay (the Abstract layer) is delegated to several channels in different underlays (Modality1 and Modality2). Some of these lower level layers are unidirectional (input devices or output devices), but communication in the higher level layer is bidirectional: several modalities are used as input, and some others are used as output. Only the combination of these various modalities enable the higher level interaction. Here are some examples of the multimodality pattern:

- WIMP interface: The Abstract layer represents abstract interaction between the user and the UI, and there are 3 different modalities layers: Mouse, Keyboard and Screen. These three modalities are unidirectional, but their combination allows a bidirectional interaction on the higher level layers.
- Multimodal interfaces: The modality layers can be gesture recognition, multitouch, screen, mouse, voice recognition or various modalities that are combined at a higher level of abstraction.
- Multi-protocol application: Applications that communicates using different protocols. Each protocol is represented as a modality.

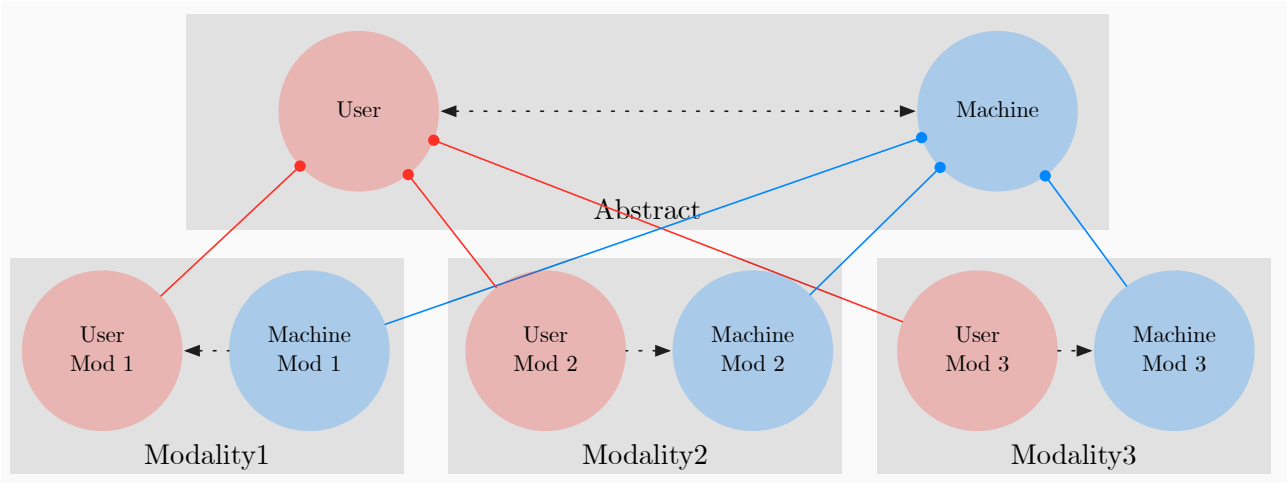


Figure 3.11: Multimodality pattern

3.3.3 Mobility

Mobility is a pattern that comes from network systems, but has applications in the UI world. It is inspired on a simplified and deformed version of mobility as described in [49].

Figure 3.12 presents this pattern. An interactor (Base) in an overlay (Abstract layer) is represented by several interactors (Base Agent 1 and 2) in the underlay (Mobility layer). An interactor (Mobile) interacts with Base in the Abstract layer. This interaction is implemented in the Mobility layer. Depending on different conditions, the Mobile agent in the Mobility layer can interact with either of the Base agents. Examples of this pattern:

- Mobile networks: The base is a mobile broadband provider internet gateway. The Mobile is a mobile phone. Connection from the mobile phone to the internet is represented in the Abstract layer. Base Agents in the mobility layer represent antennas of different cells. Depending on its physical position, the mobile phone will connect to either of these antennas.
- Redundant Cockpit Display Systems (CDSs): The base is an aircraft application, the mobile is a Pilot. The Mobility Layer is the abstract UI layer. The Pilot can interact with several different UIs at the same time. If one of them has a problem, the pilot can still interact with the aircraft using the remaining base Agents, which are CDSs
- Redundant hands: The Base is a Human user, the Mobile is an Application on a Touchscreen device. The Base Agents are the two hands of the human user. The mobile agent is the touch screen. If the user cannot reach the touchscreen with one of his hands, she can still use the device using the other hand.

Again, since the geomorphic model does not detail interactions, we note that the same pattern matches very different situations. These different situations share the same general structure, but nothing more. For example, algorithms that implement mobile phone mobility have nothing to do with those that manage multiple CDSs.

3.3.4 Client-server

The client server pattern is the situation that arise when an entity (the server) interacts with several other entities (the client) at the same time. Clients on the other hand communicate with only one server. Figure 3.13 depicts such a situation. Occurrence of this patterns in models gives an important

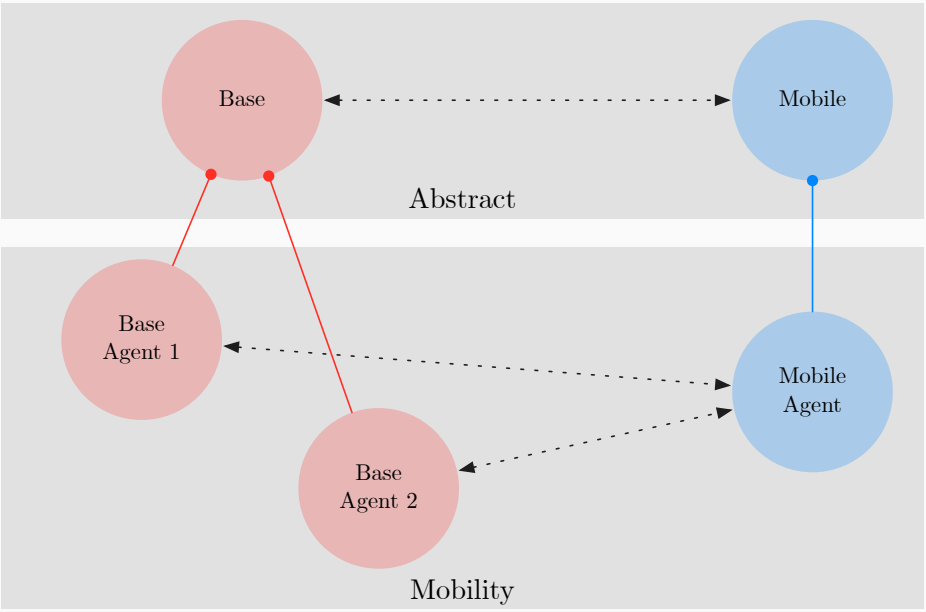


Figure 3.12: Mobility pattern

information: The interactor which takes the role of the server has to be relatively complex, because it has to manage the several connections it has. Examples of this pattern include:

- Web server: The layer represents HTTP. The server is a web http server, and clients are client browsers. A web server can deal with thousands of simultaneous connections, the fact that HTTP is a stateless protocol means that the server does not have to store a state for each client, which greatly eases the work of the server.
- Database connection: The layer represents a high-level connection to a database, for example a Structured Query Language (SQL) database. Clients are web servers agents that need to access the database. In order to maintain good performance while serving numerous clients, the database engine can use techniques such as connection pooling.
- Multi User UI: A central UI is used by several user at the same time. The UI has to manage the different users in order to offer them a consistent experience. For example a multi touch table that can be touched by several user has to remember which user is where, and be able to know which touch comes from which user.

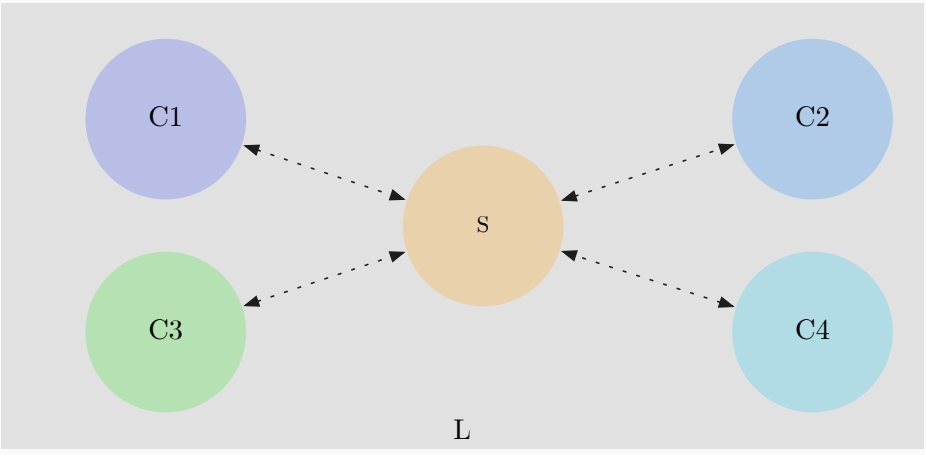


Figure 3.13: Client-Server Pattern

3.3.5 Interface

The interface pattern is very common in both UIs and network systems. Figure 3.14 describes this pattern: Two interactors Xa and Ya interact directly in an overlay O. Their interaction is implemented by interactors Xu and Yu in an underlay U. In the underlay U, a third party interactor I acts as an interface between Xu and Yu. For obvious reasons, this interactor is often called "interface". Examples of the Interface pattern:

- HMIs: X is the human user, Y is the machine. O is an abstract layer where the human interacts directly with the machine. For example, in this layer, the User sends a target speed to his car. In a lower level layer U, we see that three entities exist: the User, the Machine, and the HMI. Interactions between the user and the machine always happen through the HMI. If the HMI ceases operating normally, then the link between the user and the machine in the overlay cannot happen anymore. In this layer, the user interacts with the HMI, and the HMI interacts with the car. These two interactions can be implemented in two different even lower layers.
- Chat server: Two users X and Y of a chat application think they are interact with each other directly (layer O). But if we dig deeper, we see that their interaction goes through a server in a lower layer U. At a higher level, the Two users do not have to be aware of that.

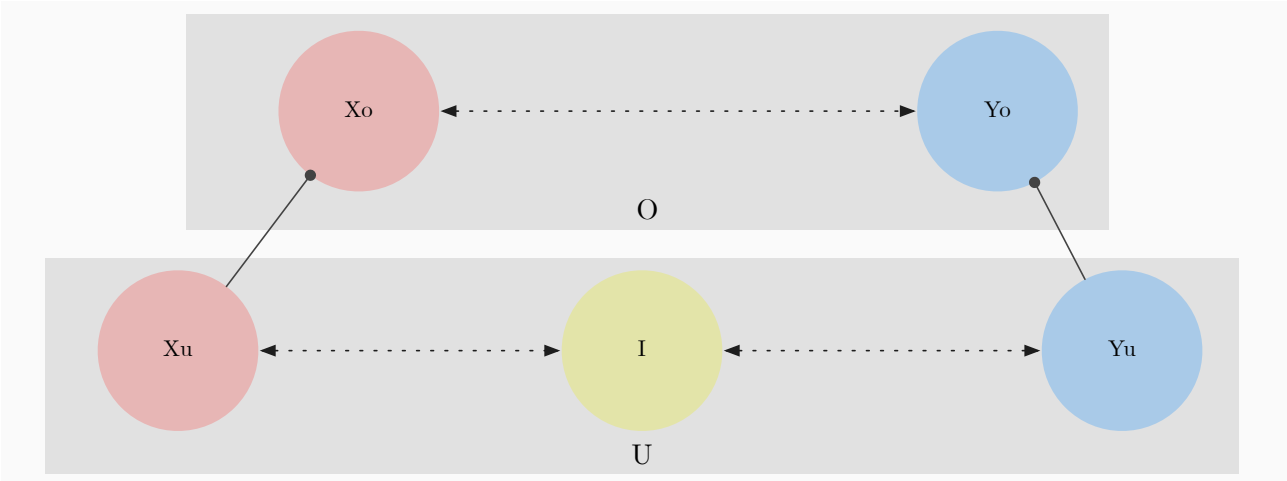


Figure 3.14: Interface Pattern

3.3.6 Routing

The routing pattern is common in the field of networking. In this pattern, interactors in a layer have several possible routes to communicate, with other intermediate interactors. Along with actual messages, interactors must identify which interactors their messages are destined to. Intermediate interactors (routers) must manage the routing of signals, by remembering appropriate routes, and be able to adapt to activation of deactivation of certain channels. Examples of this pattern:

- Network routers can be represented as interactors that belong to the Network layer (IP). They forward messages that are sent by other agents. The destination of messages is specified in the IP header of messages, and Routers manage the routing of messages by maintaining a routing table.
- Network switches follow the same pattern, in a different layer (The link layer).

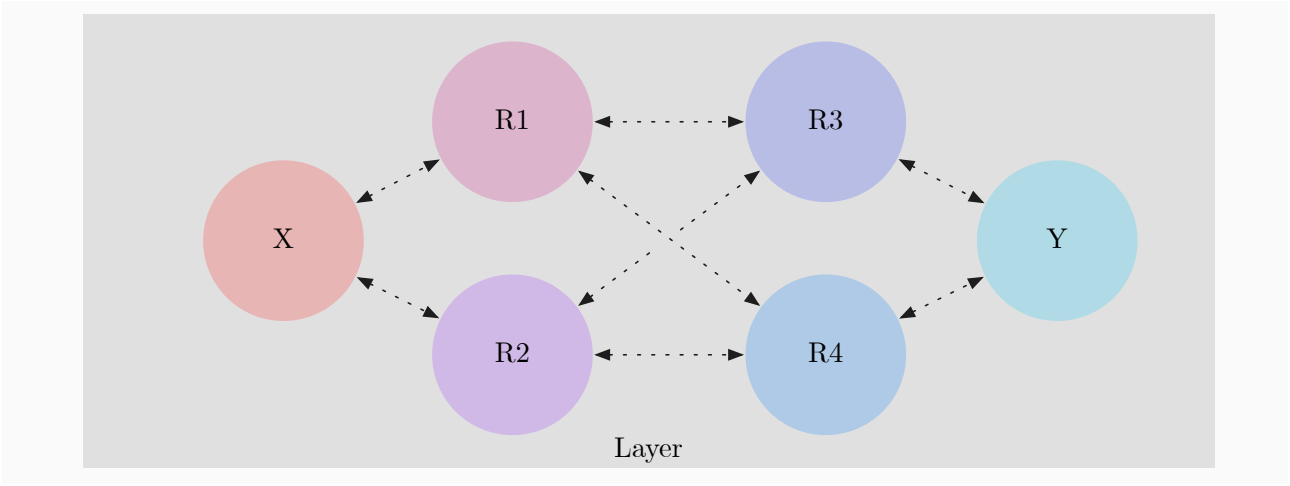


Figure 3.15: Routing Pattern

3.4 Example systems

The geomorphic view of interactive systems can be put to use in order to model real-life systems. In this section, we will present some relevant examples.

3.4.1 WIMP Interfaces

3.4.1.1 Description

Layers WIMP applications can be represented using the geomorphic view of interactive systems. The architecture of such systems is expressed using several layers. Figure 3.16 and Figure 3.17 shows two views of this same architecture model. For a first approach on the model, we will focus on Figure 3.16, which emphasises layers. The layers depicted in Figure 3.16 are:

- The conceptual layer: represents the interaction between the user and the machine at the highest level of abstraction. The information exchanged in this layer denotes things like "The user turns on the weather radar". At this level of abstraction, the UI does not exist. This layer represents what the UI has to implement.
- The Abstract UI layer: represents the abstract interaction between the UI and the user on one side and the UI and the machine on the other. The information exchanged in this layer denotes things like "The user opens the $WeatherRadar$ control component" or "the user triggered the $ToggleRadarOn$ action". This represents the dialogue Control part of the Arch architecture model, or the Model of the MVC architecture.
- The concrete UI layer: represents the interaction between the User and the UI at a lower level of abstraction. Information exchanged in this layer denotes things like "The user clicked on the button called $RadarPowerButton$" or "The $RadarPowerButton$ is in the *pushed* position".
- The Toolkit: represents the Toolkit part of the Arch model. The information exchanged in this layer denotes things like "The user moved the mouse to a given position", "The user moved tapped a key on the keyboard" or "The UI displays a list of polygons and letters to the User".
- The Modalities layers: Display, Mouse and Keyboard, represent concrete interactions which happen between the user and the machine using the three modalities of WIMP UIs.
- The Application Application Programming Interface (API) layer: represents the level of abstraction provided by the API of the functional core of the application. Information exchanged in this

layer denotes things like "Set property *enabled* of *WeatherRadar* to *true*". This layer is relatively abstract, but purely functional and independent from the UI.

- The Functional layer: represents the the level of abstraction of the actual functional core of the interactive system. Information exchanged in this layer denotes things like "Send message *WeatherRadarOn* to the Weather Radar system". There are other layers under this layer, but the WIMP paradigm is not specific about those layers that concretely implement the communication between the UI and the machine. so we leave them away from this model. Depending on implementations, these layers could represent interprocess communication on the same machine (Typical desktop application), communication with a server over the internet (Web application), AFDX communication (Aircraft system UI), or many other possibilities.

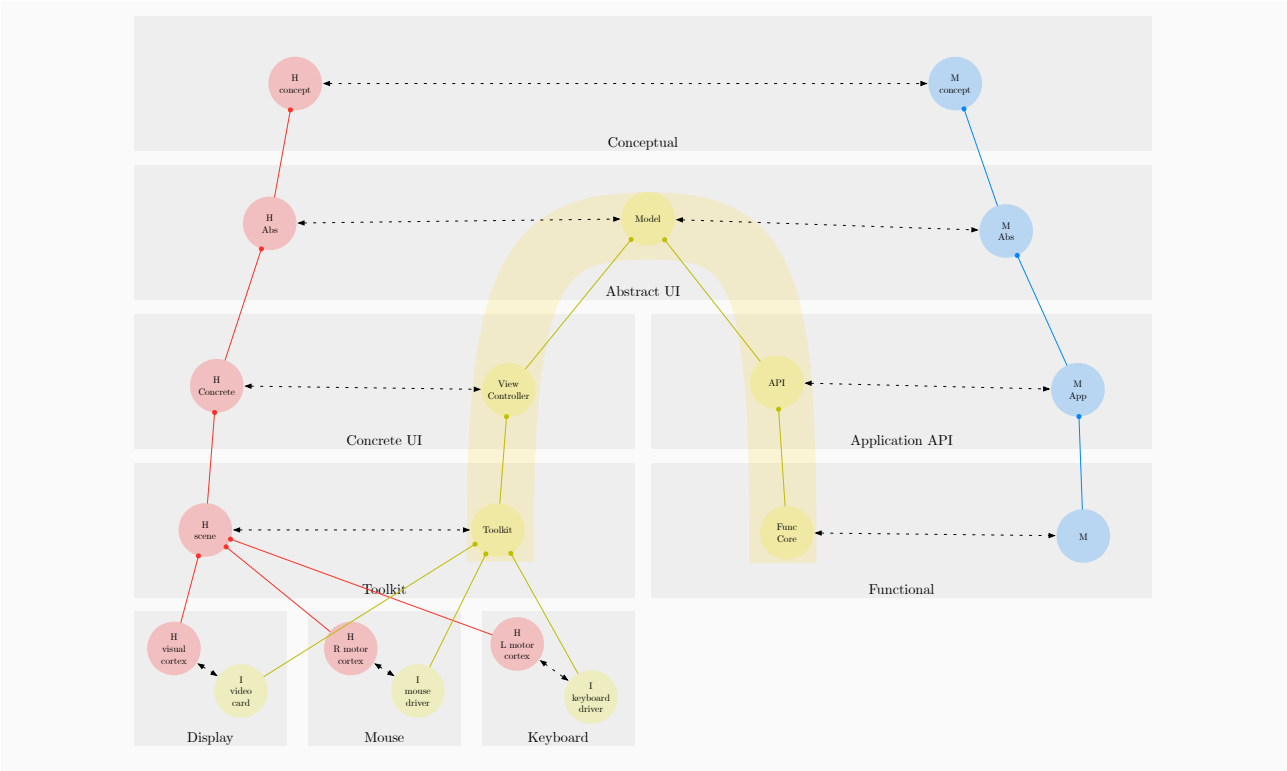


Figure 3.16: A vertical geomorphic view of a typical WIMP Architecture (Arch Model in Yellow)

Interactive systems In a second approach, we can have a look at Figure 3.17, which represents the same system as Figure 3.16, but this time with an emphasis on interactive systems rather than layers. The three interactive systems implicated in WIMP interaction are:

- The Human, depicted in red. The human interacts at various level of abstractions, from the conceptual level down to the modalities level. The actual interaction only happens at the lowest levels, using the three modalities. One of these modalities in an output modality: the Display. All the data going from the machine to the human has to go through the display as no other output modality is described in the WIMP paradigm. Communication from the user to the machine can use two channels however: the keyboard and the mouse.
- The Machine, depicted in blue. The Machine interacts at various levels of abstraction. At the higher level of abstraction, the machine interacts directly with the human, the UI is abstracted away. But actually, interactions between the machine and the human are performed via the UI.
- The Interface, depicted in yellow. It represents the only way for information exchange between the Human and the Machine. The UI interacts with the user using the three modalities. The UI interacts with the machine using various methods which are out of the context of the WIMP

paradigm, and are not expressed in detail here.

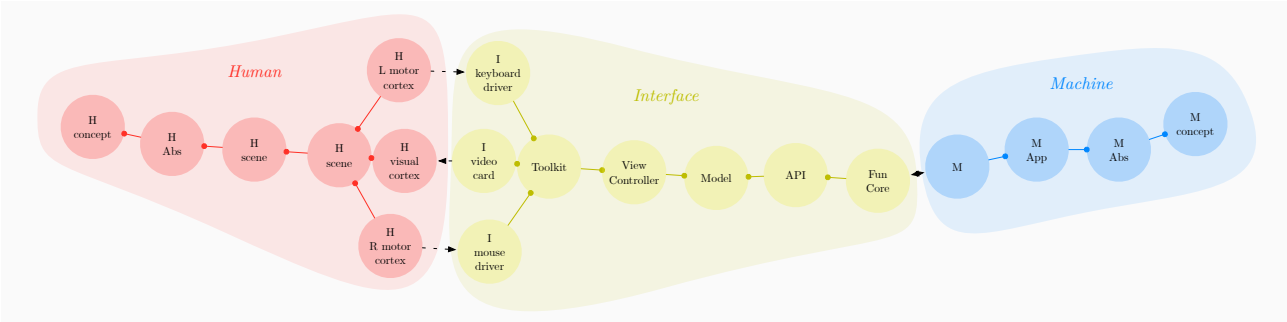


Figure 3.17: An horizontal geomorphic view of a typical WIMP Architecture

3.4.1.2 Analysis

Embedding of the Arch model A first thing to note is that the Arch Model (Section 2.1.2) is embedded in the Geomorphic model of WIMP applications. The Arch model is represented as a yellow arch in Figure 3.16. We see that the Arch model correctly cover the architecture of the whole UI system, except the lower layers related to modalities (display, mouse and keyboard). The five parts of the Arch model are present in the model, and their roles in the geomorphic model match their roles in the Arch model:

- Interaction Toolkit Component: Represented as the UI interactor in the Toolkit layer.
- The Logical Presentation Component: Represented as the UI interactor in the Concrete UI layer.
- The Dialogue Component: The Abstract UI interactor in the Abstract UI layer. This component is at the top of the Arch model because it is the most abstract of the five components. It is also at the top in the geomorphic model for the same reason.
- The Functional Core adapter: The UI interactor in the application API layer.
- The Functional Core: The UI interactor in the Functional layer

Safety considerations The extended scope of the geomorphic view allows to check that some dependability properties might be met or not. For example from Figure 3.17 we can infer that the screen is a weak point: all paths from the machine to the human have to go through this system. As a consequence, a failure of the screen means that the system will not be able to operate anymore. On the other hand, we see that the mouse and keyboard *might* provide redundancy. If the mouse disappears from the model, there is still a path from the user to the machine, going through the keyboard.

As a matter of fact, no WIMP system can be operated if the display device fails. But some WIMP systems provide full mouse-keyboard redundancy, and can be fully operated using only a keyboard (using tab-focus and enter as an alternative for mouse clicks), or only a mouse (using a virtual keyboard as an alternative for keyboard strokes). The geomorphic models allows to formalise this very simple property. Note that it provides a *necessary* but not *sufficient* condition for the dependability of the system. Dependability of the whole system can only be asserted by modelling the interactions performed by the interactors, by going in the black boxes, and analysing what happens inside.

Feedback loops The geomorphic model allows to visualise clearly the various feedback loops involved in UI systems. As an example, let us consider a classical WIMP case of feedback loop: a “Add to cart” button. Such a button presents various feedbacks when clicked, as shown in Figure 3.18. The four feedbacks presented when this button is clicked are:

- The *physical feedback* consists in a physical "click" sound and the associated haptic feedback, so that the user knows immediately that the mouse was properly clicked.
- The *Toolkit layer feedback* consists is displaying the button in the "pushed" state, so that the user knows that the click triggered the UI button.
- The *Abstract layer short feedback* consists in showing a progress indicator, to show that the system actually took into account the click, and is performing the related action.
- The *Abstract layer functional feedback* consists in showing the final functional result of the action, confirming to the user that the item was correctly added to the cart.

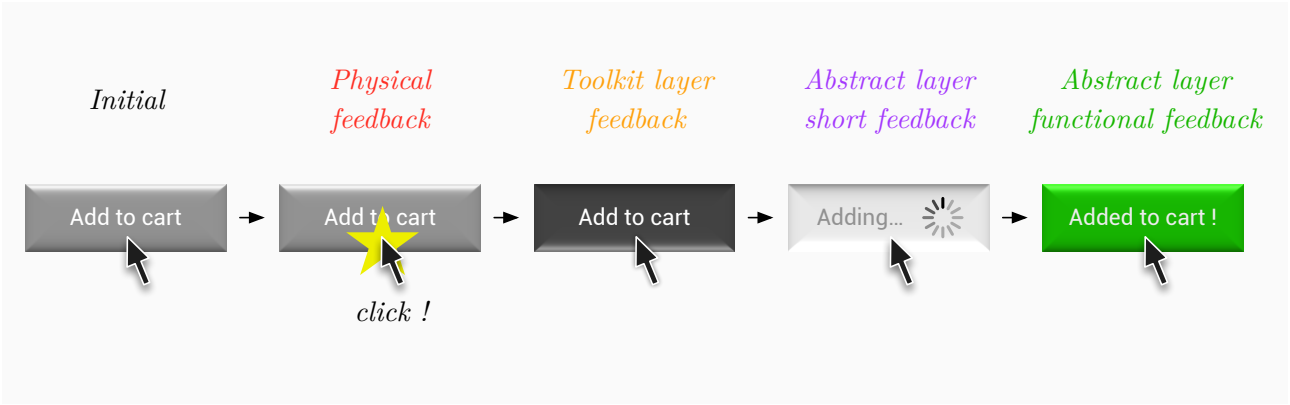


Figure 3.18: Three different feedbacks for a "Add to cart" Button

These three feedback loops are performed at various abstraction layers, by various interactors. The geomorphic view of interactive systems allows us to show clearly these three feedback loops. Figure 3.19 shows these feedback loops with the same layout as Figure 3.16, and Figure 3.20 shows them with the same layout as Figure 3.17 for reference.

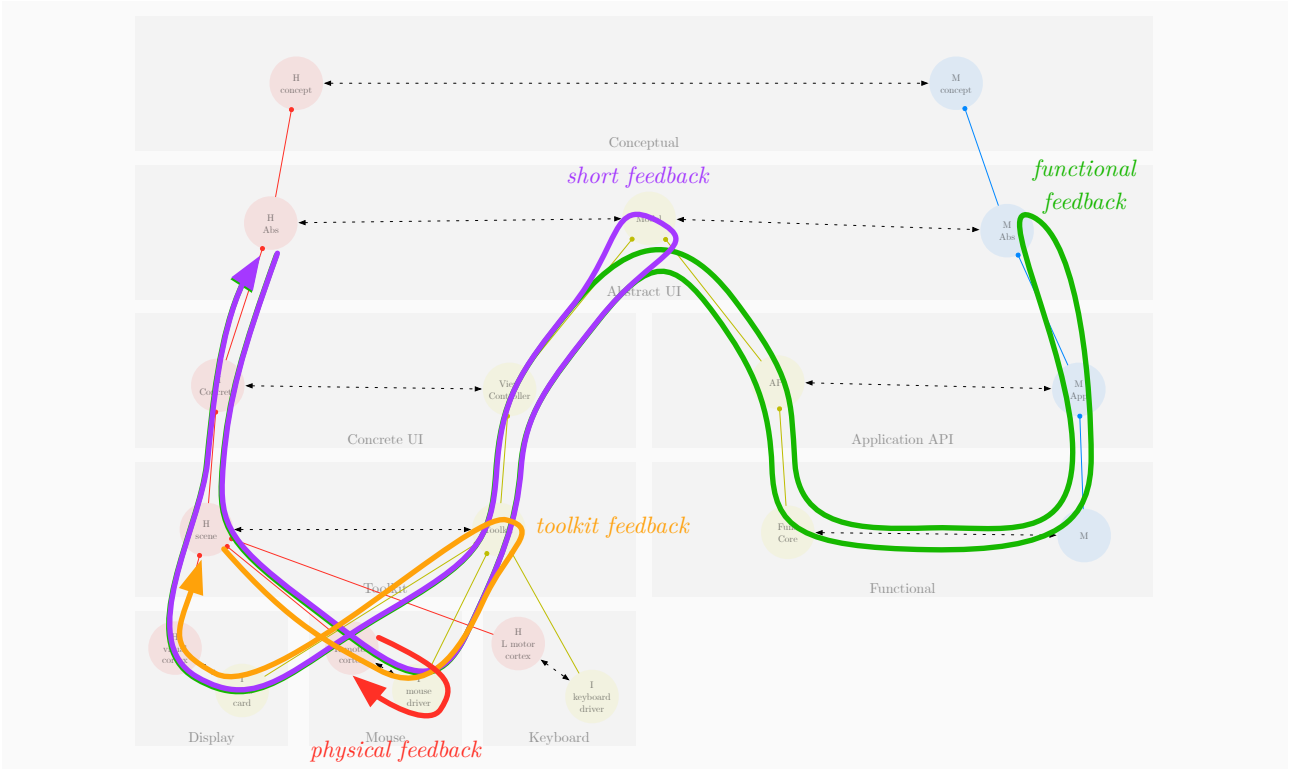


Figure 3.19: Feedback loops in a WIMP architecture shown on an vertical view

A potentially interesting use of the geomorphic model of WIMP interfaces would be the derivation

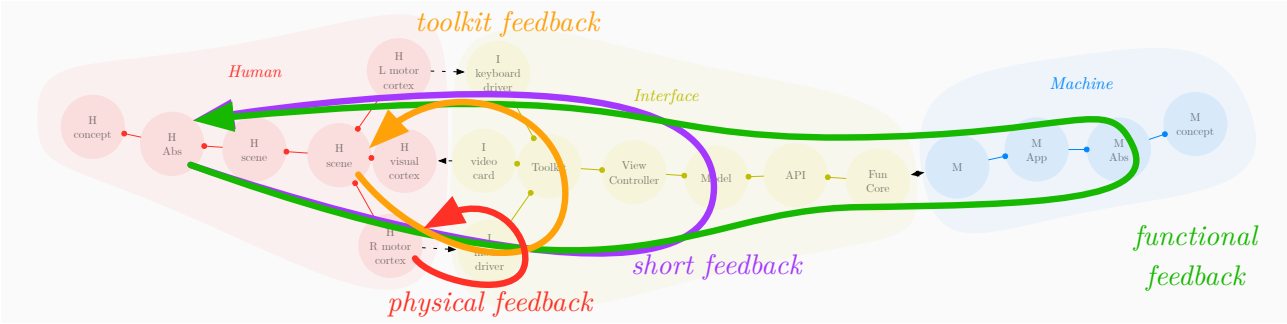


Figure 3.20: Feedback loops in a WIMP architecture shown on an horizontal view

of timing properties for these various feedback loops. If lower and upper bounds of the execution times of each nodes are given, we can infer bounds for end-to-end delays of the UI, which is a crucial indicator of usability[119]. In particular, these delay analysis could also include human tasks, such as cursor pointing or text typing, which are rarely included in typical UI usability analysis [120] which only take into account delays caused by the machine itself.

3.4.2 Multimodal interfaces

3.4.2.1 Description

The geomorphic model allows to describe the global architecture of multimodal interfaces. The basic architecture of such interfaces is relatively similar to that of WIMP interfaces, except for the addition or other modalities and toolkits. Figure 3.21 and Figure 3.22 show the geomorphic view of a multimodal interface architecture with three additional modalities: Touch, Gesture and Voice recognition. An important thing to notice is that the additional modalities can be integrated at various levels of abstraction:

- The touch modality is integrated at the Toolkit level. This means that the touch modality will be deeply integrated within the system. It will have the same status as the mouse modality. In particular, the user will be able to click on buttons using the touchscreen, as if it was the mouse. In order for this architecture to be possible, the Toolkit has to have support for touchscreens, which is the case in many modern UI toolkits such as Qt (Subsection 2.4.1),Cocoa (Subsection 2.4.3), .NET...
- The gesture recognition modality is integrated at the Concrete UI level. This is because the user gestures make sense at this level of abstraction: the user will be able to use gestures to manipulate entities of the concrete UI, such as windows or buttons. The concrete UI will act as a coupling between the WIMP interface toolkit and the gesture recognition toolkit.
- The Voice recognition modality is integrated at the Abstract UI level. This is means that the user will not be able to manipulate entities of the concrete UI with his voice. For example, the user will not be able to vocally ask for a click on a button. Instead, the integration at the abstract layer means that the user will be able to control the abstract UI. For example, the user will be able to vocally ask to add an item to a list. Voice control is almost entirely decoupled from the WIMP UI.

3.4.2.2 Analysis

Most analyses performed on WIMP architecture models can also be applied in the case of multimodal interfaces.

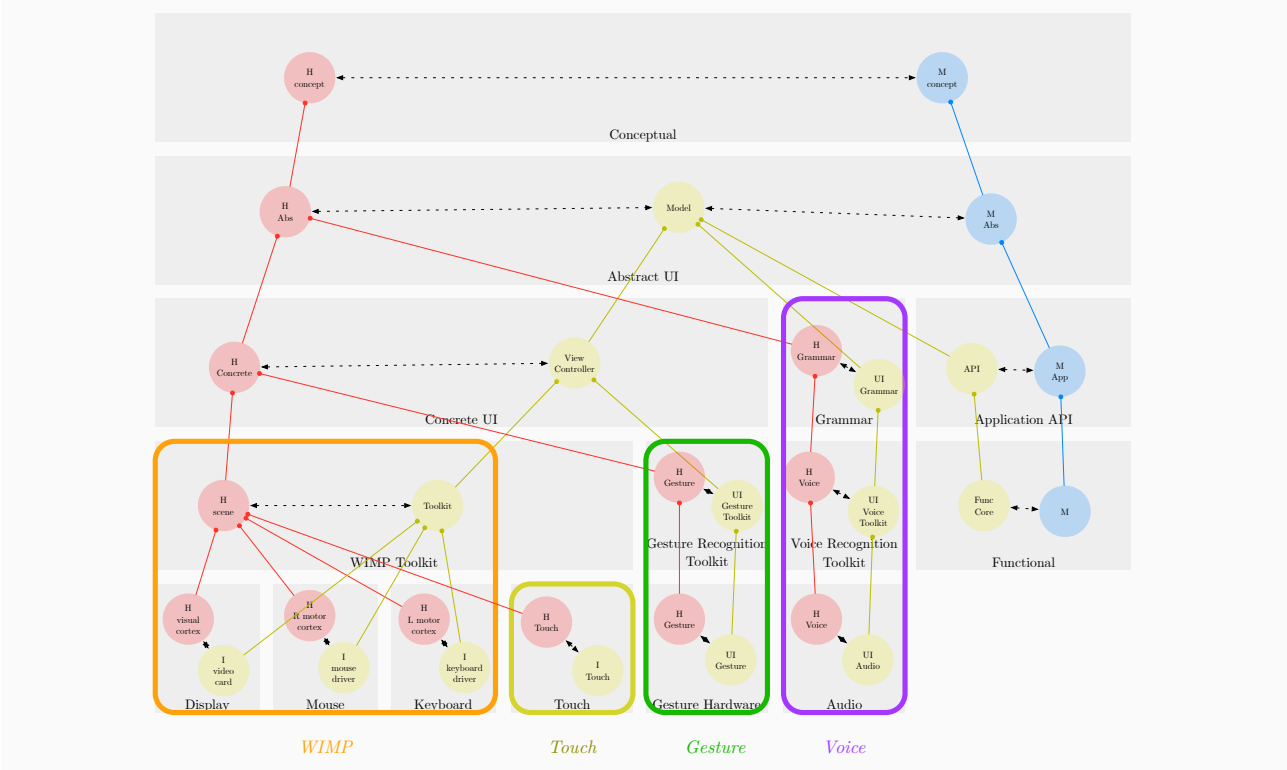


Figure 3.21: A vertical geomorphic view of a multimodal UI architecture

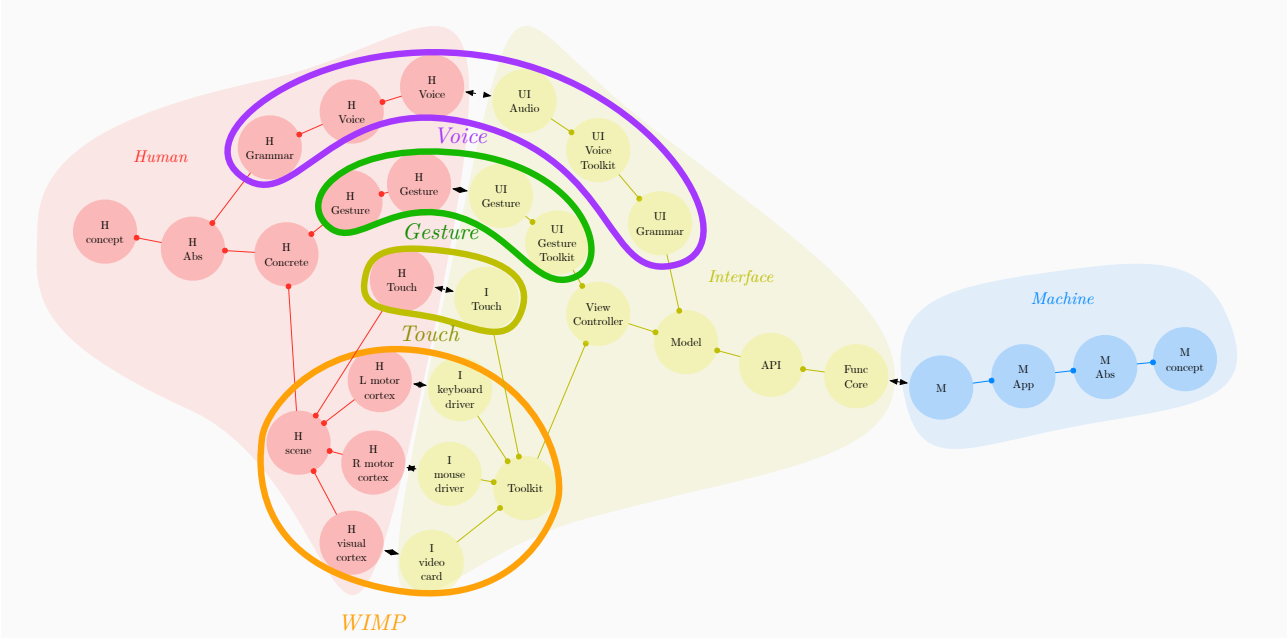


Figure 3.22: An horizontal geomorphic view of a multimodal UI architecture

Redundancy A quick look at Figure 3.22 gives us indications that in most cases, additional modalities can provide increased redundancy to the system. Of course, actual validation of this fact requires further investigation on particular systems implementations, but overall this conclusion is often valid. For example, Smartphones equipped with voice recognition can still be used to an extent when their touchscreen is broken. This is because the additional modalities provide additional pathways for the data flow from the device to the user and back.

Integration costs A quick look at Figure 3.21 shows us that Modalities can be integrated at various abstraction levels. Depending on the capabilities of toolkits, libraries and frameworks used, additional modalities can be added, but depending on their level of integration, the burden of the integration can be left to different developers. For example, many WIMP toolkits now support touchscreens. As a consequence, integration of touchscreens will be a minor burden in most cases. On the other hand, support for gesture-based interfaces is less common, and their implementation is often left at the charge of the application developper and not part of a toolkit or framework. As a result, the integration of gesture recognition will often add to the development cost of UI development.

Usability From the graph on Figure 3.22, and usability information for each node (Data Input Bandwidth, Delay, Mental load on the user...), it is possible to infer the preferred pathways for the user in specific contexts. As shown in Figure 3.23, Users tend to favour optimum modalities according to the given context, the same way they tend to favour the optimum paths when walking from a point to another. For example, on a smartphone, in a context where speed of interaction is the main concern (most of the time), users tend to use touch interactions. But when the touch interaction is not available (e.g. while driving), users tend to use voice input instead.

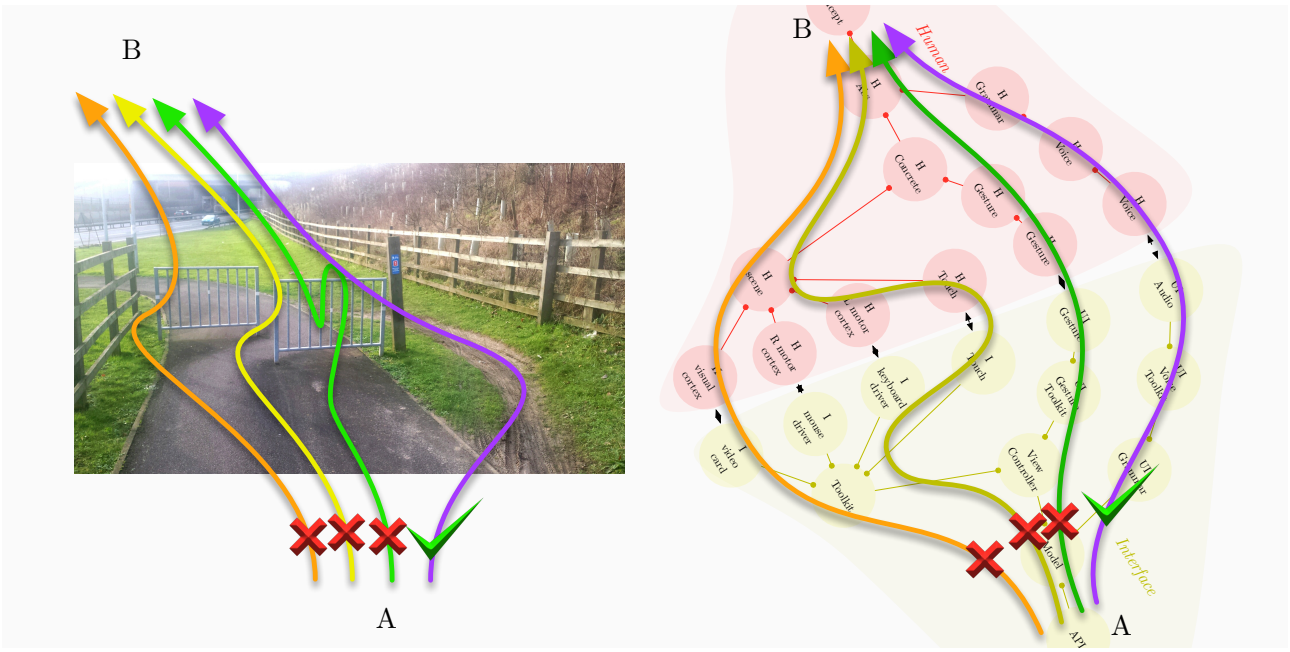


Figure 3.23: A parallel between the choice of the easiest path and the choice of the easiest modality

3.4.3 ARINC661 CDS

3.4.3.1 Description

Layers In this section ,we will use the geomorphic view to describe a particular example of an ARINC 661 compliant system. This system is composed of one UA and two Cockpit Display System (CDS), in order to enable the dialog between one human user and one physical system. The layers are similar to those of WIMP UIs (Subsection 3.4.1), with the addition of several layers:

- American Radio Incorporated (ARINC) 661: the layers in which interactors communicate using the ARINC 661 protocol as defined by the standard [5]. As defined in the standard specification, communications in this layer are bidirectional: *setParameters* messages are sent from the UA to the CDS, and *widgetEvents* are sent from the CDS to the UA.
- Avionics Full DupleX (AFDX): the ARINC 661 standard does not specify underlying protocols. In this case, ARINC 661 is implemented over AFDX. AFDX itself is a stack of several layers which are not described here.
- System protocol: The UA communicates with the CDS on one side, and with a physical system on the other side. This communication with a physical systems follows an application protocol that we are not interested in detailing here. We will just call it the "system protocol"
- ARINC 429: In this example case, the communication with the physical system is implemented using the ARINC 429 Link protocol. Again, there are other layers under the ARINC 429 link layer but we will not detail them here.

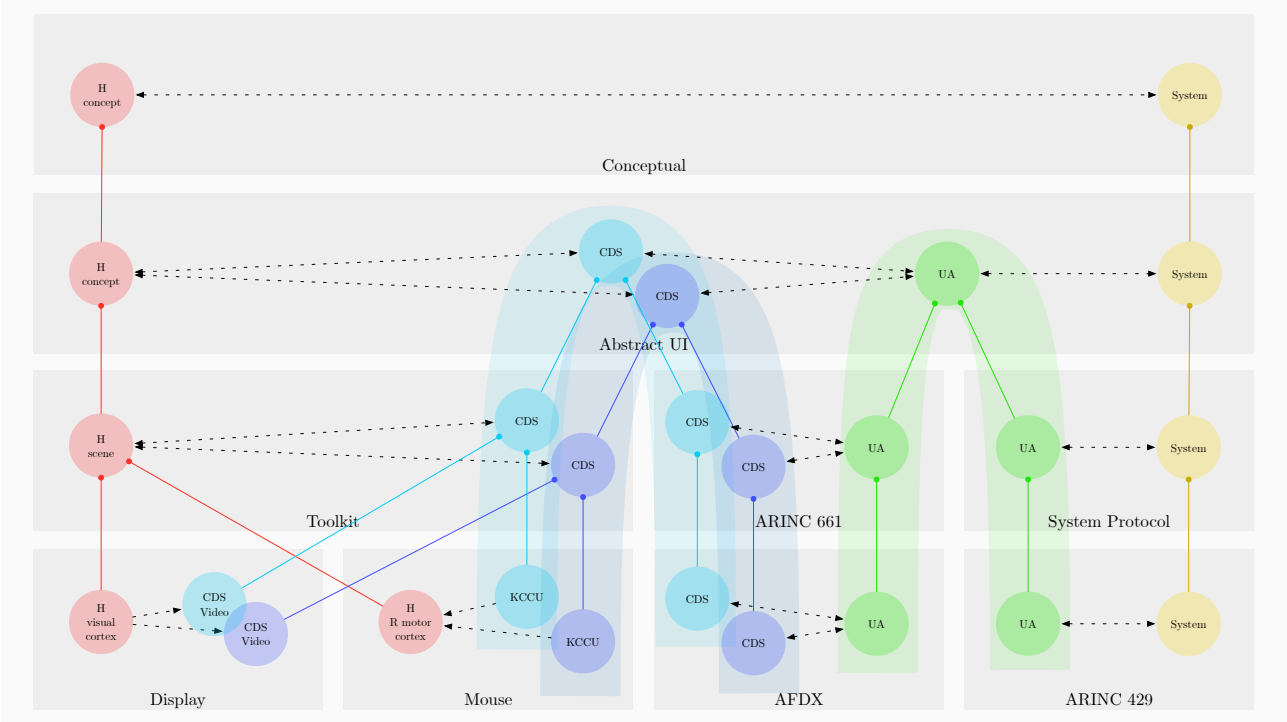


Figure 3.24: A geomorphic view of a configuration with two redundant ARINC 661 CDS

Interactive systems Figure 3.25 presents the same system, with an emphasis on interactive systems this time (horizontal view) In this diagram we see more clearly the 5 different systems involved in this interaction system.

- The human, depicted in red, which has the same role as in the WIMP case, except that here, the

human can interact with two CDS.

- The system, depicted in yellow. At a high level of abstraction, the human is interacting directly with the system, monitoring its status and sending commands to it.
- Two CDS, depicted in blue. Here, the two CDS are redundant, they have the same role, and interact with the same UA and the same User. In actual use cases with many UAs and many users, the situation can be more complex.
- One UA, in green. The UA contains the abstract UI logic, and links the system to the CDS and the user.

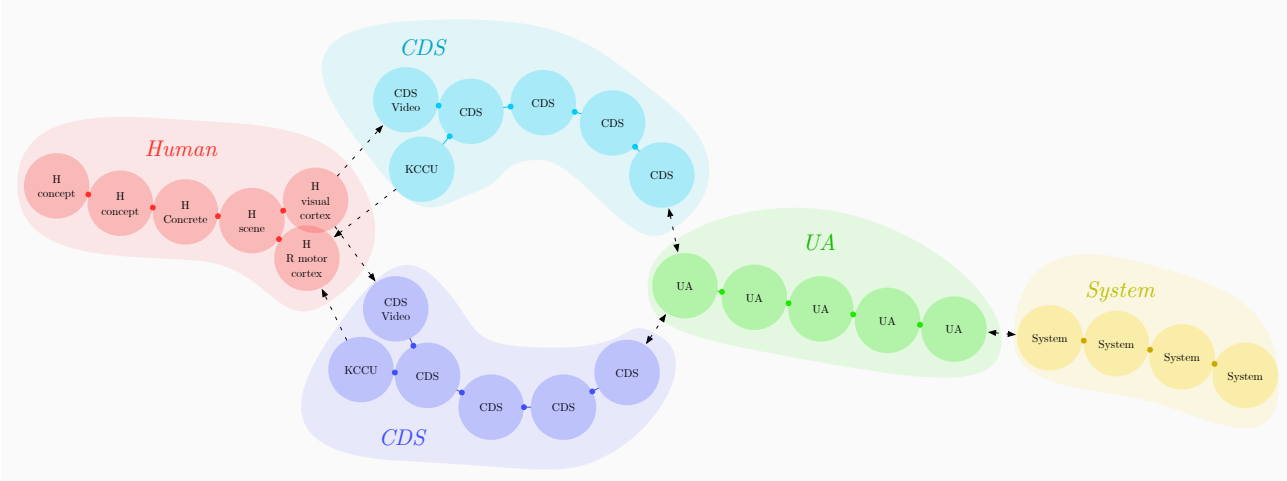


Figure 3.25: Another geomorphic view of a configuration with two redundant ARINC 661 CDS

3.4.3.2 Analysis

Examination of the geomorphic model of ARINC661 systems gives several interesting insights about the standard, its advantages and its limitations.

Several Arch model instances We notice on Figure 3.24 that there are three patterns similar to the arch model (Subsection 2.1.2). Two arches arch represent the architecture of the CDS, while the other arch represents the architecture of the UA. As said in the previous paragraph, the dialogue component on top of the CDS arch does not add any logic to the abstract UI dialogue, but the CDS adds logic to the concrete UI dialogue.

Routing pattern in the Abstract UI layer If we look at the Abstract UI layer in Figure 3.24 and compare it with the Abstract UI layer in a typical WIMP application (Figure 3.16), we notice that the CDS is in fact an additional intermediate between the system and the user. In the WIMP case, there was only one intermediate between the machine and the user: the UI. In the ARINC 661 case, we now have two intermediates: the UA, and the CDS. However, this additional intermediation does not add much complexity to the interaction in the abstract UI layer, because the role of the CDS is only to forward the abstract UI coming from the UA, to the user. The ARINC standard specifies that the CDS does not add any dialog logic to the abstract interface. The only role of the CDS is to faithfully represent the abstract UI to the User, without transforming it. In this sense, we can say that the role of CDS in the abstract UI layer follows the "routing" pattern, explained in Subsection 3.3.6.

Increased flexibility Since the UI is separated in two kinds of subsystems (UA and CDS), there is a good decoupling between two parts of the UI. For example, it is possible to upgrade or change

the CDS without touching the UA, or to upgrade the UA without touching the CDS. The ARINC standard ensure that the interface between these elements is always well defined. However, this comes at the price of increased design complexity: the whole ARINC 661 layer and implementation over a network protocol has to be taken into account during the design phase.

Redundancy We saw earlier that multimodality can sometimes increase the redundancy of systems, by increasing the number of channels between two interactive systems. In the case of ARINC 661 systems, additional redundant pathways are created by using several CDSs redundantly in order to reduce the probability of a hazardous situation. If one CDS fails, other CDSs can still be used to perform the same task, greatly increasing safety.

Increased behaviour complexity Figure 3.26 shows several feedback loops which are present in a ARINC 661 system with CDS redundancy. As compared to the WIMP example (Figure 3.20), we see that the number of feedback loops increases a lot. Care has to be taken that these feedback loop do not interact in unexpected ways. In actual use cases, several UA are interacting with several CDSs, making the situation even more complex. Thankfully, the ARINC standard enforces strict segregation between UA, lowering the possibility of interferences between the behaviour of several UAs.

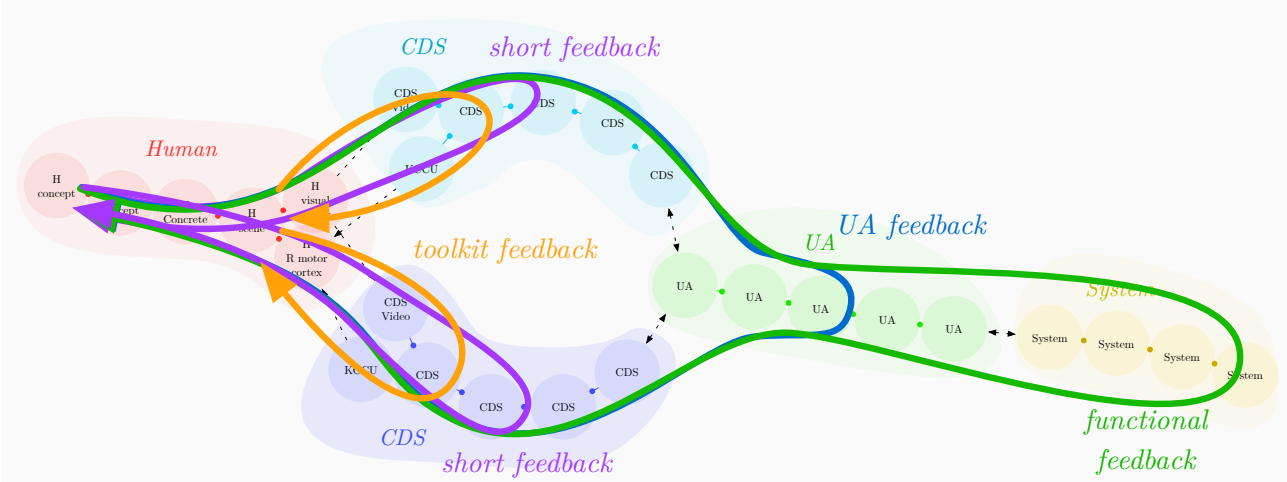


Figure 3.26: Feedback loops in an ARINC 661 system

Non-obvious separation of concerns The behaviours of WIMP UIs are relatively clearly separated: Behaviours which are purely related to the dialogue control and not related to the system are managed by the UI. Some examples are the scrolling of a list or toggling the state of a button when clicked. The behaviours that are related to actual interaction between the human and the system are managed by the system. For example when a pilot asks for landing gear retraction, the feedback comes when the landing gear actually retracts. In a ARINC 661 system, the added intermediates make this situation a bit more complex: Toggling the state of a push button is managed by the CDS, but choosing which panel to display is managed by the UA. The separation line between "low level" behaviours managed by the CDS and "high level" behaviours managed by the UA is somewhat arbitrary, but has an important impact on the performance of the system.

Increased delays As shown in Figure 3.26, some feedback loops go through a complex path: In some cases, human actions have to go all the way through the CDS, then over the AFDX network to the UA, and then back on the AFDX network to the CDSs before finally reaching the user again.

These numerous steps add delays, which can quickly become a bottleneck in real use cases, and are inherent to the architecture of ARINC 661 systems.

3.4.4 Web application

3.4.4.1 Description

In this section, we will use the geomorphic view to describe the classical web UI with a client-server architecture.

Layers The layers are similar to those of WIMP UIs (Subsection 3.4.1), but this time, communication between the UI (Client) and the machine (Server) is implemented differently. This communication involve several layers:

- Application API: This layer represent the communication between the client and the server at a high level of abstraction, this layer is often called the Application API. Many modern applications use the Representational State Transfer (REST) API model in this layer [121].
- HTTP: HTTP is the only protocol that is widely supported for web client-server communication. As a consequence, the application API has to be implemented over the HTTP protocol.
- TCP/IP and other internet layers: As described in [49], HTTP communication is implemented over a complex stack of protocols layers which form the internet architecture. For the sake of clarity, we only represent one layer here, but in reality, many layers are present under this one, down to the physical layer, for further reference about the geomorphic representation of these layers, see [49].

Interactive systems Figure 3.28 presents an horizontal view of the same system. We see that this model is similar to the WIMP model (Figure 3.17). The difference is that this time, additional systems (Part of the internet infrastructure) play the role of intermediates between the Client and the Server.

3.4.4.2 Analysis

Examination of the geomorphic model web applications gives some interesting insights.

Physical decoupling The first thing to notice is that, as compared to WIMP UIs, Client and Server are further decoupled by the addition of several intermediates: Systems that are part of the internet infrastructure, (Internet routers, Gateways...). These systems act as a bridge between the Client and the Server, allowing a physical and geographical decoupling between them. This is basically the concept of the internet: the actual business logic only has to be implemented in one place (the server), but clients can use the system from many different places.

Usability concerns Figure 3.29 shows two feedback loops at the abstract UI level in a Web UI. These two feedback loops are the same as described for WIMP UIs. However, in the case of Web applications, they have the following specificities:

- The *short feedback* loop is performed on the client, by executing Javascript code. This loop can be fast, as no network activities are required.

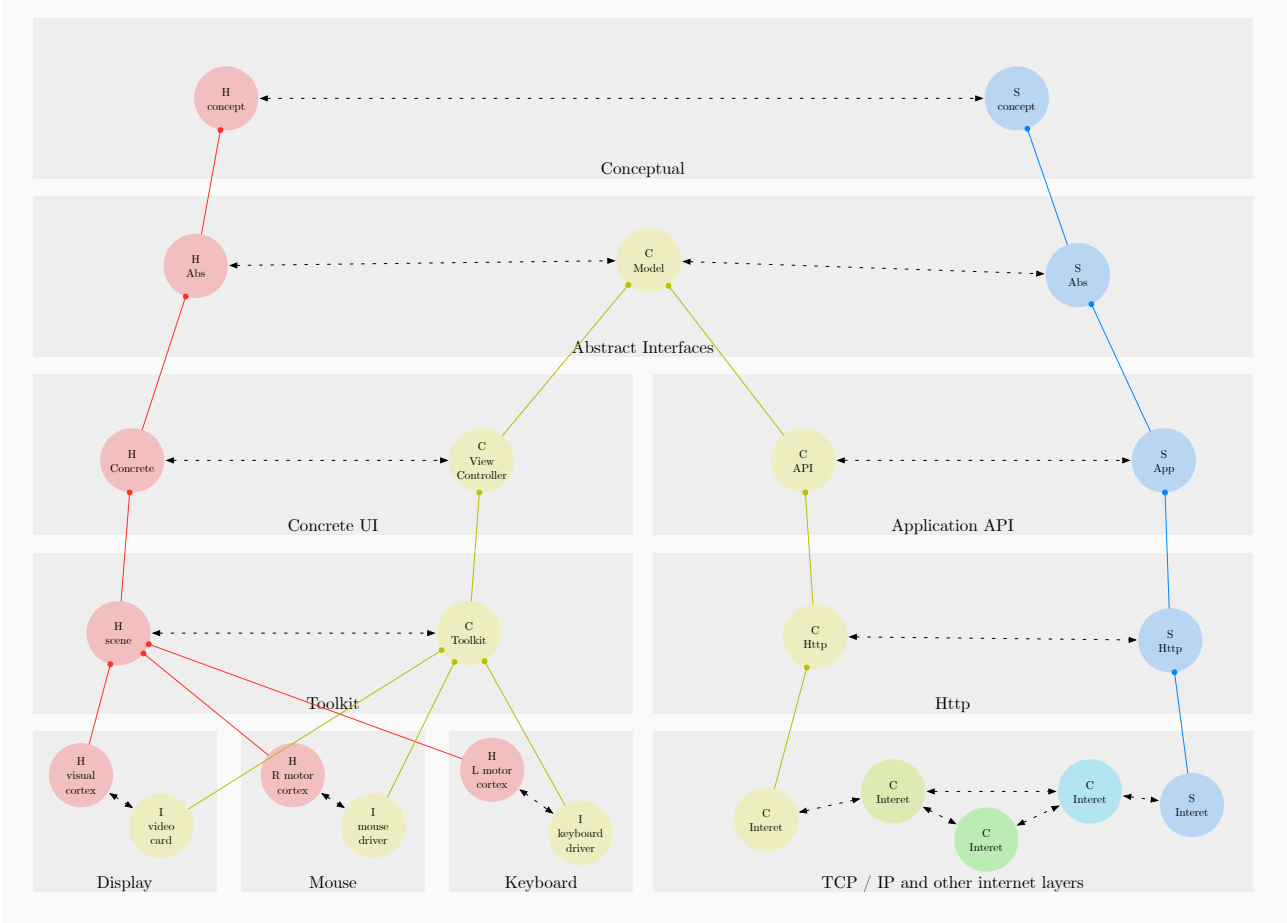


Figure 3.27: A vertical geomorphic view of the typical architecture of web applications

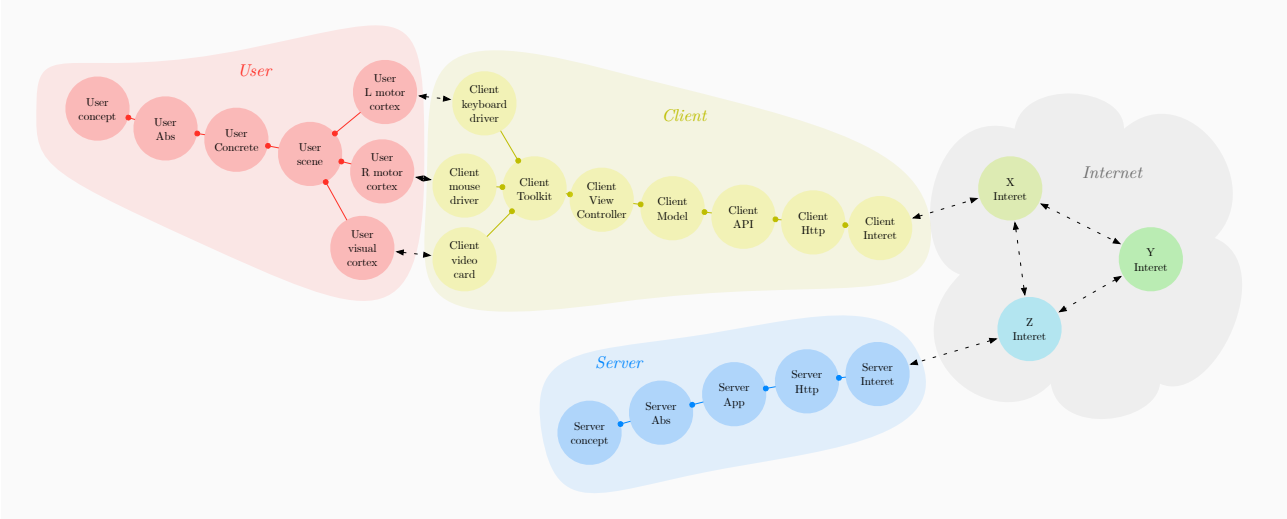


Figure 3.28: An horizontal geomorphic view of the typical architecture of web applications

- The *functional feedback* loop is performed on the server. This loop involves network activities, it goes through the internet back and forth. This causes this loop to present a long delay, and in cases where network connexion is lost, it can even have infinite delays.

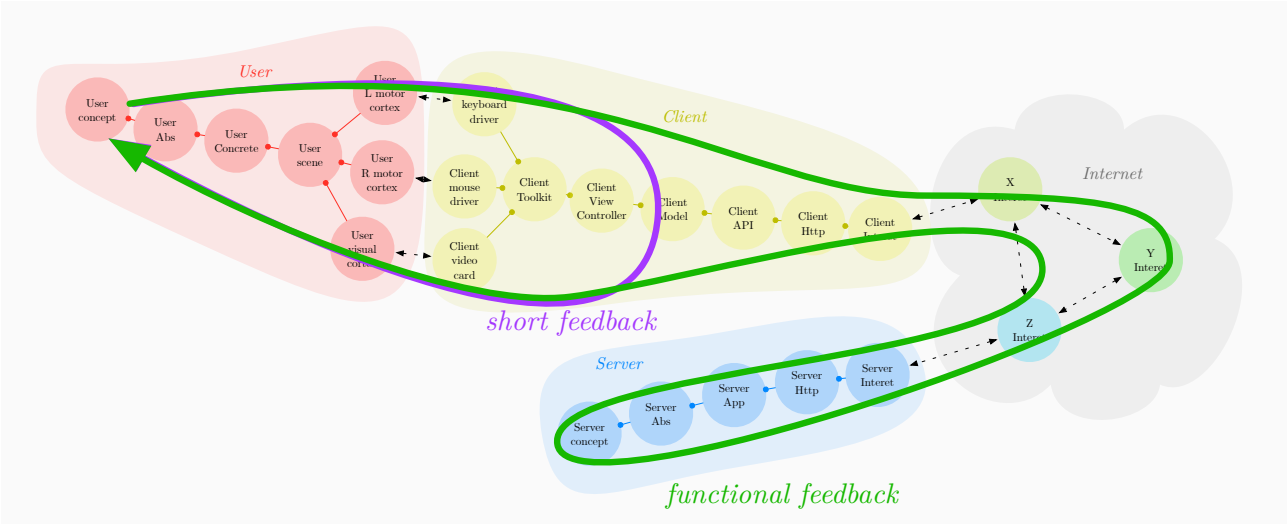


Figure 3.29: Two feedback loops in a Web UI

The large delay difference between these two feedback loops means that in order to increase usability, web applications have to try to add as much logic as possible on the client side, in order to provide short response times to the user. Historically, the trend is clearly towards an increasingly autonomous client application: older applications built using PHP: Hypertext Preprocessor (PHP) have the UI logic almost integrally implemented on the server side, while newer applications are built in an "offline-first" philosophy: they are built to work almost 100% without access to the network, all the UI logic is implemented on the client side. This trend gave rise to several means to decouple the client from the server. For example, databases such as PouchDB allows the clients to have local instances of parts of the server database, allowing to perform database operations locally, so that the delay of synchronisation with the server has no impact on the delays perceived by the user. This results in the architecture shown on Figure 3.30. As of 2015, this kind of architecture seems to be developing fast as a solution to the problem.

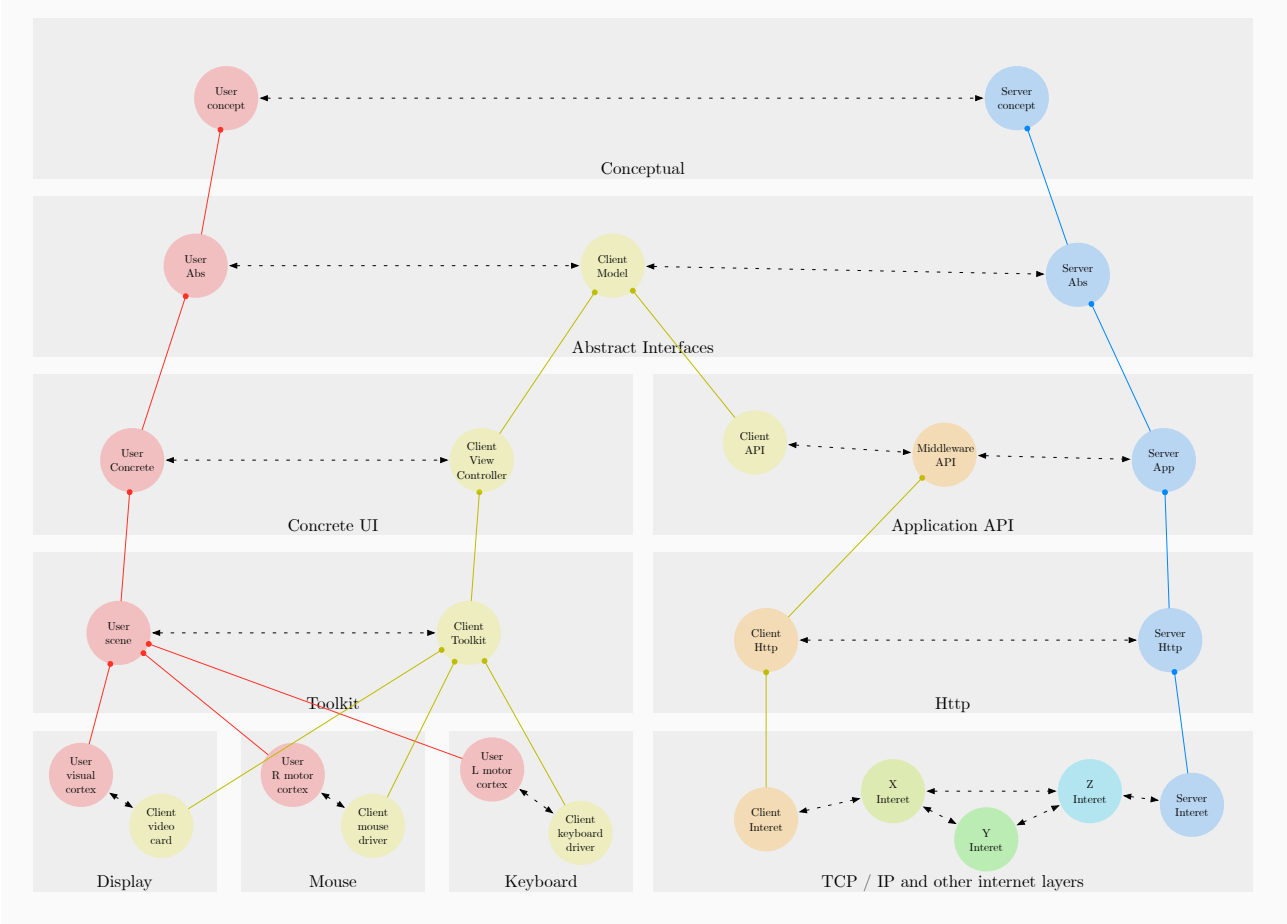


Figure 3.30: A common architecture for modern web UI with use of a local cache to reduce feedback delays

3.5 Conclusion

3.5.1 Limitations

The geomorphic model is only a static representation of two aspects of interactive systems: The notion of abstraction level, and the notion of architectural composition. It leaves out many architectural aspects of interactive systems. The geomorphic view of UIs suffers from many limitations:

- It is a global model [28], as such, it does not prescribe the internal architecture of interactors.
- The model does not say anything about the interactions performed by interactors.
- The actual protocols and models of communication within layers is left out.
- Timing constraints are not expressed.
- Dynamic reconfiguration of interactors is not expressed.
- The model is not formal. While the geomorphic view of networking has a formal model [49], the geomorphic view of UIs does not have one yet.

3.5.2 Compliance with requirements

The geomorphic view of interactive systems provides a way to express the architecture of complex interactive systems in a way which is relatively easy to understand (Requirement 3 (Simple)). By clearly segregating various parts of UI architecture, it could enhance traceability during the design of complex interaction systems (Requirement 5 (Traceability)). The geomorphic view of interactive systems provides a small set of abstractions to describe interactive systems at a high level (Requirement 8 (Abstraction)). But the main advantages of the geomorphic view is that it provides a way to manage the various abstraction layers (Requirement 9 (Manage)), and it is highly flexible and open to the description of novative UI architecture patterns (Requirement 12 (Flexibility)).

On the downside, the geomorphic view leaves out several requirements: it is not formally defined (Requirement 1 (Formal)) and does not really lend itself to verification techniques (Requirement 11 (Verification)). Apart from segregating the components that belong to different abstraction levels, it is also not really well suited to express the composition of interactors (Requirement 6 (Composition) and Requirement 7 (Modularity)).

Finally, some requirements are out of the scope of an architecture model for UIs: Requirement 2 (Collaborative), Requirement 4 (Projection), Requirement 10 (Prototyping).

3.5.3 Perspectives

Since the geomorphic view of interactive systems is relatively close to the geomorphic view of networking, merging the two models in a consistent way could open the way for a unified way for describing both UI and networking systems in a unique view. This would allow to globally perform verification of properties on systems that involve a mix of network systems and interactive systems. Since this kind of systems are really common, the application could be interesting.

Addition of further information to interactor nodes could make the model prone to the verification of some properties, such as feedback loop delays (addition of all delays in a feedback loop) or safety analyses (probability of failure of a UI pattern, given the probability of failure of each node). However, more useful verification necessarily involve the knowledge of the behaviour of each interactor, which is out of the scope of the geomorphic model.

As said earlier, the presented model says nothing about the behaviour of interactors. This part is left to the next chapter, where we present a language to describe the interactions performed by interactors.

Chapter 4

LIDL

If I had asked people what they wanted, they would have said faster horses
Henry Ford

4.1 Getting started

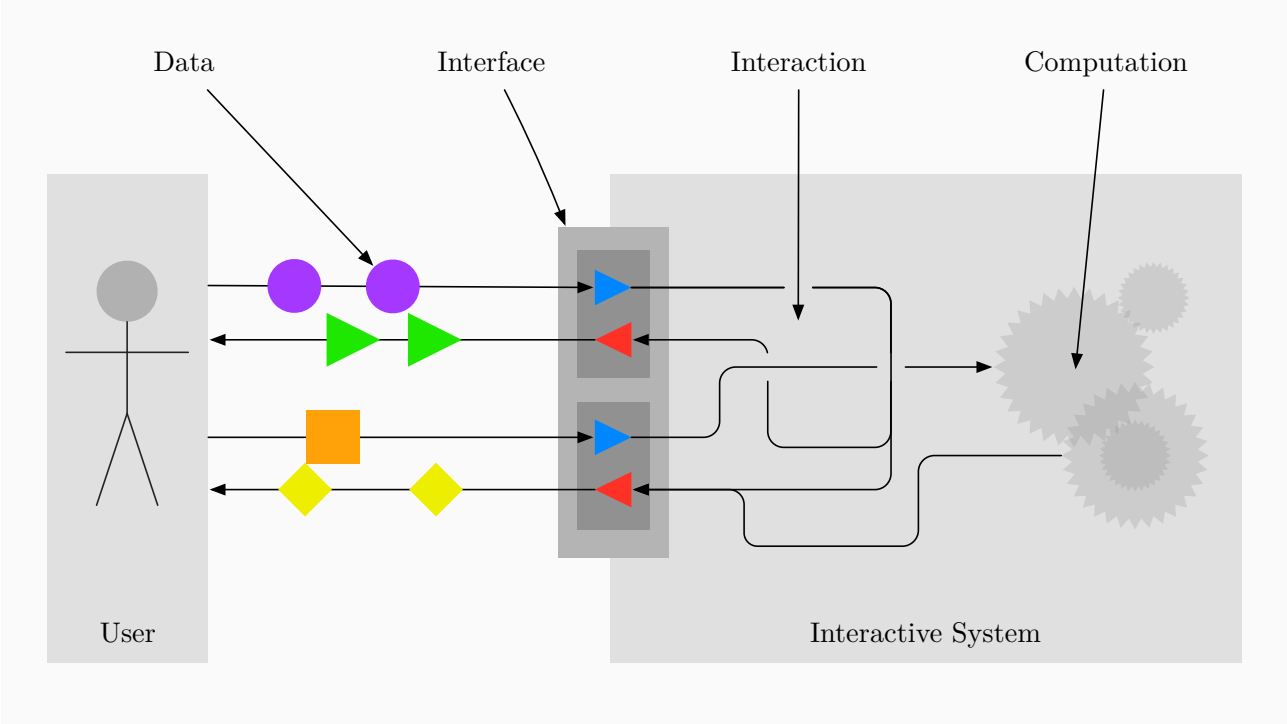


Figure 4.1: An overview of the four components of interactive systems

Figure 4.1 presents a schematic view of the four main ingredients needed to describe interactive systems:

- Data: Description of the data types, the structure of the objects that are exchanged between the interactive system and other systems or human users.

- Interfaces: Structured description of the various data flows offered to the system to communicate with other systems, or the user: How is the data exchanged? Which data is sent by who to who? Which data is received?
- Interactions: Description of the dynamic behaviour of the system: What happens when the system receives certain signals? At which condition does the system sends this signal? What is the link between this input and this output?
- Computations: Description of computations that are performed by the system. By computation, we mean calculations performed by the system, and which are not related to the organisation of its interaction with the outside world.

LIDL focuses on the description of parts which are specific to interactive systems: interfaces and interactions. LIDL also allow a limited description of data types, and relies on other programming languages to describe computations.

4.1.1 A first simple example

LIDL is a language that allows to describe three aspects of interactive systems: data exchanged between interactive systems, interfaces between interactive systems, and interactions performed by interactive systems. As a consequence, there are three kinds of expression in LIDL: Data type expressions, Interfaces expressions, and Interaction expressions. Let us put these to use in order to describe a very simple example. The specification of this example is simple:

A system that constantly outputs the number 42

The first thing to describe is data types involved in the system. This system deals with numbers, and LIDL has the basic data type `Number` to represent numbers. We can write the following data type expression:

```
1 Number
```

We can now describe the interface of our system. An interface is represented as a combination of data types and data flow directions (`in` or `out`). In our case, the system *outputs* a *number*, so its interface is described by the following interface expression:

```
1 Number out
```

Finally, we can describe the interaction performed by this system. Interactions describe the process by which interactive systems take data as input and return data as output. Interactions are associated with data flows. Each interaction has an interface (a combination of data type and direction). In our case, the interaction performed by the system must comply with the interface `Number out`. The interaction performed by the system can be described very simply:

```
1 (42)
```

Interactions are easy to spot in LIDL programs: they written between parentheses. It is important to understand that interactions do not denote values, but data flows and the processes that generate them. For example the expression above does not directly represent the value 42. Instead, this expression represents the interaction which consists in constantly outputting the value 42.

We can now put all this together and write the definition of our system. Definitions use the keyword `is` in order to associate an interaction signature with an interaction expression. Here, we will call our interaction `My simple interaction`:

```
interaction
  (My simple interaction): Number out
is
  (42)
```

LIDL interactions are executed synchronously. At each instant, interfaces with the `in` direction receive values, while `out` interfaces send values. In our case, there is only a `Number out` interface. At each instant, the interaction defines the current value send through this `Number out` output. Execution of our interaction will yield the following data flow:

Instant	(My simple interaction)
0	42
1	42
2	42
3	42

4.1.2 A second example

We are now going to describe a simple UI using LIDL. As shown in Figure 4.2, this interactive system is made of a number input (called theNumber), and a number output (called theResult). The specification of the behaviour of this system is simple:

The result displayed should be equal to the number given by the user incremented by one.

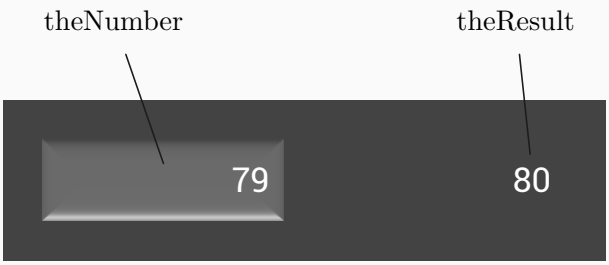


Figure 4.2: A simple UI

We will use LIDL to describe the abstract UI of this system. The abstract UI of this system is simple: there is a number input and a number output. We can describe this interface using an interface expression:

```
{
  theNumber:  Number in,
  theResult:  Number out
}
```

This interface expression denotes an composite interface, made of two basic interfaces labelled `theNumber` and `theResult`. `theNumber` is an input (it represents the number given by the user to the system), and

`theResult` is an output (it represents the number returned by the system to the user). The interface of our system with the user is a composition of these two interfaces. In order to be able to reuse this interface without having to write the whole expression each time, we are going to name this interface `SimpleUI`, and write its definition:

```
interface
  SimpleUI
is
  {
    theNumber:  Number in,
    theResult:  Number out
  }
```

This definition means that we can now use `SimpleUI` as a shorter synonym for `{theNumber: Number in, theResult: Number out}`.

Now that we have described the interface of our system with the user, it is time to describe the interaction performed by our system, the process of how it handles the values going through this interface. The interaction has to comply with the newly defined interface `SimpleUI`, and we call this interaction `My simple UI interaction`:

```
interaction
  (My simple UI interaction):SimpleUI
is
  ({
    theNumber:  (x)
    theResult:  ((x)+(1))
  })
```

This interaction expression is a composition of several interactions. The main interaction is a composition interaction. It has a syntax similar to the interface syntax: the use of braces, and labels followed by colons, but note that it is surrounded by parentheses which denotes that it is an interaction. This interaction complies with the `SimpleUI` interface, and takes two sub interactions: one for the field called `theNumber` and another for `theResult`. The interaction associated with the field `theNumber` is `(x)`. This means that numbers received by the interaction will be sent to a signal that we arbitrarily called `(x)`. The interaction associated with the field `theResult` is `((x)+(1))`. This means that the interaction will output a value which is the result of the addition of `(x)` and `(1)`. To recap, we have written the following LIDL program:

```
interface
  SimpleUI
is
  {
    theNumber:  Number in,
    theResult:  Number out
  }

interaction
  (My simple UI interaction):SimpleUI
is
  ({
    theNumber:  (x)
    theResult:  ((x)+(1))
  })
```

This program can be executed. Here is a table presenting the result of an execution of this interaction, where red text denotes output, blue denotes input, and black denotes interactions which are used as input in some parts of the program, and as output in different parts of the program.

Instant	(My simple UI interaction)	(x)	(1)	((x)+(1))
0	$\{theNumber : 10, theResult : 11\}$	10	1	11
1	$\{theNumber : 20, theResult : 21\}$	20	1	21
2	$\{theNumber : 0, theResult : 1\}$	0	1	1
3	$\{theNumber : 0.1, theResult : 1.1\}$	0.1	1	1.1
4	$\{theNumber : -100, theResult : -99\}$	-100	1	-99

4.2 Presentation of the language

4.2.1 Definitions

LIDL programs are described by creating definitions. Definitions use the keyword **is** in order to associate a signature with an expression. There can be three kinds of definitions: data type definitions, interface definitions, and interaction definitions. Here are example of definitions of these three kinds:

```
data
  MyDataType
is
  ...

interface
  MyInterface
is
  ...

interaction
  (My interaction of (x:AnInterface) and (y:OtherInterface)): MyInterface
is
  ...
```

Nested definitions Definitions can be nested: any definition can contain other definitions, using the key word `with`. Here is an example of nested interaction definitions:

```
interaction
  (X)
with
  interaction (Y) is ...
  interaction (Z) is ...
is
  ...
```

4.2.2 Data

Types LIDL provides four basic data types: `Activation`, `Boolean`, `Number` and `Text`. `Activation` is a unit type: it only has one possible value, named *active*. This data type may seem useless, but we will see later that it does have a use. `Boolean`, `Number` and `Text` are encountered in many other programming languages.

LIDL offers a way to define and use additional basic data types, by defining them in the target programming language that LIDL programs compile to. For example, the following expression defines a LIDL data type named `Graphics` by referencing a data type called `SVGGraphics` in the target programming language:

```
data
  Graphics
is
  native SVGGraphics
```

LIDL offers a way to define and use composite data types. For example, the following expression defines a composite data type named `Point2D` containing two numbers labelled `x` and `y`:

```
data
  Point2D
is
  {
    x: Number,
    y: Number
  }
```

Activation All data in LIDL programs has a notion of activation. This means that all data types have an additional value, named *inactive*, that denotes the absence of data. As an example, the value of a `Number` can be 0, 1, 3.14 or *inactive*. This means that, assuming we defined an interaction `(x)` of the type `Number`, an execution of the interaction can lead to the following value sequence:

Instant	(x)
0	*inactive*
1	*inactive*
2	1
3	3.14
4	*inactive*
4	0

4.2.3 Interfaces

Input ports and output ports of LIDL systems are specified by definition of their interfaces. The names of these ports, as well as the data types that goes through them, are specified in the interface. Interfaces are not specific to a system, several different systems can implement the same interface, they are just structural descriptions.

Atomic interfaces The basic building blocks of interfaces are atomic interfaces. They are interfaces which are defined by a data type, and a direction. For example, here is the definition of an atomic interface, that describes a numeric input:

```
interface
  MyNumberInputInterface
is
  Number in
```

Composite interfaces LIDL interfaces are composable: it is possible to create arbitrarily complex interfaces by composition, using the syntax for composite interfaces. For example, here is the definition of an interface that is composed of an input labelled `theNumber`, and an output labelled `theResult`.

```
interface
  MyCompositeInterface
is
  {
    theNumber:  Number in,
    theResult:  Number out
  }
```

Conjugation Each interface has one and only one conjugate interface: the interface which is similar to it, but where all directions are reversed. Conjugate interfaces are denoted by appending a prime character (') to the name of interfaces. For example the interface `MyCompositeInterface'` is the conjugate of `MyCompositeInterface`. The definition of `MyCompositeInterface'` would be:

```
interface
  MyCompositeInterface'
is
  {
    theNumber:  Number out,
    theResult:  Number in
  }
```

Compatibility of interfaces Interfaces are to LIDL what data types are to most programming languages: each interaction expression has an interface, and expressions can only be composed if their interfaces are compatible. The notion of compatibility of interfaces is a bit more subtle than the notion of compatibility of data types though: In order to be linked, two interactions must have *conjugate* interfaces. This might seem strange (why not the *same* interface ?), but it is actually straightforward: in order to connect two systems, input of the first must be output of the other, and vice-versa.

For example, consider the following interaction:

```
( (a) = (b) )
```

This interaction is valid only if `(a)` and `(b)` have conjugate interfaces. If `(a)` has the interface `Number in` then `(b)` must have the interface `Number out`. This situation means that `(a)` receives the value output by `(b)`. If both interactions `(a)` and `(b)` had the same interface, say `Number in`, then this statement would make no sense.

4.2.4 Interactions

Interactions are expressions that describe the behaviour of interactive systems. They are the most important part of LIDL.

Syntax The syntax for interactions is flexible: it is not prefix, not suffix, not infix, but would be better described as multifix. Here is an example of these several notations options:

This flexibility of the notation imposes one constraint: each interaction in a LIDL program has to be surrounded by parentheses, as this allows to clearly separate description of the operators from operands. This makes LIDL interaction expression rather parentheses heavy, but this issue has its own solution.

Syntax	Example
prefix	myoperator x y
suffix	x y myoperator
infix	x myoperator y
multifix (LIDL)	description of my operation where the operands (x) and (y) can be anywhere

Table 4.1: Several notation options

Referential transparency LIDL is referentially transparent: two identical expressions will always have the same value, they actually represent the same object. For example, consider the following interaction:

```
( all
  ( (y) = (my function (x)) )
  ( (z) = (my function (x)) )
)
```

In this expression, it is guaranteed that at every instant, `(y)` and `(z)` will have the same value, whatever `my function` is.

Resolving It is common to write LIDL code were different values are sent to the same interaction. For example, consider the following interaction:

```
( all
  ( (x) = (a) )
  ( (x) = (b) )
)
```

This interaction means that, at each instant, the variable `(x)` receives the value from `(a)` and also the value from `(b)`. This might seem inconsistent: what is the value of `(x)` if at a given instant, `(a)` and `(b)` have different values ? The solution to this problem is linked with the notion of data activation (Subsection 4.2.2). Data activation denotes the presence or absence of data, and each LIDL interaction includes a notion of activation. As a consequence, four situations can happen in the interaction above. If both `(a)` and `(b)` are inactive, then `(x)` is inactive: nothing was sent, nothing was received. If either `(a)` or `(b)` is active, then `(x)` receives the value of `(a)` or `(b)` respectively. Finally, if both `(a)` and `(b)` are active, then we fall in the problematic condition where two different values are sent to `(x)`, and a runtime error is thrown.

The process of finding which value to give `(x)`, given the values of `(a)` and `(b)`, is called resolving. LIDL resolving strategy is:

When an interaction receives several values, all of them but at most one must be inactive. If more than one input value is active, there is a runtime error.

Here is an example of execution of the interaction above:

Instant	(a)	(b)	(x)
0	*inactive*	*inactive*	*inactive*
1	5	*inactive*	5
2	*inactive*	*inactive*	*inactive*
3	*inactive*	6	6
4	4	2	*ERROR*

4.2.5 Execution

Description LIDL programs are executed synchronously. Interactions represent data flows, and each of these data flows is assigned a value at every instant. As shown in Figure 4.3, at each instant, the previous state and current input are processed by a transition function in order to generate the next state and current output. The initial state is generated by an initialisation function. Both of these functions are described by the semantics of LIDL programs. LIDL programs denote Mealy FSMs [122]. The correspondence between LIDL programs and Mealy machines is given by Table 4.2.

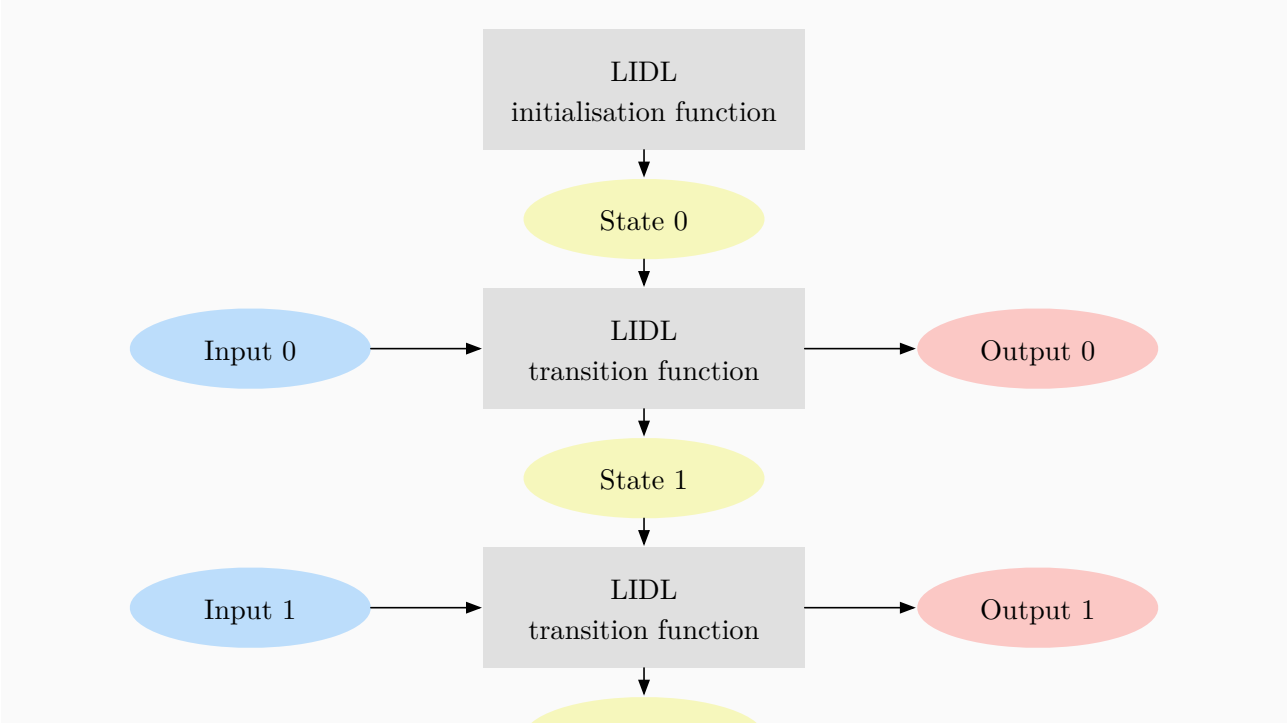


Figure 4.3: Execution of LIDL programs

Example For example, consider the interaction of Listing 4.1. This interaction has one input: `theInput`, one output: `theOutput`, and one state variable: `theState`. At each execution step, the interaction is executed synchronously: `theInput` is given a value by the user, `theState` may be incremented depending on the value of `theInput`, and `theOutput` is given the *true* or *false* value depending on `theState`. Table 4.3 presents the trace of an execution of this program, while Figure 4.4 presents the corresponding Mealy machine, where edges are labelled with the value of the input (shown in blue) and the value of the output (shown in red), and states are labelled with the value of the state variable. Note how the trace of Table 4.3 could have been generated by the Mealy machine.

Mealy machines		LIDL programs
State space	S	Possible values of the interactions that represent state variables
Input alphabet	Σ	Possible values of input ports of the system's interface
Output alphabet	Λ	Possible values of output ports of the system's interface
Transition and Output function	$T : (S \times \Sigma) \to (S \times \Lambda)$	Transition function inferred from interactions
Initial state	S_0	Return value of the Initialisation function inferred from interactions

Table 4.2: Correspondence between Mealy machines and LIDL programs.

```
interaction
  (my system): { theInput: Activation in, theOutput: Boolean out}
is
  ({
    theInput  : ( increment (theState) )
    theOutput : ( (theState) is a multiple of (3) )
  })
```

Listing 4.1: An interaction with one input, one state variable and one output

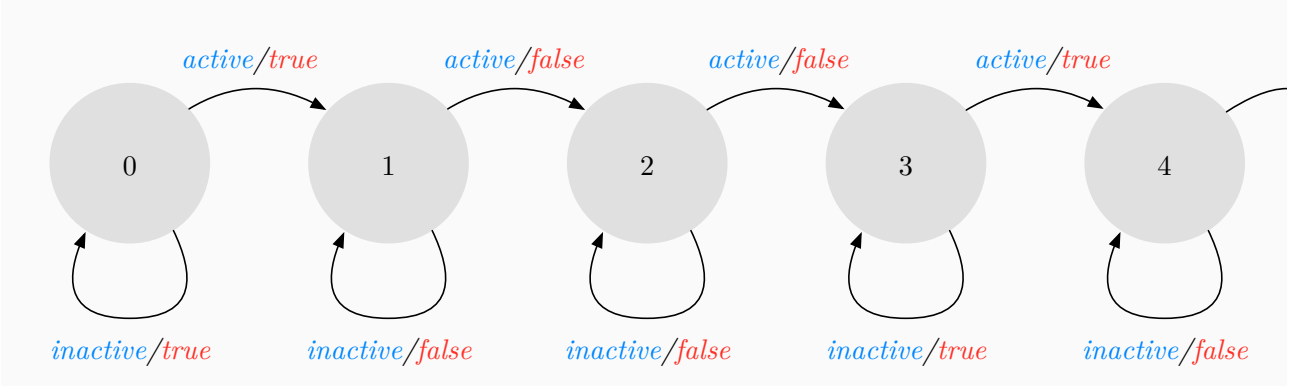


Figure 4.4: A mealy machine corresponding to the program of Listing 4.1

Instant	(my system)	(theState)
0	$\{theInput : inactive, theOutput : true\}$	0
1	$\{theInput : active, theOutput : false\}$	1
2	$\{theInput : inactive, theOutput : false\}$	1
3	$\{theInput : inactive, theOutput : false\}$	1
4	$\{theInput : active, theOutput : false\}$	2
5	$\{theInput : active, theOutput : true\}$	3
6	$\{theInput : active, theOutput : false\}$	4
7	$\{theInput : inactive, theOutput : false\}$	4

Table 4.3: Timing diagram of an execution of the LIDL program of Listing 4.1

4.2.6 Base interactions

LIDL offers a set of built-in basic interactions that are sufficient to describe all possible LIDL interactions. This set of basic interactions is relatively restricted. There are seven kinds of basic interactions:

- Literals: constantly output a given value
- Variables: share values between interactions.
- Composition: compose other interactions in order to form a more complex interaction
- Affectation: affect the output of an interaction to the input of another
- Function application: apply a function to data
- Previous: output values that were input at previous instants
- Behaviour: add behaviours to an interaction

The following paragraph details these seven kinds of basic LIDL interactions.

4.2.6.1 Literals

All literals related to the four basic data types (`Activation`, `Boolean`, `Number` and `Text`) are supported by LIDL. Function literals that refer to functions coded in the target programming language can be used, with the keyword `function`. For example here are some literals interactions:

```
(active)
(true)
(-23.16)
("Hello world !")
(function sin)
```

Literals constantly output the same value. They comply with their respective `out` interface. For example, the literal `(-23.16)` complies with the `Number out` interface.

4.2.6.2 Variables

Variables are interactions that have no definition, they can be used in various places of the code to refer to the same value. What is input to a variable in some parts of the program is be output from it in other parts of the program. For example, the following piece of code uses the variable `(x)`, and it guarantees that `(y)` will have the value `(5)` at each instant:

```
( (y) = (x) )
( (x) = (5) )
```

4.2.6.3 Composition

Composition interactions allow to group several interactions by creating a composite interface. Their syntax is based on braces and labels followed by colons. For example consider the following interaction:

```
({
  firstInput   : (x)
  secondInput  : (y)
  theOutput    : ((x)+(y))
})
```

This interaction allows to compose three other interactions `(x)`, `(y)` and `((x)+(y))`, by creating a composite interaction that complies with the following interface:

```
{
  firstInput   : Number in,
  secondInput  : Number in,
  theOutput    : Number out
}
```

An execution of this interaction would lead to the following timing diagram, where we note that the composition interaction has created a composite interface from a set of simple interfaces.

Instant	({ firstInput:(x) secondInput:(y) theOutput:((x)+(y) })	(x)	(y)	((x)+(y))
0	{*firstInput* : 2, *secondInput* : 3, *theOutput* : 5}	2	3	5
1	{*firstInput* : 10, *secondInput* : 1, *theOutput* : 11}	10	1	11
2	{*firstInput* : *inactive*, *secondInput* : 3, *theOutput* : *inactive*}	*inactive*	3	*inactive*

4.2.6.4 Affectation

The affectation interaction performs the equation between its two sides when it is active. For example, here is an affectation interaction:

```
((y)=(x))
```

This interaction complies with the `Activation in` interface. This means that it constantly receives an activation signal. When the activation signal is *active*, then the affectation interaction receives the signal from `(y)`, and sends it to `(x)`. When the affectation interaction receives the *inactive* value, then it does not perform anything, and `(x)` receives no value, defaulting to the *inactive* value. For example, here is an example execution of the affectation interaction:

Instant	(x)	((y)=(x))	(y)
0	4	*active*	4
1	5	*active*	5
2	*inactive*	*active*	*inactive*
3	0	*active*	0
4	1	*inactive*	*inactive*
5	2	*inactive*	*inactive*

4.2.6.5 Function application

The function application interaction is very similar to the affectation interaction, except that it applies a function to its input before sending it to its output.

For example, here is an affectation interaction:

```
((y)=(f)(x))
```

Like the affectation, this interaction also complies with the `Activation in` interface. When it is activated, it takes two inputs: a function `(f)` and a value `(x)`, it applies the function to the value, and sends the result to the output `(y)`. When it is not active, it does not perform any action, and `(y)` defaults to the *inactive* value. Here is an example of application of the function application interaction:

4.2.6.6 Previous

The previous interaction is very similar to the affectation interaction, except that it adds a delay to its input before sending it to its output. It is the only built-in LIDL interaction that allows to access previous values. Here is an expression that uses the previous interaction:

```
((y)=previous(x))
```

Like the affectation and the function application, this interaction also complies with the `Activation in` interface. When it is activated, `(y)` takes the value that `(x)` had previously. When it is not active, it does not perform anything, and `(y)` defaults to the *inactive* value. Here is an example of application of the previous interaction:

Instant	(f)	(x)	((y)=(f)(x))	(y)
0	*cos*	0	*active*	1
1	*cos*	$\frac{\pi}{2}$	*active*	0
2	*cos*	π	*active*	-1
3	*sin*	π	*active*	0
4	*sin*	$\frac{\pi}{2}$	*active*	1
5	*sin*	*inactive*	*active*	*inactive*
6	*sin*	0	*inactive*	*inactive*
7	*inactive*	0	*active*	*inactive*
8	*sin*	0	*active*	0

Instant	(x)	((y)=previous(x))	(y)
0	0	*active*	*inactive*
1	1	*active*	0
2	2	*active*	1
3	3	*active*	2
4	4	*inactive*	*inactive*
5	5	*inactive*	*inactive*
6	6	*active*	3
7	7	*inactive*	*inactive*
8	8	*active*	6

4.2.6.7 Behaviour

We just saw that the three previous interactions, (affectation, function application, previous) comply to the interface `Activation in`: they need an activation signal to determine if they perform an action or not. Such interactions are called behaviours, and are an important part of LIDL programs.

The behaviour interaction allows to add behaviours to any interaction. For example consider the following interaction:

```
(
  (x)
  with behaviour
  ((x) = (4))
)
```

This interaction is equivalent to the interaction `(x)`, except that it continuously runs the behaviour `((x)=(4))`. In this particular case, this interaction is actually equivalent to `(4)`, but the behaviour interaction can be used to build more complex behaviours. Here the result of an execution of the behaviour interaction above:

Instant	(4)	((x)=(4))	(x)	((x) with behaviour ((x) = (4)))
0	4	*active*	4	4
1	4	*active*	4	4
2	4	*active*	4	4

4.3 Conclusion

4.3.1 Overview

In this chapter, we presented LIDL, a language to describe interactive systems formally. The prominent aspect of LIDL is the notion of "interaction". LIDL Interactions denote synchronous data flow operators: at each instant, they receive data through their interfaces, compute internally how these pieces of data interact, and output the result as data through their interfaces.

It is important to note that unlike other languages such as Lustre [123], LIDL does not syntactically segregate input interfaces and output interfaces. Lustre nodes have a declaration of the form `node Example(x:X,y:Y) returns (a:A,b:B)`, where `x` and `y` represent inputs, and `a` and `b` represent outputs. On the other hand, LIDL offers much more flexibility in terms of interfaces. For example, the following LIDL interaction is equivalent to the Lustre node above: `interaction (example (a:A out) (x:X in) (b:B out)):Y in`. This flexibility greatly increases the expressivity of the language, allowing to compose complex interactions with minimal glue code.

LIDL provides a way to declaratively compose complex interactions by assembling simpler interactions. There is a limited number of basic interactions provided by LIDL, and LIDL offers a way to compose them in order to create highly complex interactions that represent the behaviour of complete systems.

Finally, LIDL programs represent a DAG of data flow nodes, which denote the transition function of a state machine. This gives a simple starting point for code generation and application of verification techniques.

4.3.2 Assessment of requirements

Table 4.4 summarises LIDL positioning relatively to the requirements identified in Section 1.3. We note that LIDL provides answers to all of these requirements. However, this assessment is purely theoretical at this point, and LIDL needs to be confronted with real life examples in order to gain more confidence in these answers. This will be done in Chapter 6 (Case study).

Requirement	LIDL solution
1 Formal	Compiles to state machines (Subsection 4.2.5)
2 Collaborative	LIDL is a textual language, so it is compatible with version control systems. Hierarchical definitions allow to structure the division of work
3 Simple	The multi fix syntax allow to write programs which are close to natural language. LIDL is based on a relatively small number of concepts, which all revolve around the concept of interaction.
4 Projection	LIDL program denote nothing more than two functions: an initialisation function, and a transition function, which can be implemented in several programming languages.
5 Traceability	Code generated using LIDL can be traced back to lines in the LIDL program. Interfaces between components are defined clearly using the interface formalism.
6 Composition	LIDL is declarative and works by composing interactions.
7 Modularity	LIDL definitions allow to encapsulate complex interactions in a single interaction
8 Abstraction	LIDL introduces the notions of interaction, interface, and data activation, which are useful abstractions for interactive systems.
9 Manage	The hierarchy of LIDL definitions allow to clarify and classify abstractions that are developed
10 Prototyping	LIDL allows to quickly create high-level UIs by composing lower level components. Behaviour of UIs can be added and refined later in the development cycle
11 Verification	LIDL is formal and compiles to state machines, which can then be used by many verification tools.
12 Flexibility	LIDL is general and not tied to any particular modality. The flexible syntax allows to model various kinds of interactions, from task models to concrete widgets

Table 4.4: Assessment of LIDL solutions to requirements

Chapter 5

LIDL implementation

There is a great satisfaction in building good tools for other people to use
Freeman Dyson

Several tools and libraries have been developed around LIDL. They are all in an early experimental stage of development and the focus is not on performance of the tools or of the generated code, but merely on providing a way to learn LIDL and to use it on as many platforms as possible at a reduced development cost. As a consequence, we made the choice of developing these tools using web technologies. The idea driving this choice is to sacrifice some performance of the tools in order to make them highly portable and easy to develop and maintain.

In this chapter, we will describe four of these products: The LIDL compiler, the LIDL standard library, the LIDL sandbox and the LIDL canvas. These four tools provide have the following roles:

- LIDL compiler: A tool that compiles LIDL code to Javascript code, and offers an API to access and visualise various points of the compilation process. The compiler is written in Javascript ES2015, and can be executed in Command Line mode or in the browser.
- LIDL standard library: A set of LIDL interactions that provide useful interactions which can be used in order to create complex systems. The standard library is written in LIDL and uses some functions written in Javascript.
- LIDL canvas: A web-app component that allows LIDL programs to interact with web browsers, and effectively enables the execution of LIDL GUIs embedded in browsers.
- LIDL sandbox: A web-app that embed the LIDL compiler along with a set of visualisation, interaction and editing tools that allows users of the LIDL language to experiment easily and graphically without having to install anything on their machine.

The next sections are dedicated to the description of these tools along with a description of some interesting challenges faced during their development.

5.1 Compiler

The LIDL compiler takes LIDL code and compiles it to Javascript. More precisely, the LIDL compiler generates two functions: An initialisation function that returns the initial state of the interactive system, and a transition function that takes a state and a set of input values, and returns a state and a set of output values. This complies with the semantics of LIDL, which represent state machines.

5.1.1 Overview

Figure 5.1 presents the process used by the compiler. It follows a classical compiler architecture with a front end, a middle end and a back end. The main characteristic of the LIDL compiler is that it uses a graph as an internal representation along most of the compilation process [124]. The Front end generates a graph which represents the Abstract Syntax Tree (AST) of the LIDL program, which is a hierarchy of definitions. The middle end can be separated in two main phases: Expansion of the definitions, followed by interpretation of basic interactions. Both of these phases are performed as sequences of around 15 graph transformations each. The result of these graph transformations is an ordered DAG of data flow nodes. Each of these nodes contains a code snippet that define their behaviour. The back end is simple: it only has to print all these snippets of code in order.

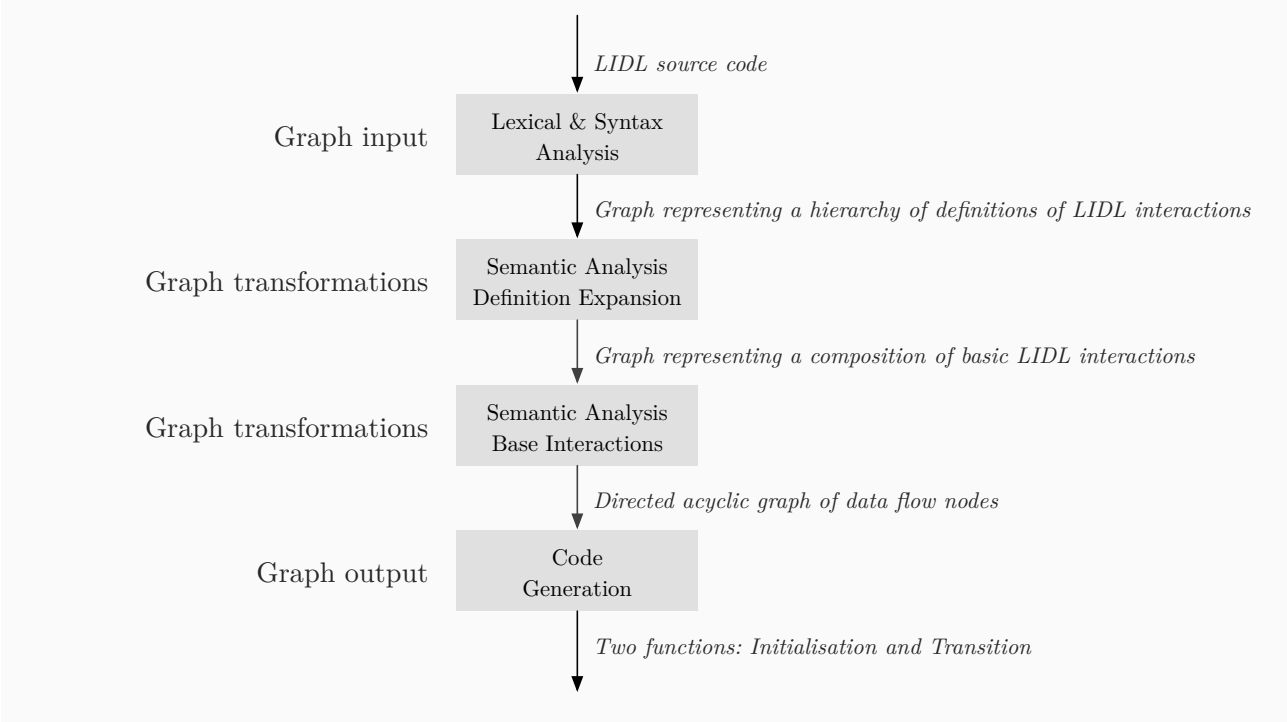


Figure 5.1: LIDL compilation process

Source code for the LIDL compiler represents around 17000 lines of code. The whole compiler was developed using techniques of Test Driven Development. A test suite allows to check non-regression and was developed conjointly with the program. This suite provides around 50% branch coverage of all the LIDL package. Modules such as the parser have branch coverage as low as 22%, while more important modules such as the objects module have 80% branch coverage.

5.1.2 Architecture

Figure 5.2 presents the architecture of the LIDL compiler. The LIDL compiler is implemented in Javascript. The LIDL compiler is a package that complies with the Node Package Manager (NPM) format, which allows it to be reused and distributed using NPM. It depends on three other packages that are available on NPM:

- PEGjs is a parser generator for Parsing Expression Grammars (PEGs), it generates Javascript code that can parse strings. It is used to generate the LIDL parser.

- Lodash is a functional programming library, it provides functional programming primitives for Javascript. Lodash is used extensively in the LIDL compiler, in order to benefit of advantages offered by functional programming.
- Graph is a graph matching and transformation library.

The LIDL compiler is designed in a modular way, with a classical compiler architecture containing a front end, a middle end, and a back end. These components are contained in 5 packages:

- Graph Input: This represents the front-end of the compiler. It contains functions that populate a graph, given input from various sources. The most important element of this package is the function that imports the AST of LIDL programs into the graph.
- Graph Transformations: This represents the middle end of the compiler. It contains functions that perform elementary transformations on the graph. There are around 30 different graph transformations, which are executed sequentially. These transformations give the semantics of LIDL programs. Together, they perform the interpretation of LIDL programs by transforming the AST into a directed acyclic graph of data flow nodes.
- Graph Output: This is the back end of the compiler. It contains functions that extract information from the graph. At the moment, the most important module of this package is the JS code generator, but other modules exist, such as a LIDL code generator or a module that output metrics about the complexity of the LIDL program
- Parser: Contains the LIDL parser, generated using PEGjs from a Parsing Expression Grammar. The parser is used in the graph input package.
- LIDL Objects: Contains utility functions for basic objects of LIDL: data, interfaces, interactions and definitions. For example, this package contains functions that checks if two interfaces are equal, functions that checks if an interaction is a basic interaction... This package is used during graph transformations.

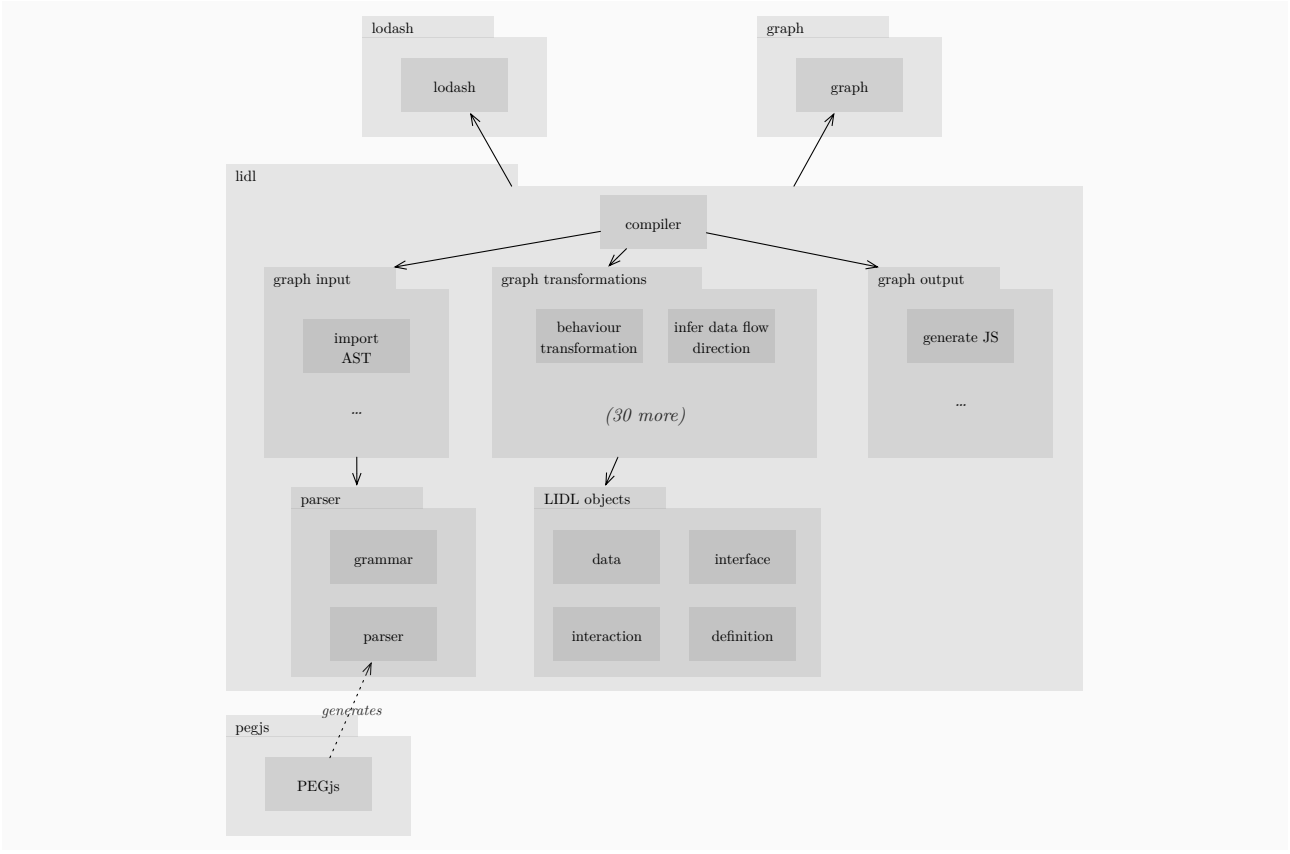


Figure 5.2: Architecture of the LIDL compiler

5.1.3 Example compilation

In this section, we detail the compilation of a simple LIDL program, as performed by the compiler. Since the actual compilation process involves more than 30 steps, we only detail some of them. We follow the four-step process explained in Figure 5.1, with one subsection for each processing step.

5.1.3.1 Lexical and Syntax analysis

This first phase of the compilation process is performed by functions of the *Graph Input* package. The compilation process starts from the source code of our interaction, shown in Listing 5.1. This code represents the definition of an interaction called `(My Simple User Interface)`, which contains an interaction expression (Line 4 - 11), and an interface (Line 2).

```
1  interaction
2    (My Simple User Interface):{theNumber: Number in, theResult: Number out}
3  is
4    (
5      ({
6          theNumber:  (x)
7          theResult:  (y)
8        })
9      with behaviour
10     (apply (function addOne) to (x) and get (y))
11   )
```

Listing 5.1: LIDL code of our example interaction

The parser generates an AST of this program. This AST is used to construct a graph, in a step called *addDefinitionToGraph*. This results in the graph of Figure 5.3. On this graph, the interaction expression is shown in red on the left, the interface is shown in blue on the right and The definition is represented as the root node (light blue). In most use cases, the AST represents a hierarchy of definitions, where each definition contains an interaction expression tree and an interface expression tree, and eventually some children definitions. In our case, there is only one interaction, with no children definitions.

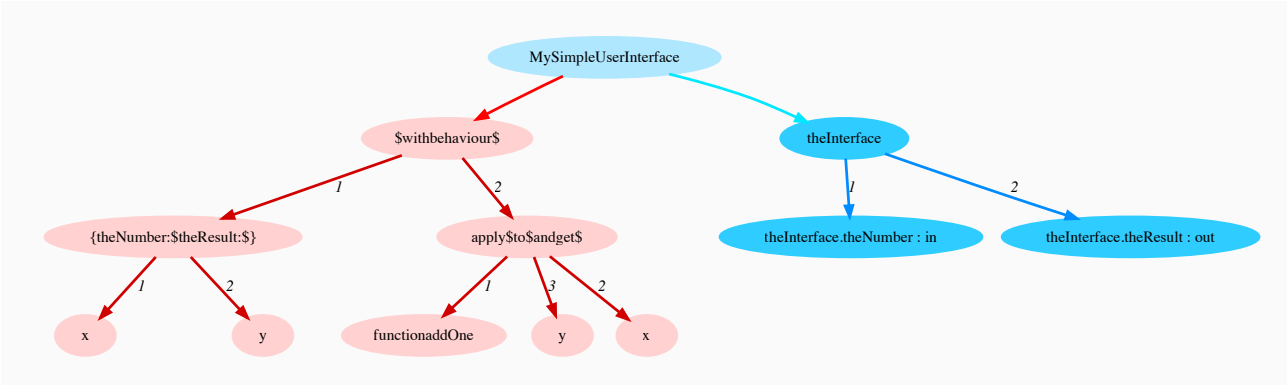


Figure 5.3: Compilation graph after the *addDefinitionToGraph* step

5.1.3.2 Expansion of definitions

This second phase is the first sequence of graph transformations, as seen in Figure 5.1. In this phase, the goal is to go from a hierarchy of interaction definitions to a single expression made of base LIDL interactions. In our example case, this phase is relatively simple because there is only one definition.

One of the transformations in this phase is called *referentialTransparency*, it identifies identical AST nodes. For example, in our example, the interaction `(x)` is used in two different places, line 6 and line 10 of Listing 5.1. The two nodes representing `(x)` are merged into one. This formalises the fact that in LIDL, two identical expressions will always have the same value at a given instant, they actually represent the same node. This step results in the graph shown in Figure 5.4, where duplicate `(x)` and `(y)` nodes have been merged.

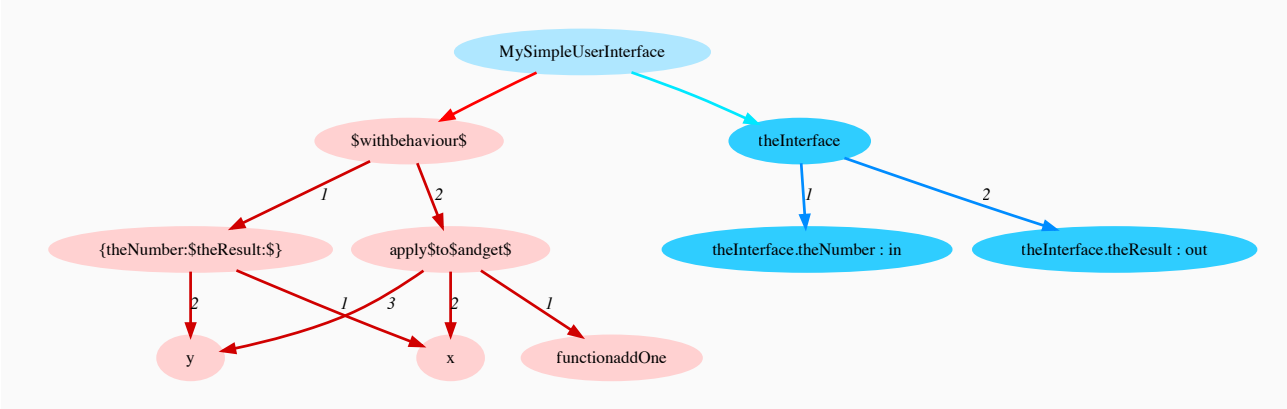


Figure 5.4: Compilation graph after the *referentialTransparency* step

Another transformation of this sequence is called *expandDefinitions*, it starts from the interaction expression of the root definition, and recursively replaces non-base interactions of this expression by their definition, until only base interaction remain. In our case, there is only one definition, and it already only contains base interactions, so this step does not have large effects on our example. The result is shown in Figure 5.5. The only difference with the previous graph is that interactions (red nodes) have been marked as expanded (yellow nodes). In actual use cases, this step has a much bigger impact on the structure of the graph. In any cases, after this step, only the main definition remains.

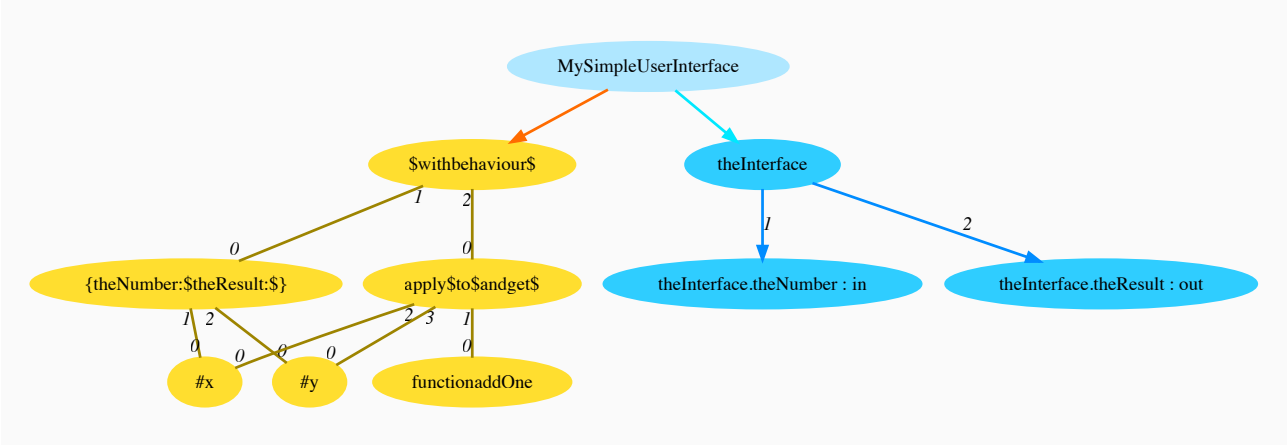


Figure 5.5: Compilation graph after the *expandDefinitions* phase

Finally, a step called *instantiateInterfaces* is performed. The goal of this step is to replace the interface of the main definition with corresponding interactions. For example, in our case, the interface of the

main interaction is the following composite interface: `{theNumber: Number in, theResult: Number out}`. This is represented as the three blue nodes in Figure 5.5: One node for the atomic interface `Number in`, one node for the atomic interface `Number out`, and one node for the composite interface `{theNumber: Number in, theResult: Number out}`. Atomic interfaces are replaced with snippets of code that access and write to the appropriate variable in the target programming language, while composite interfaces are replaced with composition interactions. The interaction associated with the main interface is then linked to the interaction at the root of the expression tree. This results in Figure 5.6.

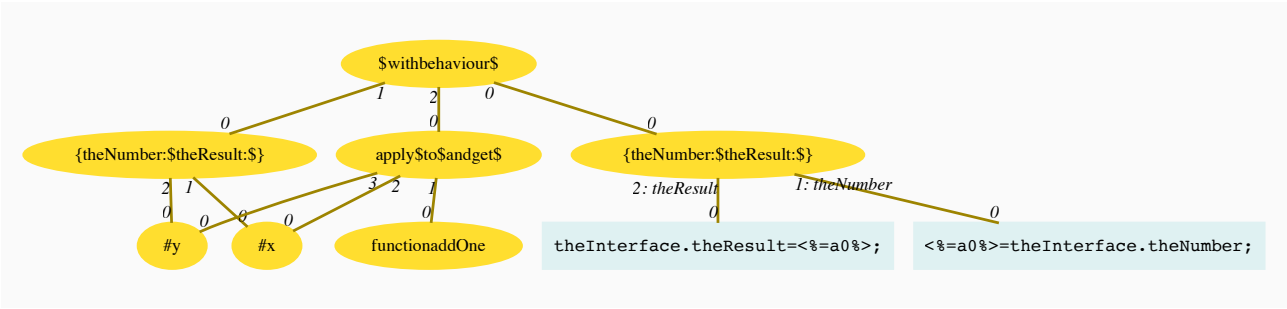


Figure 5.6: Compilation graph after the *instantiateInterfaces* phase

5.1.3.3 Interpretation of base interactions

At this point, the graph is only made of base interactions (yellow nodes) and some code snippets (light cyan nodes). The goal of the following phase is to interpret the semantics of base interactions by progressively eliminating all the base interaction nodes and replacing them with appropriate code snippets nodes and edges. Each category of base interaction nodes is associated with a graph transformation which defines their semantics.

For example, a step called *linkIdentifiers* aims at eliminating identifier interactions, such as `(x)` and `(y)` in our case. The idea of this step is to remove the identifier nodes by linking their neighbours together. For example, in our case, `(x)` is used in two different places of the code. As a consequence the node representing `(x)` is linked to two interaction nodes. The transformation removes the node representing `(x)` and directly link these two interaction nodes together. This results in the graph shown in Figure 5.7. This graph does not contain identifier nodes anymore, they have been replaced by edges.

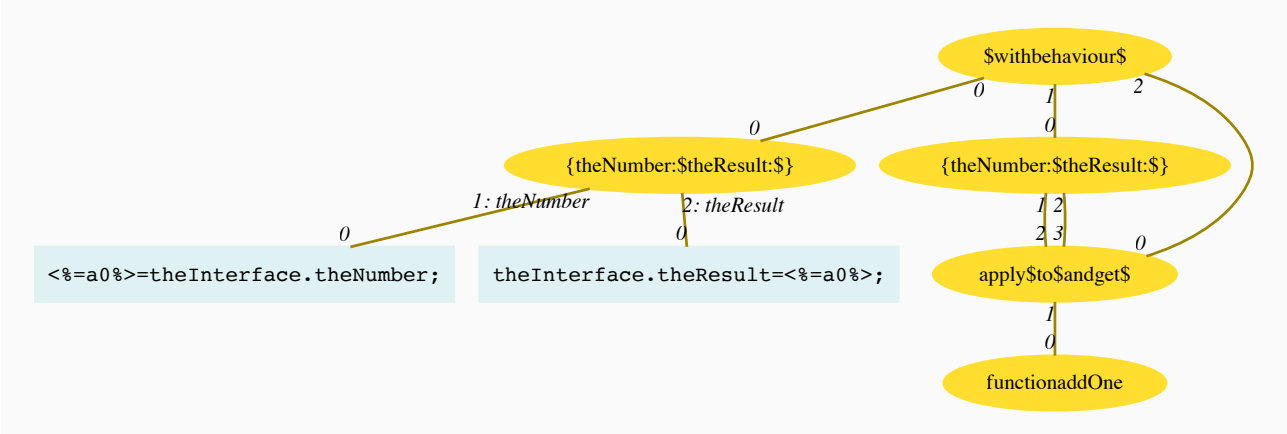


Figure 5.7: Compilation graph after the *linkIdentifiers* phase

A second step is called *behaviourSeparation*, it aims at eliminating the `(()with behaviour())` interactions. This graph transformation defines the semantics of these interactions: it links their second argument to a node that constantly outputs the `active` value, and it links their first argument to their parent. In our case, there is one `(()with behaviour())` interaction. The corresponding node is removed and the appropriate links are created, resulting in the graph shown in Figure 5.8.

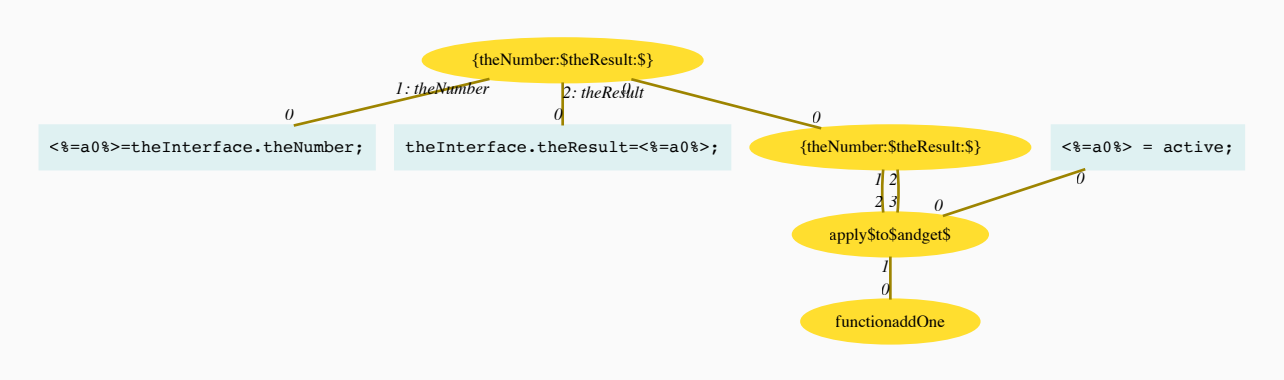


Figure 5.8: Compilation graph after the *behaviourSeparation* phase

Another step is called *functionApplicationLinking*. It interprets the function application interactions, by replacing them with appropriate code snippets. This results in the graph shown in Figure 5.9, which does not contain any function application interaction anymore.

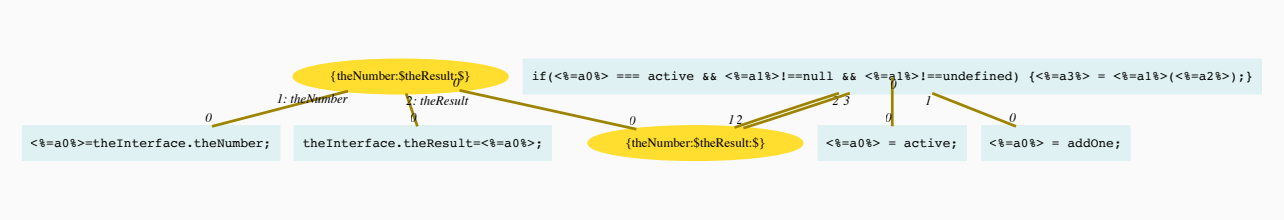


Figure 5.9: Compilation graph after the *functionApplicationLinking* phase

At this point, our graph only contains composition interactions, and the goal of the following step is to eliminate them. When trying to eliminate composition interactions, two possibilities can happen: Either a composition node is paired with a matching composition node, or not. These two situations correspond to two different graph transformations. In our case, we see that the two composition nodes are matching: they are directly linked together by an edge. They will be eliminated simultaneously, resulting in the graph shown in Figure 5.10.

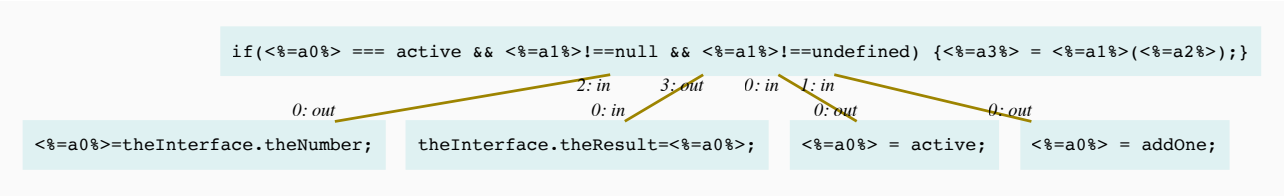


Figure 5.10: Compilation graph after the *affectationLinking* phase

Now, the graph does not contain any interaction nodes anymore. It is only composed of code snippets which describe data flow operations. The only transformation left to do is to find a correct execution order for these data flow operations. This is obtained by performing a topological ordering using Tarjan's algorithm [125], resulting in the graph shown in Figure 5.11. This ordering step would be optional if we generated code for data flow programming languages such as Lustre [123], where the various processing blocks do not have to be written in their execution order.

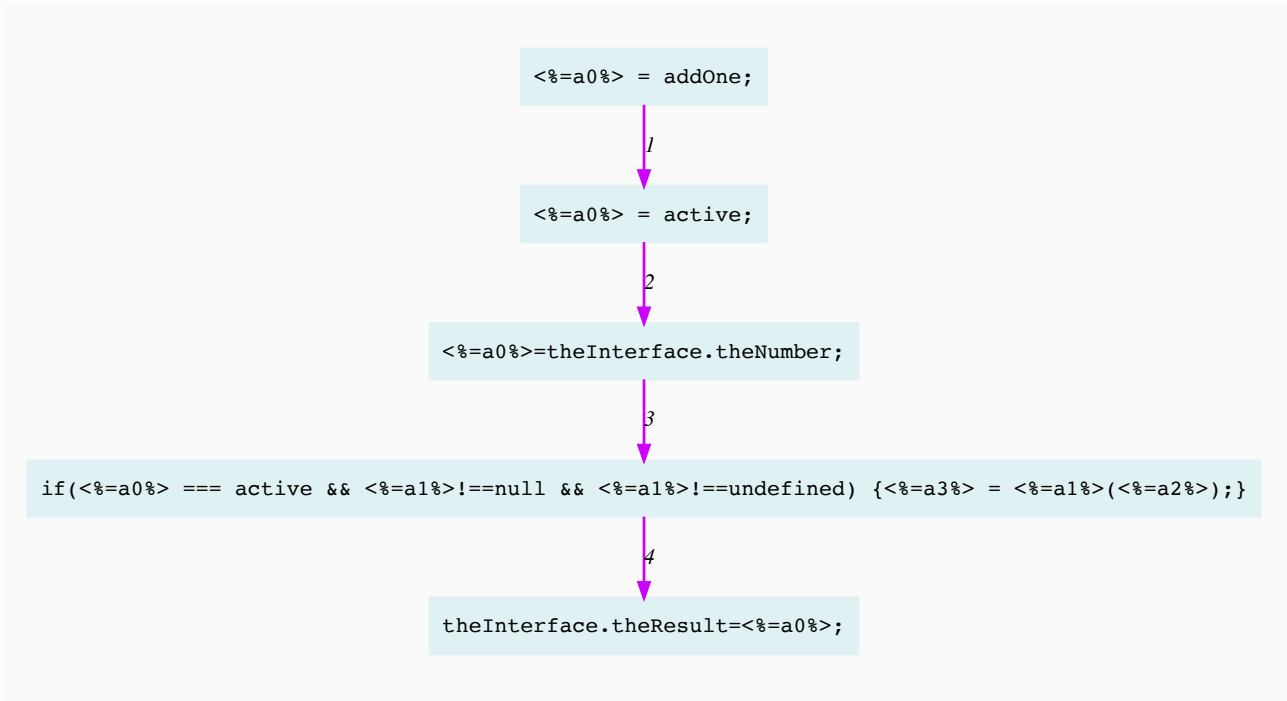


Figure 5.11: Compilation graph after the *orderGraph* phase

5.1.3.4 Code generation

From the situation depicted in Figure 5.11, the code generation phase is relatively trivial, it simply consists in concatenating the code snippets, and putting them inside a code template. Each edge of the graph represents a variable, and each node of the graph represent an operation on the variables it is linked to. This results in the code shown in Listing 5.2. The graph was used to generate the code at Lines 17-37.

The graph of Figure 5.11 is made of 5 nodes and 4 edges, we find these elements in the generated code. Each edge of the graph is associated with the declaration of a variable in the generated code (Lines 17-22). Each node of the graph is associated with a block of code which works with variables that are associated with the edges connected to the node (Lines 24-37).

This JS code can now be executed an perform the interactions described in the LIDL program.

5.1.4 Limitations and perspectives

5.1.4.1 Performance

As explained in the introduction of Chapter 5, the main objective of this compiler implementation is to be a proof of concept, offer a reference implementation, and allow easy development and future evolutions. As a consequence, the compiler is highly perfectible in terms of performance of the generated code, and in terms of performance of the compiler itself. On the compiler side, the following optimisation opportunities have been identified:

- Optimisation of the graph library: The graph transformation engine used in the compiler is not optimised. Using a more optimised graph library could speed up the compilation.

```
function transitionFunction(data) {
    ////////////////////////////////////////////////////////////////////
    // Standard LIDL Header (Standard JS function definitions)
    var theInterface = clone(data.inter);
    var inactive     = null;
    var active       = "lidl_active_value";
    ...
    ////////////////////////////////////////////////////////////////////
    // Custom LIDL Header (Custom JS function definitions)
    var addOne = function(x){
      if(isActive(x)){ return (x+1); } else { return inactive; }
    };
    ...
    ////////////////////////////////////////////////////////////////////
    // Declaration of variables (Edges of the graph)
    var edge_133 = inactive;
    var edge_134 = inactive;
    var edge_145 = inactive;
    var edge_146 = inactive;
    ////////////////////////////////////////////////////////////////////
    // Data flow processing (Nodes of the graph)
    // node`130
    edge_134 = addOne;
    // node`127
    edge_133 = active;
    // node`95
    edge_145 = theInterface.theNumber;
    // node`132
    if (edge_133 === active && edge_134 !== null && edge_134 !== undefined) {
        edge_146 = edge_134(edge_145);
    }
    // node`97
    theInterface.theResult = edge_146;
    ////////////////////////////////////////////////////////////////////
    // Return statement
    return {
        state: nextState,
        args: theArgs,
        inter: theInterface
    };
}

function initializationFunction(data) {
    return {
        state: {},
        args: {},
        inter: {}
    };
}
```

Listing 5.2: Code generated from our example interaction

- Optimisation of graph transformations: The graph matching and transformation operations are not optimised. In most cases, naive algorithms have been used for the sake of understandability of the code. Using more optimal matching and transformation algorithms could speed up the compilation.
- Use of a more performant language. For portability reasons, the whole compiler is implemented using JS. While JS execution engines have improved a lot in the recent years, allowing execution of JS code with speeds of the same order of magnitude as native code. This still leaves room for improvement. Using a more performant language such as C or OCaml could potentially divide the compilation time by two or more.

The following opportunities have been identified to optimise the generated code.

- The compiler acts by naively replacing each edge of the graph with a variable, and each node with a piece of code. This result in a large number of variables, when many optimisations would be possible. For example, in Listing 5.2, one could replace 20 Lines of code (Lines 14-33) with a single line of code: `theInterface.theResult = addOne(theInterface.theNumber);`. This is a very drastic optimisation because this is a simple use case, but in most actual use case, simple optimisations could reduce the size of the code by a factor of two or more. However, these optimisations might not always be useful: in some cases, the compiler or interpreter for the target programming language could perform these optimisations itself.
- The generated code recomputes all the values of the whole DAG on each step, even when the input does not change. It should be relatively simple to add optimisation steps after the graph transformations in order to simplify the transition function, so it does only compute values when they change, in a process similar to memoization. Some infrastructure to support memoization is already present in the LIDL compiler.

5.1.4.2 Language coverage

The compiler does not yet support all features of the LIDL language. The modular architecture of the compiler allows to incrementally add features to the language, by adding the appropriate graph transformation steps in the compilation pipeline. The current limitations includes:

- No support for polymorphism of custom interactions. LIDL allows to define interactions which have the same operator, but different interfaces. For example, the built-in interaction `((a)=(b))` is polymorphic, it is defined whatever the interfaces of `(a)` and `(b)` are, as long as they are compatible. At the moment, it is impossible for a LIDL user to define such polymorphic interactions.
- No support for imports. LIDL programs must be in a single file and cannot use the `import` statement to import definitions of other files.
- No support for interface and data definitions. All interfaces and data types must be used in their canonical form, and cannot be references to interfaces or data types defined elsewhere in the code. Interface and Data types definitions are not taken into account.

5.2 Standard library

The LIDL standard library provides a set of predefined interactions that are built using the basic interactions that exist in LIDL. These interactions fall in different categories:

- Data operators: Interactions that perform the classical operations that are standard to most programming languages, such as numeric operations, string operations...
- Behaviours: Interactions that allow to represent the behaviour of interactive systems.
- State related interactions: Interactions that are related to temporal aspects of systems behaviour.

- Abstract widgets: A library of widgets that support the creation of abstract UIs.
- Concrete widgets: A library of widgets that support the creation of concrete WIMP UIs
- Task modelling: A library of interaction that can be used to represent human task models.

In this section, we present an overview of the LIDL standard library, by presenting some examples and interactions of each of these categories in order.

5.2.1 Data operators

Data operator interactions are stateless interactions (they do not have any internal state, their output at a given instant only depend on their input at the same instant), they only perform synchronous computations on base data types, and they have simple interfaces. Most of these operators are nothing more than wrappers around underlying language operators. During compilation of LIDL programs, most of these interactions compile back to the original data operators of the underlying language. Data operator interactions defined in the LIDL standard library are the basic operators that works on the following data types: `Activation`, `Boolean`, `Number` and `Text`.

Example The following interaction is a composition of various data operators interactions:

```
1 ( ( (speed) > (maximum speed) ) or ( ((width) * (height)) > (maximum area) ) )
```

This interaction takes several input values of type `Number` (`(speed)`, `(maximum speed)`, `(width)`, `(height)`, `(maximum area)`), and combines them in a result of type `Boolean`. The meaning of each interaction in this expression should be clear to any programmer.

Definitions Listing 5.3 presents several definitions of data operators as defined in the standard library. Lines 1-4 is the definition of the products of numbers, Lines 6-9 is the definition of the boolean disjunction, and Lines 11-14 present the definition of the comparison of two numbers. These interactions define infix operators, by using functions that implement the appropriate operators.

```
1  interaction
2    ((a:Number in) * (b:Number in)):Number out
3  is
4    ( (function numericProduct) ((a),(b)))
5
6  interaction
7    ((a:Boolean in) or (b:Boolean in)):Boolean out
8  is
9    ( (function booleanOr) ((a),(b)))
10
11 interaction
12   ((a:Number in) > (b:Number in)):Boolean out
13 is
14   ( (function numericGreater) ((a),(b)))
```

Listing 5.3: Definitions of several data operators in the LIDL standard library

5.2.2 Behaviours

Behaviours are interactions that comply with the interface `Activation in`. They represent interactions that can be either *active* (performing some action) or *inactive* (not doing anything). Behaviours are interactions that depends on an activation input, thus they comply with the interface `Activation in`.

Behaviours are part of the base LIDL language: LIDL provides a base interaction which allows to add behaviours to any interaction: `((a)with behaviour(b))`. Furthermore, LIDL has three built-in behaviours: the affectation `((a)=(b))`, function application `((f)(x)=(y))` and previous`((x)=previous(y))`. They represent equations that are effective only when they receive the value *active*.

The LIDL standard library provides several behaviour composition operators, which allow to create complex behaviours by composing simpler behaviours. This section describes some of these behaviour composition interactions defined in the LIDL standard library.

Example Listing 5.4 presents an example expression made by composing behaviour interactions defined in the standard LIDL library:

```
1  ( all
2    ( when (obstacle ahead)
3      then ( all
4        (alert)
5        (brake)
6      )
7      else ( either
8        (accelerate)
9        (coast)
10     )
11   )
12 )
```

Listing 5.4: An example of composition of behaviours

This interaction describes a behaviour created by using behaviour composition operators (`all`, `either`, `when then`) to compose four behaviours (`alert`, `brake`, `accelerate` and `coast`) into a more complex behaviour.

The `(all(a)(b))` interaction runs several behaviours in parallel. When `(all(a)(b))` is activated, then the two behaviours `(a)` and `(b)` are activated. The `(either(a)(b))` represents a non-deterministic choice between several behaviours. When `(either(a)(b))` is active, one and only one of the several sub behaviours `(a)` or `(b)` is activated. The `(when(cond)then(a)else(b)` behaviour represents a deterministic choice. When `(when(cond)then(a)else(b)` is active, `(a)` or `(b)` is activated depending on the value of the `Boolean (cond)`.

Definitions Listing 5.5 presents several definitions that are part of the behaviour section of the LIDL standard library. First, Lines 1-4 defines the `Behaviour` interface. Lines 3-10 define the `all` interaction, which is a composition operator for behaviours: when the `all` behaviour is active, then both of its sub behaviours `(a)` and `(b)` are activated. Lines 12-19 define the `either` interaction, which is also a composition operator for behaviours: when the `either` behaviour is active, only one of its sub behaviour is activated, and the choice is non-deterministic. Finally, Lines 21-28 define

the deterministic composition operator, where the the activation of the sub behaviours `(a)` or `(b)` is chosed deterministically depending on a boolean condition.

```
interface Behaviour is Activation in

interaction
  (all (a:Behaviour') (b:Behaviour')):Behaviour
is
  (
    (x)
    with behaviour
    ( (function behaviourAll) (x) = ((a),(b)) )
  )

interaction
  (either (a:Behaviour') (b:Behaviour')):Behaviour
is
  (
    (x)
    with behaviour
    ( (function behaviourEither) (x) = ((a),(b)) )
  )

interaction
  (when (cond:Boolean in) then (a:Behaviour') else (b:Behaviour')):Behaviour
is
  (
    (x)
    with behaviour
    ( (function behaviourWhen) ((x),(cond)) = ((a),(b)) )
  )
```

Listing 5.5: Definitions of several behaviour interactions in the LIDL standard library

5.2.3 State related interactions

As explained in Chapter 4, LIDL only provide one base interaction that allows to represent the notion of state. This interaction is `((a)=previous(b))`. It offers a way to get the value of an interaction at a previous point in time. However, it is sometimes simpler to use interactions defined in the standard library, which provides shortcuts for commonly used stateful interaction patterns. Some of those interactions can be seen as some sort of temporal operators, and actually share similarities with Linear Temporal Logic (LTL) operators.

Example The following interaction is a composition of state related interactions:

```
( from (mouse down) until ( previous (mouse up) ) )
```

The `(from (a) until (b))` interaction continuously returns an active value from the instant when `(a)` is active, until the instant when `(b)` is active. The `(previous (a))` interaction returns the value

of (a) at the previous instant. Table 5.1 shows a timing diagram of an execution of this interaction, showing the temporal behaviour that can be described using the expression above.

Step	(mouse down)	(mouse up)	(previous (mouse up))	(from (mouse down) until (previous (mouse up)))
1	.	.	.	.
2	*active*	.	.	*active*
3	.	.	.	*active*
4	.	*active*	.	*active*
5	.	.	*active*	.
6	.	.	.	.
7	*active*	.	.	*active*
8	*active*	*active*	.	*active*
9	.	.	*active*	.

Table 5.1: Timing diagram of an execution of nested state interaction

Definition Listing 5.6 presents are some definitions of state related interactions as defined in the LIDL standard library. The `previous` interaction (Lines 1-4) is equivalent to the `pre` operator of Lustre: it outputs the value that an interaction `(a)` had at the previous instant. The `next` interaction (Lines 6-9) is its symmetric: it receives a value, and sends it to an interaction `(a)` receives it at the

next instant. The past interaction is the equivalent of the current operator of Lustre: it outputs the last active value that was output by a given interaction (a), even if (a) has been inactive for a long time since. The future interaction is its symmetric: it gives a value to a variable (a), and ensures that (a) stays active in the future. Finally, the from()until() is a more complex interaction, which is similar to the past interaction, except that it has a "switch" call (b) which makes the output value become inactive again.

```
interaction
  (previous (a:t in)) : t out
is
  ( (x) with behaviour ( (x) = previous (a) ) )

interaction
  (next (a:t out)) : t in
is
  ( (x) with behaviour ( (a) = previous (x) ) )

interaction
  (past (a:t in)) : t out
is
  ( (x) with behaviour ( (x) = ((a) fallback to (previous (x))) ) )

interaction
  (future (a:t out)) : t in
is
  ( (x) with behaviour ( (next(a)) = ((x) fallback to (a)) )  )

interaction
  ( from (a:t in) until (b: Activation in)) : t out
is
  ( (x) with behaviour
    ( when ( (b) is inactive )
      then ( (x) = ((a) fallback to (previous (x))) )
    )
  )
```

Listing 5.6: Definitions of several state related interactions in the LIDL standard library

5.2.4 Abstract widget library

The abstract widget library provides a set of abstract representation of widgets, supporting the creation of abstract user interfaces by composing widgets. Composition of widgets is done using data composition interactions. This allow to create abstract interfaces, which represent the high level behaviour of actual UI, without the particular implementation details.

Example Listing 5.7 presents an example abstract UI defined using a composition interaction to group several abstract widgets. It describes the abstract UI of a counter. The (button (a) displaying (b) triggering (c)) interaction is an abstract widget representing a button which is active when (a) is active, which displays the text (b) to the user, and which triggers the behaviour

(c) when clicked by the user. The (label (a) displaying (b)) interaction is an abstract widget representing a label which is active when (a) is active and which displays the text (b) to the user.

```
({
  increment : (button (active) displaying ("+") triggering (increment (value)) )
  decrement : (button (active) displaying ("-") triggering (decrement (value)) )
  display   : (label (active) displaying ((value) as text) )
})
```

Listing 5.7: An example abstract UI using standard widgets

Definitions Listing 5.8 presents the definition of the abstract button widget of the LIDL standard library. Lines 1-7 define the abstract interface of the button. At this level of abstraction, a button allows to display a text to the user (Line 5), and allows the user to trigger things (Line 6). Lines 9-18 define the interaction performed by an abstract button. The button has a parameter call (a) which allows to enable or disable the button. The button has another parameter called (b) which represents the text to be displayed on the button. The button can also trigger behaviours through the argument called (c). The actual interaction of the button is straightforward (Lines 12-19). The main thing to notice is that when the button is inactive, then no text is displayed to the user, and user clicks are not transmitted to (c).

```
1  interface
2    Button
3  is
4    {
5      label: Text out,
6      click: Activation in
7    }
8
9  interaction
10   (button (a:Activation in) displaying (b:Text in) triggering (c:Activation out)
     ↪ ):Button
11 is
12   ( if ((a) is active)
13     then ({
14       label: (b)
15       click: (c)
16     })
17     else ({})
18   )
```

Listing 5.8: Definition of the abstract button widget

5.2.5 Concrete widget library

The abstract widget library provides a set of concrete widgets, supporting the creation of user interfaces by composing widgets. While the abstract widget library only provides widgets, and relies on data composition interactions to compose complex UIs, the concrete widget library also provides widget composition interactions, or containers, because there can be several ways of composing concrete widgets (juxtaposition, superposition...).

Example Listing 5.9 presents an example concrete UI defined using a composition interaction to group several abstract widgets. This interaction is similar to the interaction presented in Subsection 5.2.4. There are two main differences:

- The three widgets are instances of concrete widgets, which are more complex than abstract widgets, because they include information about presentation, and they include interactions related to mouse pointing and other details which are abstracted away from the abstract widgets.
- The composition interaction that glues the three widgets together is not a data composition interaction as in abstract UIs. There are several ways to compose concrete widgets, and in this case we chose to compose the widgets by juxtaposing them in a row layout. We could also have chosen to put the three widgets in three different tabs, or other options...

```
1 ( row layout with (panelStyle) containing
2   (button (active) displaying ("+") triggering (increment (value)) with
  ↪  (buttonStyle) )
3   (button (active) displaying ("-") triggering (decrement (value)) with
  ↪  (buttonStyle) )
4   (label  (active) displaying ((value) as text) with (labelStyle) )
5 )
```

Listing 5.9: An example concrete UI specified using standard LIDL components

Definitions The concrete widget library defines several interfaces and interactions related to concrete WIMP UIs. The most important of the interfaces is the unique interface that all concrete widget comply with. This interface is formalised and described using LIDL. Listing 5.10 shows the definition of this interface, called `WimpUi`, along with some other sub interfaces.

The `WimpUi` interface is composed of five parts, which define everything that is needed to create composable WIMP UIs:

- `layout` (line 3): The layout information is an input of the UI. It contains information about the position and size of the window (or screen) the UI must be displayed on. This information can evolve dynamically during the execution of the UI.
- `time` (line 4): The current time is an input. It can be used to generate animations and transitions.
- `mouse` (line 5): The mouse input from the user. Depending on the mouse position, active mouse button and their own position, UI components can detect clicks, drag and drops and other mouse interactions.
- `keyboard` (line 6): The keyboard input from the user. It is a record that contains the state of each key of the keyboard, and can be used for any keyboard interaction, typically to fill text boxes.
- `layout` (line 7): The graphics output to the user. It is a vector graphics description of the UI. The LIDL canvas is in charge of rendering this vector graphics into raster graphics that can be painted on the HTML canvas element.

The standard concrete widget library defines several interactions, that belong to three categories::

- Utility interactions that allow to work on objects related to WIMP UI. For example, this includes interactions to detect if a mouse click happened inside a given rectangle, to detect double clicks, interaction generating basic graphic primitives, and so on.
- Composition interactions, also known as containers, that allow to group several widgets or containers inside a unique object.
- Basic concrete widgets, such as buttons, labels, sliders, progress bars, drop down menus...

```
interface WimpUi is
  {
    layout    :  Layout in,
    time      :  Number in,
    mouse     :  Mouse in,
    keyboard  :  Keyboard in,
    graphics  :  Graphics out
  }

data Layout is
  {
    position  :  Point2D,
    size      :  Point2D,
    screen    :  Point2D
  }

data Mouse is
  {
    position  :  Point2D,
    wheel     :  Point2D,
    button1   :  Activation,
    button2   :  Activation,
    button3   :  Activation,
    button4   :  Activation
  }

data Keyboard is
  {
    key0      :  Activation,
    key1      :  Activation,
    ...           ...
    keyA      :  Activation,
    keyB      :  Activation,
    ...           ...
    shift     :  Activation,
    cmd       :  Activation,
    ctrl      :  Activation
  }

data Graphics is Native

data Point2D is
  {
    x         : Number,
    y         : Number
  }
```

Listing 5.10: The interfaces used in definitions of WIMP interfaces

Listing 5.11 presents the definition of a container which allows to compose two widgets by juxtaposing them in a row layout. This definition is part of the standard library.

```
1  interaction
2    (row layout with (style:Style in) containing (a:WimpUI') (b:WimpUI')):WimpUI
3  is
4    ({
5      mouse:    ( all
6        ((a).mouse)
7        ((b).mouse)
8      )
9      layout:   ( split at (50)% between
10       ((a).layout) on the left
11       ((b).layout) on the right
12     )
13     graphics: ( group
14       ((a).graphics)
15       ((b).graphics)
16     )
17   })
```

Listing 5.11

We see that the behaviour of this container is formalised clearly:

- Lines 5-8: Both widgets receive the same mouse signal, they can both interact with the mouse
- Lines 9-12: The widgets have different layouts, `(a)` is on the left and `(b)` on the right.
- Lines 13-16: The widgets graphics are grouped into a single graphics signal, they are both visible at the same time.

Other concrete widgets are part of the standard LIDL library. For example, Subsection 6.4.1 presents the definition of a concrete Button component.

5.2.6 Task modelling

The standard library contains a set of interactions that support the creation of task models.

Example Listing 5.12 presents an example task model described using elements of the standard library. This task model is similar to the task model expressed using CTT on Figure 2.23 (Subsection 2.10.1). It uses two task composition interactions that are part of the standard task model library: `(sequentially (a) (b))` and `(either (a) (b))`.

Definitions The standard library contains the definition of the `Task` interface, which represents composable tasks. Listing 5.13 presents the `Task` interface. This interface contains several signals that, when combined, support the creation of complex task models:

- `command` is a channel that allows a set of events (`start`, `cancel`, `pause` and `resume`) to be sent by the parent task to change the state of the task.
- `status` allows parent task to know the current status of the task (`running`, `suspended`, `success`, `error`, `idle`...)

```
( sequentially
  ( either
    ( select single room as (chosen room type))
    ( select double room as (chosen room type))
  )
  ( sequentially
    ( ask for availability of rooms of (chosen room type) as (available rooms) )
    ( select one of (available rooms) as (chosen room) )
  )
)
```

Listing 5.12: An example task model modelled using elements of the standard library

- `time` represents the current time, which is useful to create task that have specific temporal properties (duration, delay, timeouts...)
- `progress` allow the task to report back on its progress and the remaining work required to finish it.

```
interface
  Task
is
  {
    command  : TaskCommand in,
    status   : TaskStatus out,
    time     : Number in,
    progress : Progress out
  }
```

Listing 5.13: The task interface, part of the standard library

The standard library defines several interactions that allow to compose tasks to create more complex patterns. For example, Listing 5.14 presents the definition of the `sequentially` task, which allows to combine two task by executing them sequentially.

```
interaction
  ( sequentially (a:Task') (b:Task') ): Task
is
  (
    ({
      command  : ( (current task).command )
      status   : ( (current task).status  )
      time     : ( all  ((a).time)     ((b).time)     )
      progress : ( mean ((a).progress) ((b).progress) )
    })
    with behaviour ( all
     ( when ((a) completes) then (start (b)) )
     ( when ((a) completes) then ((current task) = (b)) )
     ( when ((b) completes) then ((current task) = (a)) )
    )
  )
```

Listing 5.14: Definition of the sequential task composition operator

5.3 Canvas

LIDL Canvas is a JS framework which allows to execute LIDL WIMP applications in web browsers. It acts as an adapter between the event-based execution of UIs in the browser and synchronous flow execution of LIDL programs. The main idea that drove the development of LIDL canvas was to provide a reference implementation of the way to use LIDL programs in event-based environments. In this section, we describe the architecture model the LIDL canvas fits in, then we describe how it is interfaced with LIDL programs. Then we focus on two important details of this implementation: its execution model (how it translates event sequences into data flows and back), and the representation of graphics. Finally we describe the limitation and perspectives of this work.

5.3.1 Architecture

Figure 5.12 presents a geomorphic view (See Chapter 3) of the architecture of interactive systems using LIDL canvas. The architecture contains several layers:

- Display, Mouse and Keyboard: The low level layers that represent the actual interaction between the user and the various modalities of WIMP interfaces.
- Browser: The low level layer which represents the inner workings of the browser and how it interact with the user by combining the lower layers. There might be other layers under this one, they are hidden here for the sake of simplicity.
- HTML API: The standardised HTML API [126] implemented by all recent web browsers such as Chrome, Safari or Firefox. This API presents high level input events such as mouse clicks, mouse movements and key strokes. The API also provides way to output data to the user, either in the form of manipulations of the DOM, or in the form of drawing commands on a HTML Canvas element.
- LIDL WIMP UI: A high level layer which is adapted to represent UI using LIDL. In this layer, inputs are represented as LIDL data flows (mouse position, key strokes...), and output is represented as a Scene model, a composition of geometric primitives which are displayed to the user.

The various interactors involved in the "machine" side of this architecture are:

- LIDL UI: The red element in Figure 5.12, this interactor is where the developer express UI interactions using LIDL. The interaction performed by this interaction has to implement a given Interface in order to be compatible with LIDL Canvas.
- LIDL Canvas: The purple element in Figure 5.12, this is the adapter between the LIDL UI and the HTML API. It is generic, this is the interactor which is described in this section.
- Browser: The blue element in Figure 5.12, This represents the web browser as an interactor. The interactions performed by this interactor are complex and out of the scope of this section, but what is important about it is that the browser eventually forwards user inputs and outputs to the LIDL Canvas via the HTML API.
- Video, Mouse Driver, Keyboard driver are interactors that represent low level interactions between the user and the machine. Their description is out of the scope of this section, but it is nevertheless useful to know that they are the only low-level interactors involved in the operation of LIDL canvas interface. In particular, we see that there is no Speaker, Microphone or Joystick used in the current version of LIDL canvas.

As its name implies, LIDL Canvas is not tied to any particular look and feel or graphical representation. The only role of the canvas is to paint and display a graphical representation to the user, while also forwarding user events to the LIDL system. This means

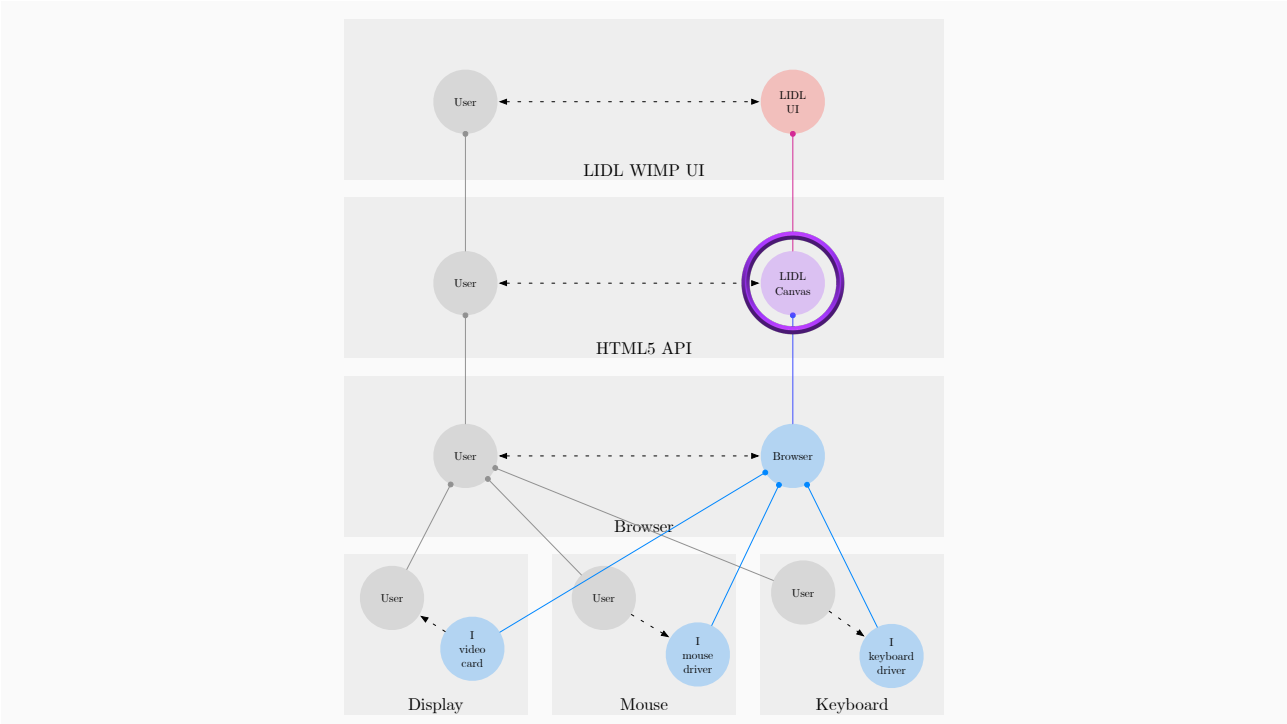


Figure 5.12: Geomorphic view of the LIDL Canvas environment

5.3.2 Interface

As shown in Figure 5.12, the LIDL canvas (Purple) is interfaced with UIs developed by third-party developers using LIDL (Red). Any LIDL interaction that complies with the `WimpUi` interface of the standard library (see Subsection 5.2.5) can be used with LIDL canvas, and be executed as a WIMP UI. For example, here is a simple LIDL UI that is compatible with the LIDL canvas:

```
interaction
  (my user interface):CanvasWimpUi
is
  ({
    mouse     :  ({
      button1: (increment (counter))
      button2: (decrement (counter))
    })
    graphics  :  ( graphic text
      displaying ((counter) as text)
      centered at ((0),(0))
    )
  })
```

This UI will simply paint a text displaying the value of a counter. A left click will increment the counter while a right click will decrement the counter. The

5.3.3 Execution model

Execution of the LIDL canvas is based on three different methods:

- Init: this function simply register new event listeners for user events (mouse move, key presses...), and calls the initialisation function of the LIDL system.
- Analyse output: Takes the output of the LIDL program, decomposes it and paints the graphics on the HTML canvas.
- Compose input: Receives user events from the browser, and use them to create the values of the input interface of the LIDL system. Then, call the transition function of the LIDL system.

As shown in Figure 5.13, LIDL canvas execution follows two different phases: Initialisation and Transition.

- Initialisation is executed once, as soon as the program is launched. During this phase, the Init method is called once, and the Analyse output method is called once to perform the initial painting of the UI.
- Transition is executed each time an event is received. This phase is triggered when a listener receives a user event. The user event is used to compose the value of the input interface of the LIDL system. The transition function is then executed, and the resulting output is analysed and used to call output functions such as graphics painting or sound playing.

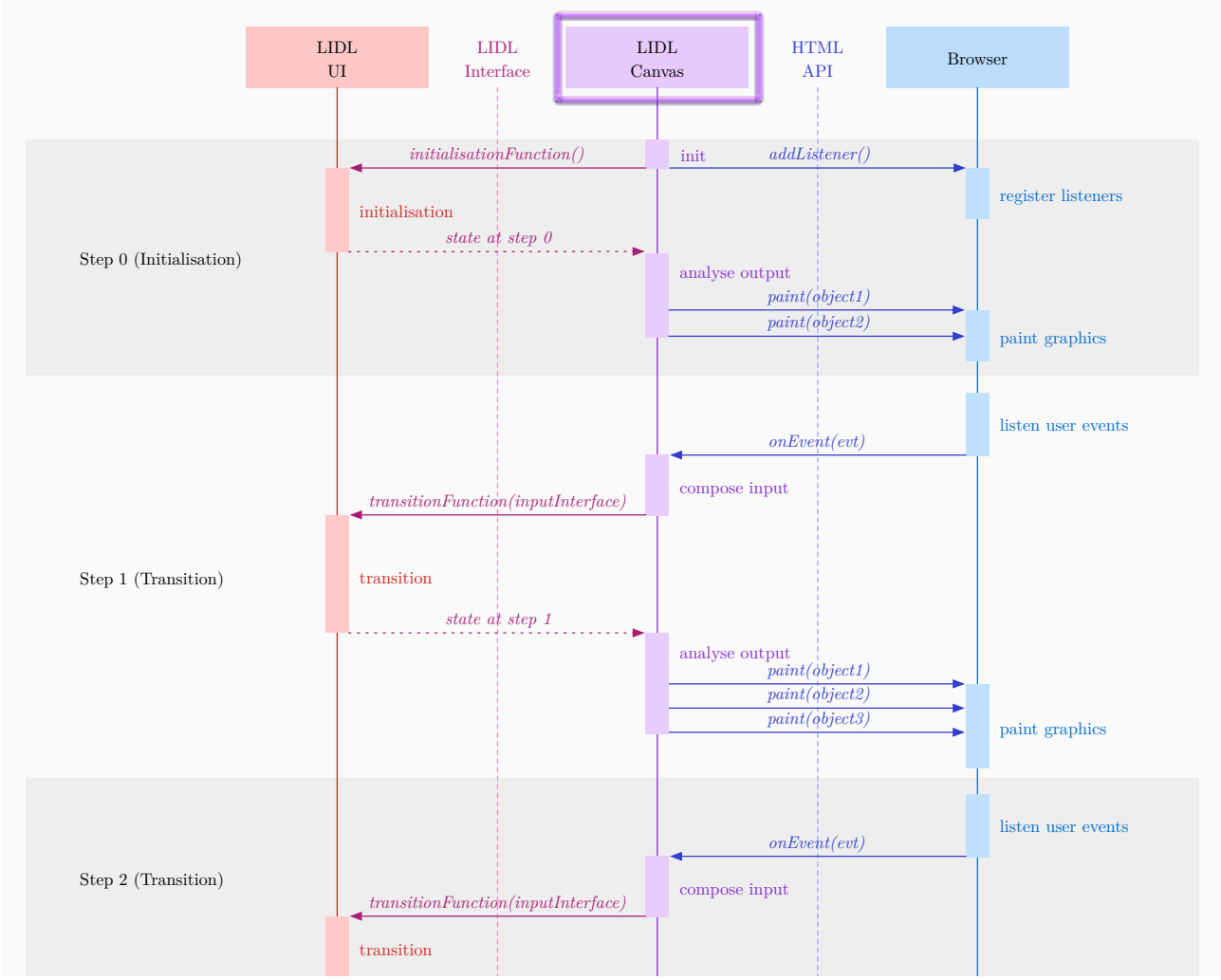


Figure 5.13: Sequence diagram of event handling in LIDL Canvas

5.3.4 Representation of graphics

The data format used to describe vector graphics is based on an abstraction of the HTML canvas 2D context API [127] defined by the World Wide Web Consortium (W3C). Each paint call of the API is associated with a data structure that represents it. This allows to represent nested API calls as nested data. For example graphics shown on Figure 5.14 is described in LIDL in Listing 5.15.

```
group containing
  text "ok"
  centred at 0 , 0
  with font Helvetica 15 Normal
 filled in white
  rectangle
  centred on 0 , 0
  of size 30 , 20
  filled in blue
 with black shadow
 of bluriness 10
 offset by 0 , 5
rotated 30 degrees
```

Listing 5.15: Example LIDL code describing the graphics of Figure 5.14

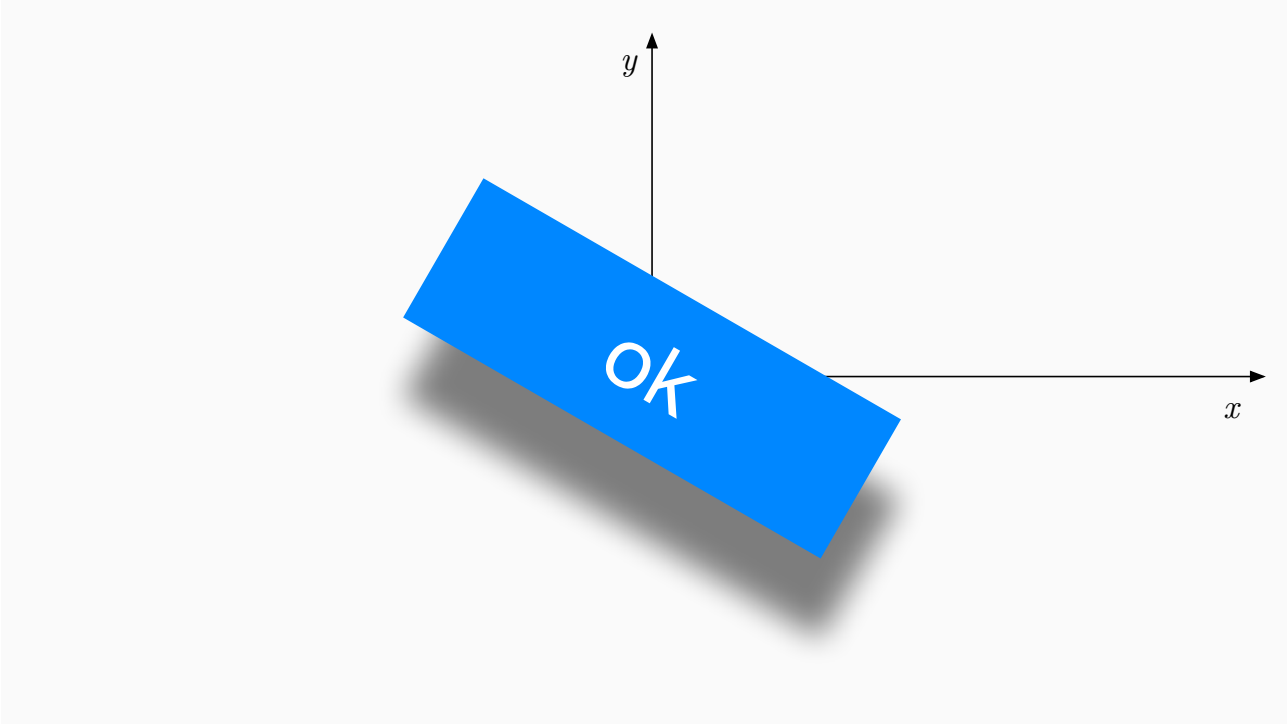


Figure 5.14: Example graphics rendered in LIDL canvas

5.3.5 Limitations and perspectives

The LIDL canvas is still limited and could be further extended in various directions:

- Take into account other modalities that are supported by standard browser APIs: Sound API, Gamepad API, Multitouch, Motion and Location API.
- Offer an additional interface, this time not directed to the user, but to other machines that can be accessed using HTTP requests, basically offering the possibility of performing network operations using LIDL.
- Extend the concept to other platforms. The only currently supported platform is web browsers. It is possible to port LIDL canvas to other platforms, for example by compiling LIDL to C/C++ and using OpenGL as a drawing API instead of the Canvas API.

5.4 Sandbox

The LIDL sandbox was developed in collaboration during the internship of Dorra Gastli at École Nationale Supérieure d'Électrotechnique, d'Électronique, d'Informatique, d'Hydraulique et des Télécommunications (ENSEEIHT). It is a web-based application (Web app) that aims at providing a lightweight Integrated Development Environment (IDE) to learn and interact with LIDL.

5.4.1 Architecture

LIDL playground is built using JS and HTML. It relies on several external packages, which are available on Node Package Manager (NPM) [128]:

- Facebook React: A UI toolkit for the web. It allows to create complex applications in a simple and modular way. It is described in detail in the state of the art of this thesis, Chapter 2. All the UI of the LIDL playground is programmed using React in the JSX language.
- Lidl: The package that contains the LIDL compiler and other utilities, described in detail in Section 5.1. It is used as the backend of the playground, handling all LIDL processing related tasks such as parsing, compilation, execution...
- Lidl-canvas: The package that contains the LIDL canvas, described in Section 5.3. It is used as a quick prototyping tool, allowing to execute and interact with LIDL GUIs in the browser.

The playground embeds the LIDL compiler in a component called runtime engine. This components listens for changes made by the user in the source code and tries to compile it into executable code. If this succeeds, then the executable is used and ready to be tried and tested by the user. The playground contains several views that allow to create, analyse and interact with LIDL programs:

- Code editor: Editors that allow to create LIDL programs. There are two variants of those:
  - Text editor: A classical textual code editor. Text entered using this editor is parsed using LIDL parser before being sent to the rest of the program
  - Visual editor: A code editor that is based on blocks. Blocks can be created by typing text, or they can be selected, dragged and dropped in order to ease the work of the programmer. The Block editor allows to edit directly the AST, so no parsing is required, and it is impossible to create syntax errors using this editor.
- Code analysis: A view that displays the code of the LIDL application once that all definitions are expanded. This code is equivalent to the code entered in the code editor, but much longer and lower level since it only contains base interactions. The code analysis view also displays statistics about

this code, such as the number of functions used, the number of identifiers and the number of state variables in the LIDL application.
- Generated code: A view that displays the ECMAScript code generated from the LIDL program.
- Graph Viewer: A view that displays the code of the current LIDL application in a graphical manner, allowing to visualise data flow. graphically.
- Scenario Editor: A view that displays and allow to edit the current scenario. A scenario is a sequence of inputs that can be fed in the input interface of the LIDL program.
- Trace Viewer: A view that displays the trace of the LIDL program. The trace is a sequence of states of the interface of the LIDL program. The trace viewer allows to see the inputs and outputs of the LIDL program in a table.
- LIDL Canvas: An instance of the LIDL canvas described in Section 5.3, supporting the execution of GUIs written using LIDL directly in the browser.

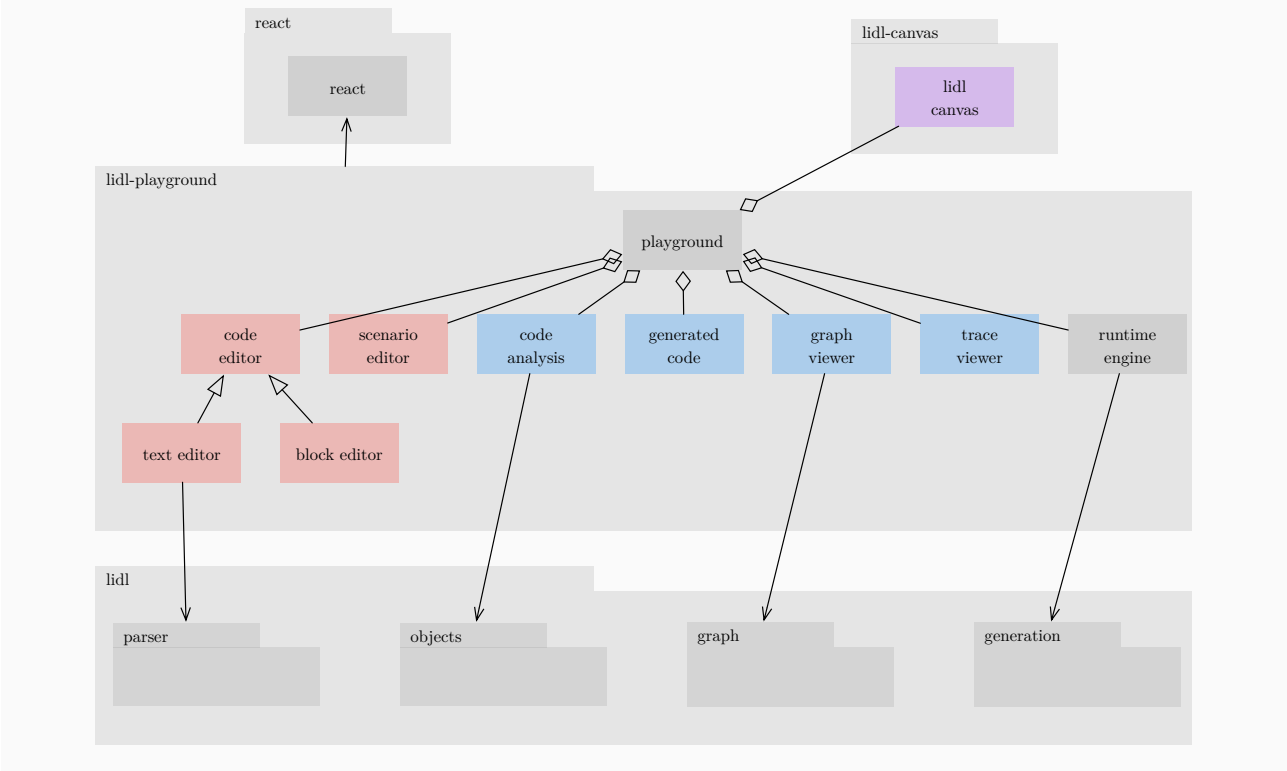


Figure 5.15: Architecture of the LIDL sandbox

5.4.2 Usage

The LIDL sandbox is a complete application with a flexible UI layout which allows to display various data related to LIDL systems. Figure 5.16 and Figure 5.17 show annotated screenshots of the LIDL sandbox in different configurations.

The flexible UI layout of the LIDL sandbox supports several use cases:

- Prototyping of simple UIs:
  1. Code a LIDL UI in the code editor
  2. Test the UI in the LIDL canvas
- Learning LIDL:
  1. Code a LIDL interaction in the code editor
  2. Set up a scenario in the Scenario editor

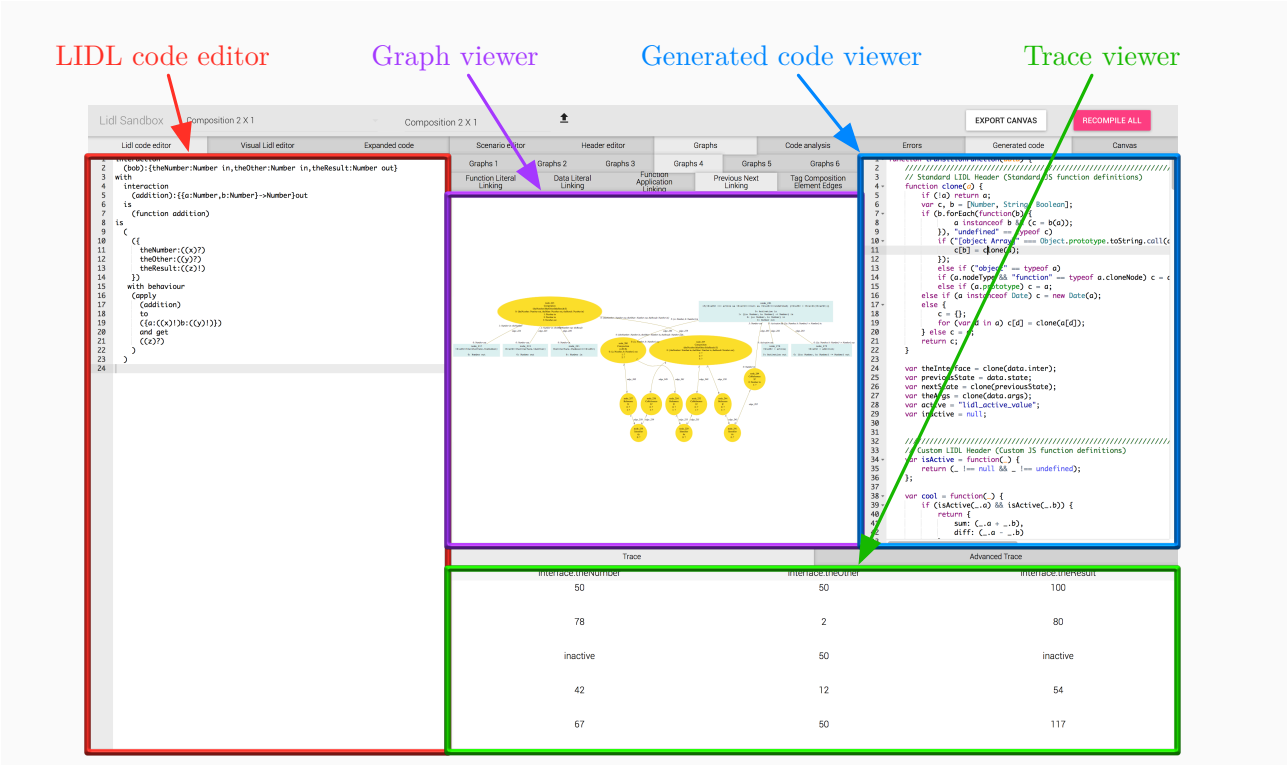


Figure 5.16: As screenshot of the LIDL sandbox

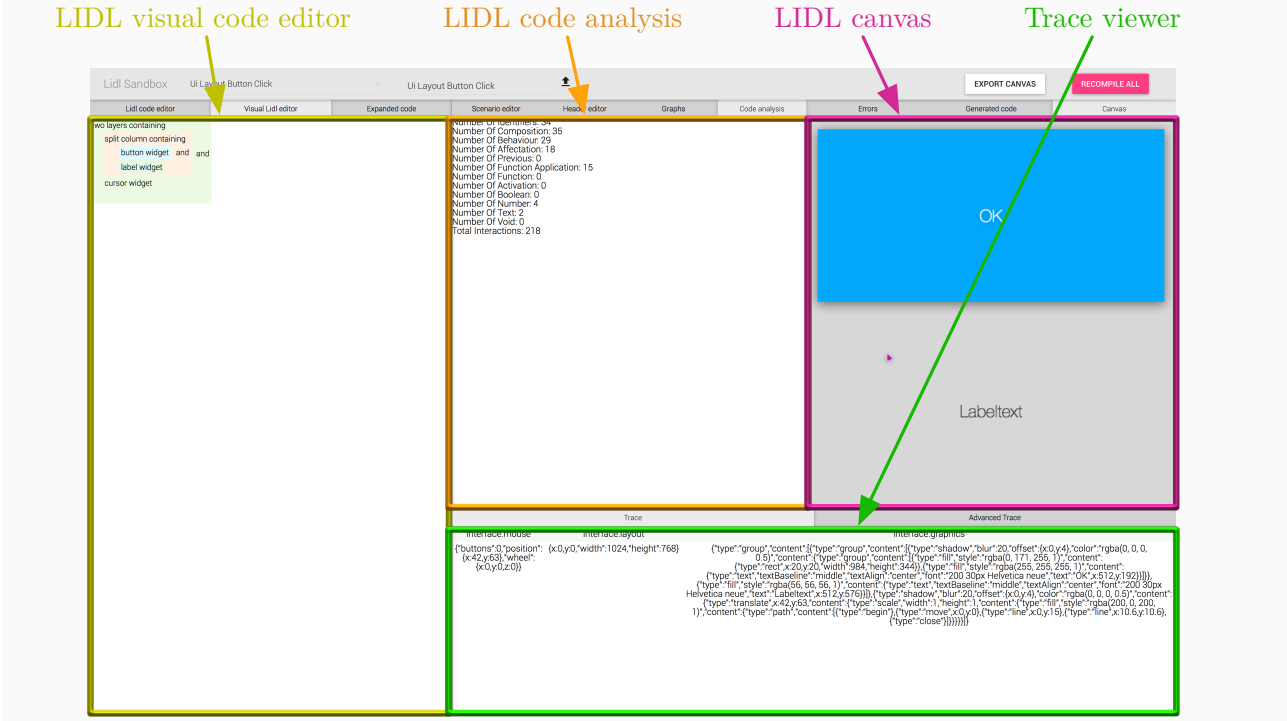


Figure 5.17: As screenshot of the LIDL sandbox

   3. Visualise execution result in the Trace Viewer
   4. Visualise the generated code in the Generated code viewer
- Debugging the LIDL compiler:
   1. Code a LIDL interaction in the code editor
   2. If compilation fails, visualise the compiler state at various steps using the graph viewers
- Expanding LIDL:
   1. Create new functions in the Header editor
   2. Integrate these functions with LIDL using the LIDL code editor
   3. Set up a test scenario in the Scenario editor
   4. Visualise the result in the Trace Viewer
- Analysing a LIDL program:
   1. Create a LIDL program using the code editor
   2. Visualise metrics about the compiled program using the Code analysis tool

Figure 5.18 shows the general workflow of the LIDL sandbox. Red boxes denote interactive views, blue boxes represent processing modules, and ovals represent data being exchanged between these modules. We note that the LIDL sandbox offers views at each step of the compilation and execution process, supporting inspection of the whole compilation and execution process.

5.4.3 Limitations and perspectives

The flexible UI of the IDE supports many different use cases, but it could be interesting to provide more specialised versions of the LIDL sandbox, oriented more toward specific use cases, such as learning to code using LIDL, or creating large complex systems using LIDL.

Being a Web-App, LIDL sandbox has performance limitations that could be overcome by developing a native desktop version.

Limitations of the LIDL compiler also apply to the LIDL sandbox, since it embeds the LIDL compiler. Expanding the subset of LIDL supported by the compiler would improve the LIDL sandbox user experience.

Finally, several bugs purely related to the LIDL sandbox UI still have to be solved (container resizing can sometimes behave incorrectly, rendering of large graph fails, rendering problems depending on browser versions ...).

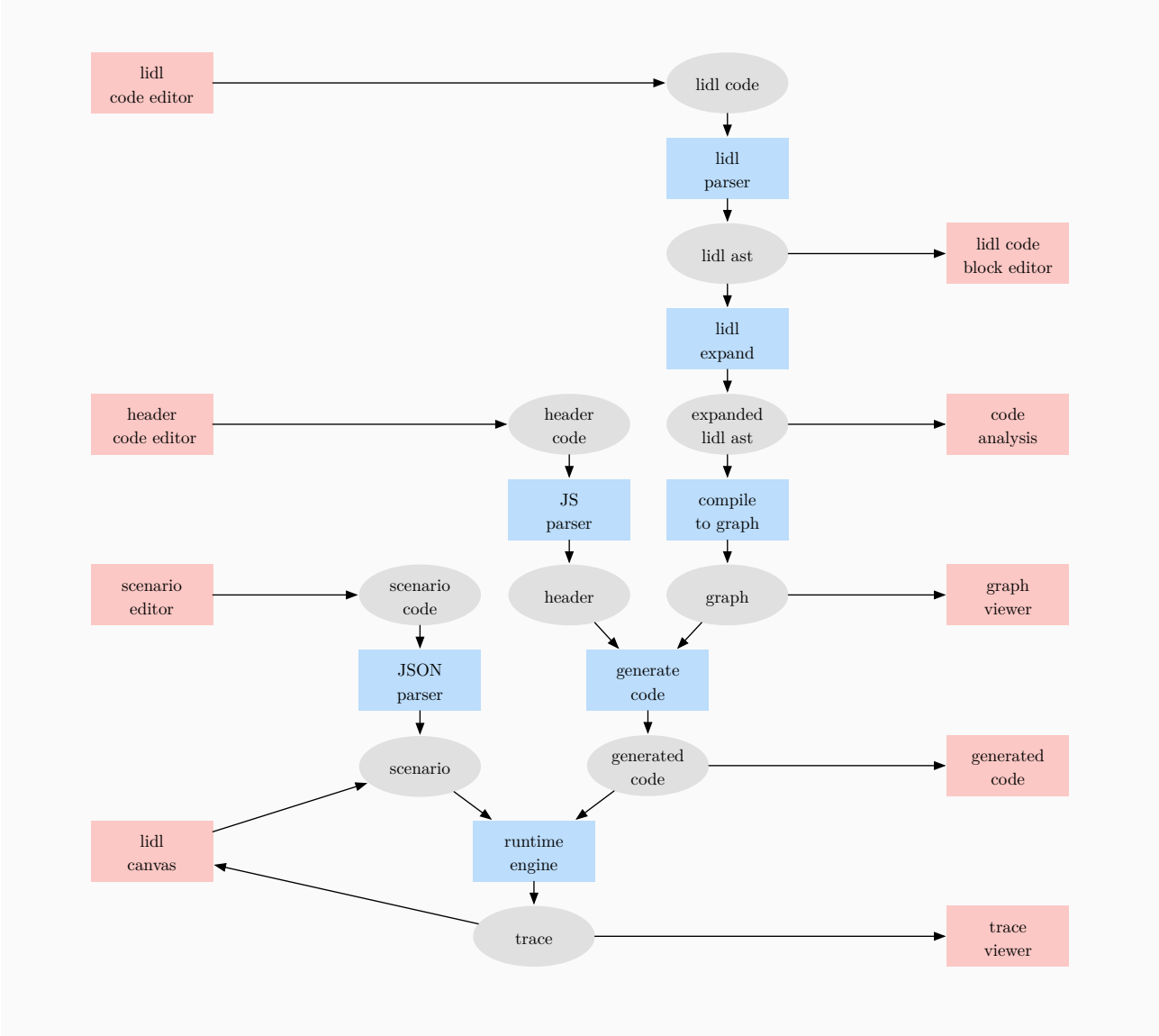


Figure 5.18: Data flow within the LIDL sandbox

Chapter 6

Case study

Measuring programming progress by lines of code is like measuring aircraft building by weight

Bill Gates

In this chapter, we will put LIDL (Chapter 4) and the geomorphic view (Chapter 3) to use on a case study. In a first place, the topic is presented in a general and informal way (Section 6.1). Then, we use LIDL to describe the underlying model of this case study (Section 6.2). We then proceed to describe the abstract UI using LIDL (Section 6.3). Then we present the concrete UI and how it is related to the abstract UI (Section 6.4). We present a task model related to the abstract UI. Finally, we use the case study to assess how LIDL provides solutions to the requirements.

6.1 Presentation

6.1.1 Overview

This case study focuses on the description of an alarm management system for a fleet of rovers. The system is depicted in Figure 6.1, it is part of a larger tool supporting other aspects of the supervision of a fleet of rovers. Various alarm events are generated by the rovers, and the UI maintains a list of alarms (List on the top half of the UI in Figure 6.1). The user can acknowledge and clear alarms, or proceed in resolving the problem that caused an alarm. When the user proceeds in resolving an alarm, a trouble shooting list containing the various step required to solve the problem is displayed as a list, allowing the user to follow a specific process (List on the bottom half of the UI in Figure 6.1).

6.1.2 Architecture

Figure 6.2 presents the global architecture of the system using the geomorphic view (Chapter 3). The architecture is made of four interactive systems: A user, a UI, and two rovers. This case study focuses on modelling the UI at various levels of abstraction. In Section 6.2 we use LIDL to describe the model on which the User and the rovers act. In Section 6.3 we describe the abstract UI of the system, and in Section 6.4 we describe the concrete UI of the system.

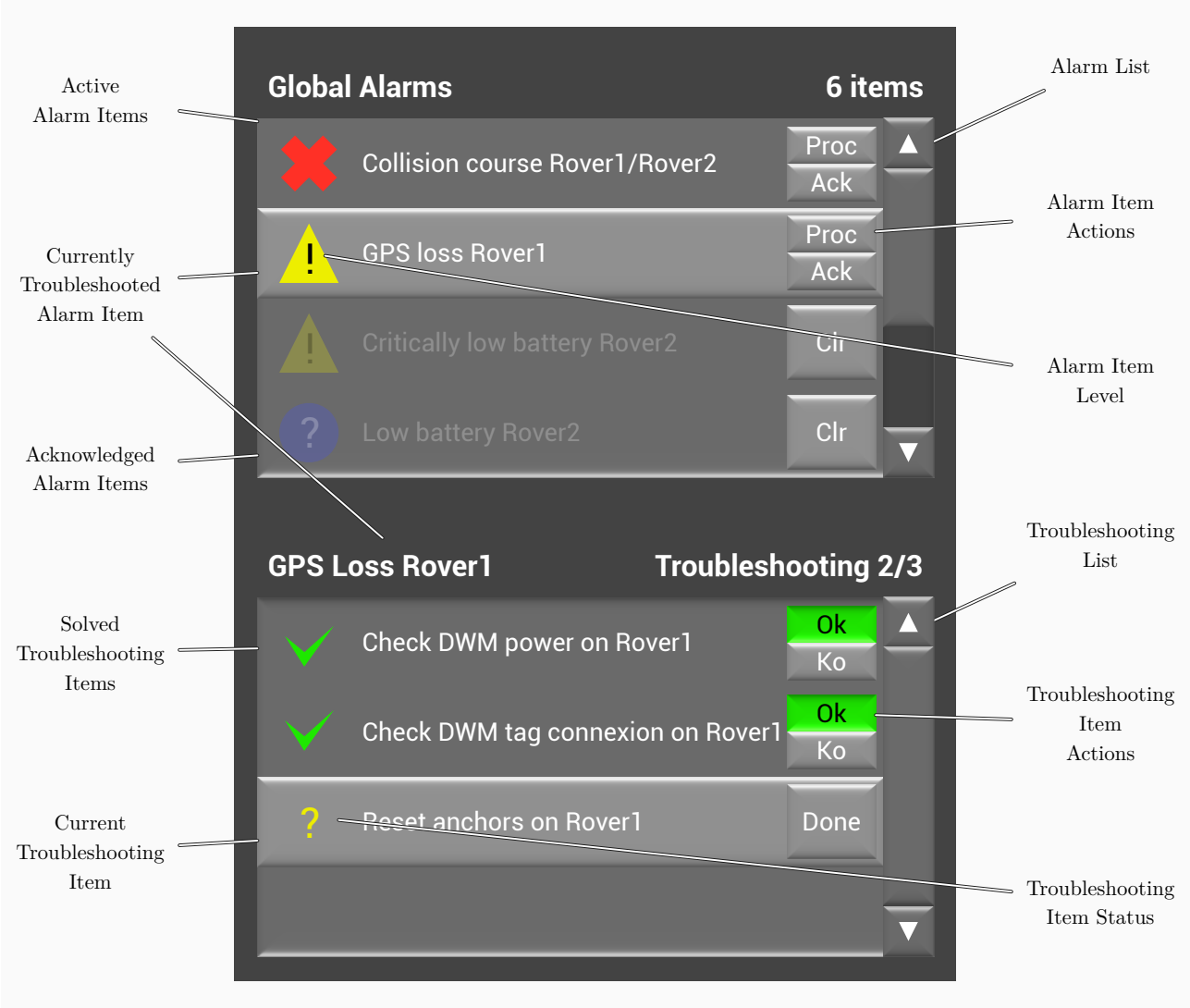


Figure 6.1: A screenshot of the HMI described in the case study

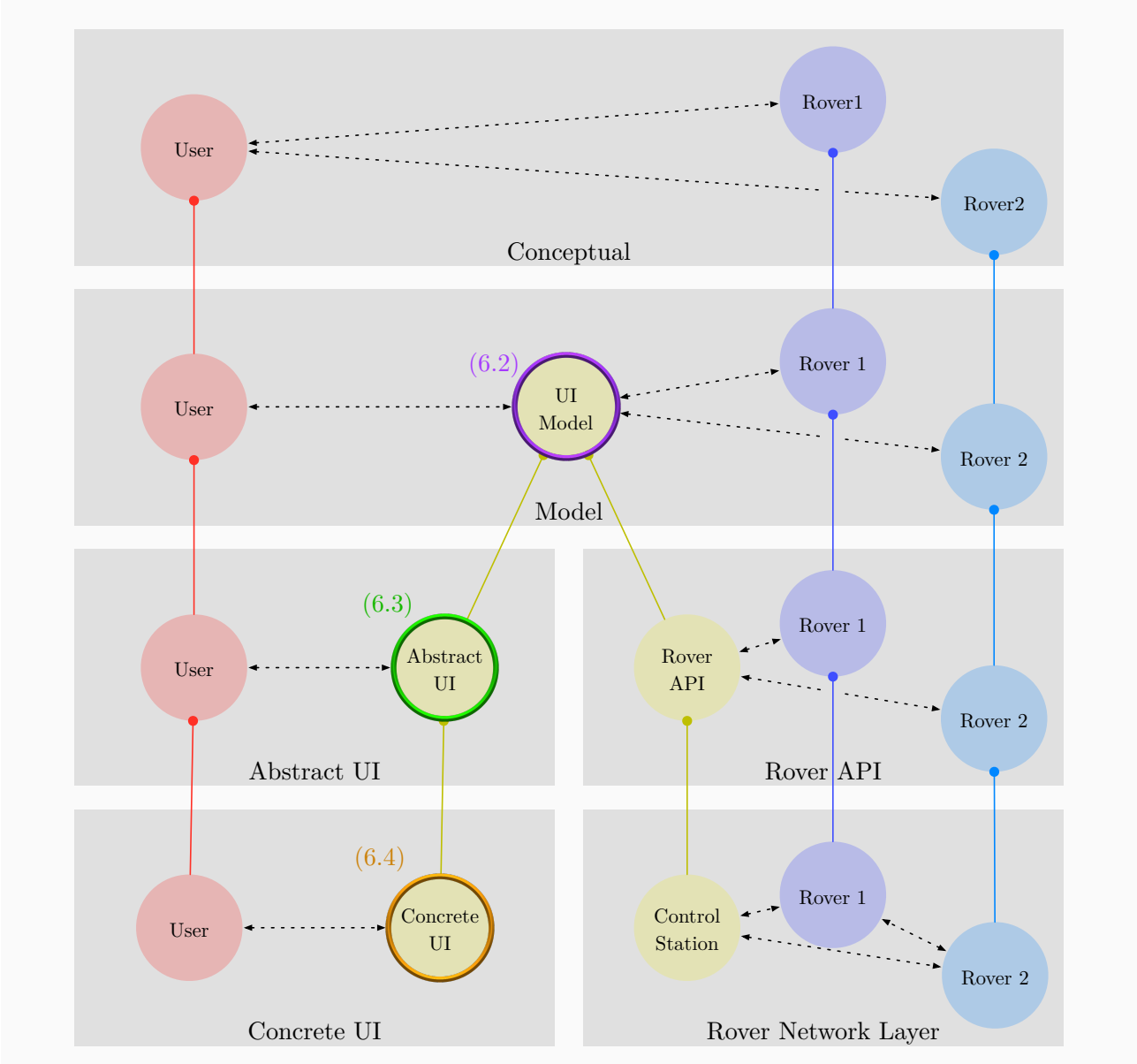


Figure 6.2: Geomorphic view of the architecture of the system described in the case study

6.2 Model

Before focusing on the actual UI of the program, we are going to describe the Model, using LIDL. The model is a high-level abstract entity which is is managed by the UI. The User and the Rovers can perform actions on the model. The rovers can add new alarms to the model, while the user can manage alarms contained in the model. In that sense, the model is a communication channel between the User and the Rovers, at a high level of abstraction.

The architecture of the model is presented as a UML class diagram in Figure 6.3. It is made of four classes: Rover (description of a Rover), TroubleShootingStep (an action that is presented to the User to help him fix a problem), Alarm (the description of a problem involving one or several rovers), and Model (The global model, a list of Alarms).

The notion of class does not exist in LIDL. Instead of describing classes like in most-object oriented programming languages, the approach is merely similar to that of functional programming: objects are defined using data types (equivalent of the fields of the class), and interactions that involve these data types (equivalent of the methods of the class). In the next paragraphs, we will present how some of the classes shown in Figure 6.3 are implemented using LIDL. Subsection 6.2.1 presents the Alarm object, and Subsection 6.2.2 presents the Model object. The other objects of the model, TroubleShootingStep and Rover, are not presented here since their implementation is similar to the implementation of the Alarm and Model objects.

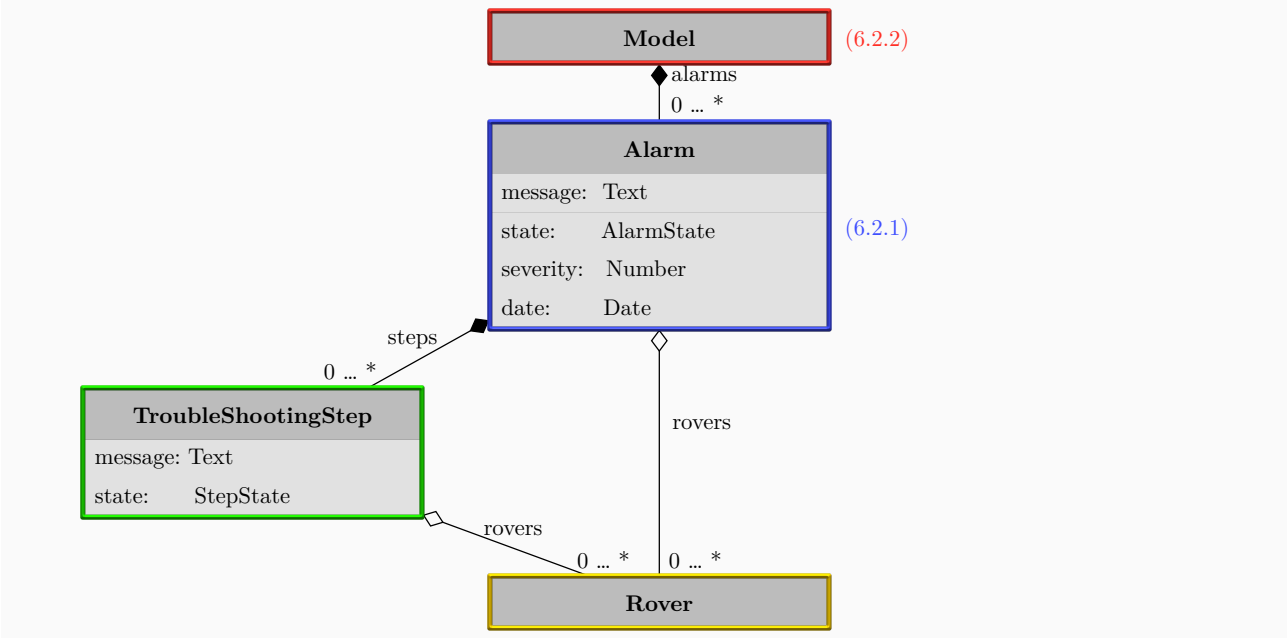


Figure 6.3: UML class diagram of the model managed by the UI

6.2.1 The Alarm object

The Alarm object represents an alarm. In this case study, alarms are relatively complex objects for two reasons. The first reason is that alarms have six different attributes (message, state, date, severity, rovers, steps). The second reason is that alarms have a non-trivial lifecycle. They can go through different states during their life, before finally being cleared.

In order to tackle this complexity comprehensively, we will address these two aspects separately. In 6.2.1.1, we define the `Alarm` object and basic associated interactions. Then, in 6.2.1.2, we implement

a Finite State Machine (FSM) in LIDL in order to formally model the life cycle of alarms. Finally, we combine these two elements in order to end up with a full-featured data type with a complex life cycle.

6.2.1.1 The Alarm data type

The LIDL equivalent of the concept of class (as in Object-oriented programming) is to define a data type that holds the attribute of the class, and a set of interactions which implement the "methods" of the class.

Definition Listing 6.1 shows the definition of the `Alarm` data type. It is a composite data type that contains a message (e.g. "Robot collision", "Low battery"...), information about alarm severity level (e.g. information, warning, error...), the date at which the alarm appeared, the state of the alarm (e.g. incoming, acknowledged, cleared...), and a list of rovers involved in the alarm.

```
1 data Alarm is
2   {
3     message      : Text,
4     severity     : Number,
5     date         : Date,
6     state        : AlarmState,
7     rovers       : [Rover],
8     steps        : [TroubleShootingStep]
9   }
```

Listing 6.1: The Alarm data type

Constructor This data type definition is automatically associated with an appropriate composition interaction (see 4.2.6.3) which allows to construct `Alarm` objects. For example, the following code is an interaction that constantly outputs an `Alarm` with specific attributes:

```
1 ({
2   message      : ("Problem!")
3   severity     : (1)
4   date         : (2016-01-01 T 12:00:00 Z)
5   state        : (acknowledged)
6   rovers       : ([ (rover1) ])
7   steps        : ([ (step ("do this")), (step ("do that")) ])
8 })
```

We would like to be able to create `Alarm` objects more simply than by using the above composition interaction, which requires six parameters (one for each field of the `Alarm` data type).

Listing 6.2 presents the definition of a simpler constructor for the `Alarm` data type. This constructor takes only three parameters instead of six. The parameters are the severity of the alarm (Line 3), the message of the alarm (Line 4) and the rovers involved in the alarm (Line 5). The constructor automatically sets the date of the alarm to the current date (Line 11), sets its state to 'incoming' (Line 12), and fetches the troubleshooting steps associated with the error message from a troubleshooting

database (Line 14-15). These elements are then combined using a composition interaction (Line 8-16), in order to output an instance of the Alarm data type, in accordance with the interface of this interaction: `Alarm out` (Line 6).

```
1  interaction
2    ( alarm
3      of severity (theSeverity: Number in)
4      with message (theMessage: Text in)
5      involving (theRovers: [Rover] in)
6    ): Alarm out
7  is
8    ({
9      message      : (theMessage),
10     severity     : (theSeverity),
11     date         : (now),
12     state        : (incoming),
13     rovers       : (theRovers)
14     steps        : (get from database the troubleshooting steps
15                     for alarm showing (theMessage) involving (theRovers) )
16   })
```

Listing 6.2: A constructor for the Alarm data type

Accessors The `Alarm` data type is also automatically associated with interactions called accessors, which allow to access its attributes. This accessors follow the usual "dot notation" accessor syntax. For example, the following LIDL code allows to access the severity of an `Alarm` object:

```
1  ((myAlarm).severity)
```

Mutations Now that we have defined a data type and a way to create instances of this data type, a programmer would probably like to implement methods to modify `Alarm` objects. But there is a problem: LIDL data is immutable: interactions can either receive data (through an `in` interface) or send data (through an `out` interface). Interactions cannot modify data and send it back to the same interface.

As an example, imagine we want to implement a method to modify the message of an alarm (known as a "Setter" in object-oriented vocable). A LIDL equivalent of a "Setter" method would look like this:

```
1  interaction
2    (set state of (theAlarm: Alarm in/out) to (theState: AlarmState in)): Behaviour
3  is
4    (...)
```

The problem with this definition is that the interface of `theAlarm` (Line 2) is impossible to define: it would have to be *both* an input (`Alarm in`) and an output (`Alarm out`).

As a consequence, we could come up with a definition that has properly defined LIDL interfaces, such as the following:

```
interaction
  ( set state of (alarmBefore: Alarm in) to (theState: AlarmState in)
    returns (alarmAfter: Alarm out) ): Behaviour
is
  (...)
```

This definition is valid. It takes an `Alarm` called `alarmBefore`, and outputs a new `Alarm` called `alarmAfter`, which is similar to `alarmBefore` except that its state was changed to `theState`. This definition could be used in order to modify the state of `Alarm` objects.

However, interactions following this model would not be very convenient to use. Imagine we want to modify several attributes of an alarm. We would have to chain such interactions in this way:

```
(all
  ( set state   of (alarmBefore) to (incoming) returns (temp1)      )
  ( set message of (temp1)       to ("toto")   returns (temp2)      )
  ( set date    of (temp2)       to (now)      returns (alarmAfter) )
)
```

As we see, this approach is not composable. We need four different alarms variables just to change properties of an alarm. It leads to intractable and heavy code as soon as nested data types are involved.

A composable way to describe modifications of objects is to define composable mutations. This approach is explained with more detail in Chapter A. The main idea is to create composable interactions that allow to declaratively represent modifications of data. These interactions are called mutations. This approach is inspired by the concept of lens in functional programming [129], and leads to simple code even in the case of complex nested data types. Each data type comes with a set of standard mutations. Listing 6.3 presents an example of use of the standard mutation associated with the `Alarm` data type.

```
1  (modify (alarmBefore) into (alarmAfter) using
2    (mutation{
3      message    : (change to ("toto"))
4      severity   : (change nothing)
5      date       : (change to (now))
6      state      : (change to (incoming))
7      rovers     : (change nothing)
8      steps      : (change nothing)
9    })
10 )
```

Listing 6.3: An example of use of the mutation of the alarm data type

In this approach, we use an interaction called a mutation (Lines 2-9). The mutation interaction declaratively represents the modification of a piece of data, an `Alarm` in this example. In this example, the mutation specifies that the message of the alarm should be changed to 'toto' (Line 3), the date of the alarm should be changed to the current date (Line 5), and the state of the alarm should be changed to incoming (Line 6). Other attributes of the alarm should not be modified (Lines 4, 7 and 8). This mutation is then used by providing it a source `Alarm` (`alarmBefore`) and a target `Alarm` (`alarmAfter`) (Line 1-10).

The interest of this approach is that mutations are composable. This allows to describe mutations of complex nested objects in a simple way.

In this case study, once an alarm is created, it can only be modified in two ways. The first one is the modification of its state as a consequence of user actions (detailed in 6.2.1.2). The other one is the modification of its troubleshooting list as a consequence of troubleshooting actions performed by the user. Listing 6.4 presents the definitions of these two possible mutations.

```
1  interaction
2    (change state by (changeOfState: MutationOfAlarmState)): MutationOfAlarm
3  is
4    (mutation{
5      message      : (change nothing)
6      severity     : (change nothing)
7      date         : (change nothing)
8      state        : (changeOfState)
9      rovers       : (change nothing)
10     steps        : (change nothing)
11   })
12
13 interaction
14   (change steps by (changeOfSteps: MutationOfTroubleShootingStepList)):
     ↪ MutationOfAlarm
15 is
16   (mutation{
17     message      : (change nothing)
18     severity     : (change nothing)
19     date         : (change nothing)
20     state        : (change nothing)
21     rovers       : (change nothing)
22     steps        : (changeOfSteps)
23   })
```

Listing 6.4: Mutations of the alarm data type

These two mutations are specialisations of the generic mutation for the `Alarm` data type. The mutation of state (Lines 4-11) is a mutation of the `Alarm` data type where only the `state` attribute is modified, and the mutation of steps (Line 16-23) is a mutation where only the `steps` attribute is modified.

Conclusion on the Alarm data type The definition of the alarm data type is now complete. In this subsection we have defined the following entities:

- The `Alarm` data type, which automatically defines several interactions:
  - A generic constructor interaction which allows to create `Alarm` objects
  - Several generic accessors interactions which allows to read attributes of `Alarm` objects
  - A generic mutation interaction which allows to modify `Alarm` objects.
- A custom constructor based on the generic constructor
- Two custom mutations based on the generic mutation

These interactions allow us to perform all necessary actions on `Alarm` objects. However, we still need to specify the behaviour of `Alarms` in response to user actions.

6.2.1.2 Alarm life cycle

Life cycle of Alarms as a FSM Figure 6.4 shows the lifecycle of an alarm as a Finite State Machine (FSM). The next paragraph is an informal description of this FSM.

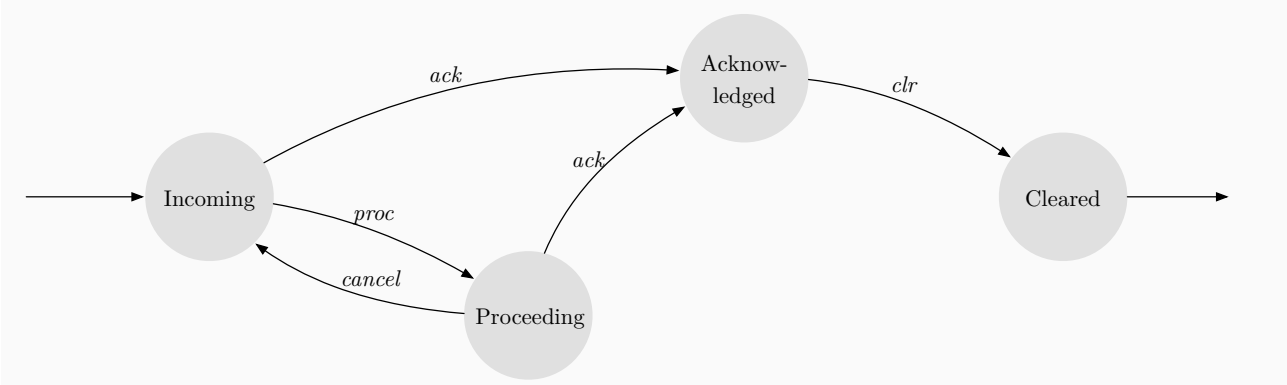


Figure 6.4: Finite State Machine representing the life cycle of Alarms

Each alarm can be in four states: Incoming, Proceeding, Acknowledged and Cleared. Four actions can be performed on alarms: *proc*, *cancel*, *ack* and *clr*. Alarms initially arrive in the Incoming state. The user then has two options: proceeding in resolving the alarm (*proc*), or acknowledging the alarm without trying to solve it (*ack*). Acknowledging the alarm (*ack*) sets the alarm in the Acknowledged state, while proceeding with the alarm (*proc*) sets it in the Proceeding state. Only one alarm can be in the Proceeding state at a time, if the user decides to proceed on an alarm, then other alarms in the Proceeding state go back to the Incoming state (*cancel*). When an alarm is in the Proceeding state, a troubleshooting list displays a sequence of actions to solve the problem. When the user has finished troubleshooting the alarm, the alarm can be acknowledged (*ack*), going to the Acknowledged state. Alarms which are acknowledged are displayed in shaded tones, and only offer one action to the user: Clear (*clr*). When cleared (*clr*), an alarm disappears from the alarm log.

Implementation of the FSM with LIDL FSMs can be implemented easily in LIDL using switch interactions. Listing 6.5 presents the implementation for the FSM of Figure 6.4. This implementation is based on the declaration of two data types and an interaction.

A data type describes the possible states (Lines 1-4) and another describes the possible actions received by the FSM (Lines 6-9).

The interaction describes the behaviour of the FSM. It is written as a switch interaction (Lines 18-25). This interaction encodes two aspects of the state machine: the accepted actions in each state, and the target state for each action each state. It means that depending on the value of the couple `((previousState), (action))`, the output `((nextState), (accepts))` can take various values. For example, Line 19 means that when `(previousState)` is `(incoming)` and `(action)` is `(ack)` then the transition is accepted, and `(nextState)` should be `(acknowledged)`.

```
data
  AlarmState
is
  incoming | proceeding | acknowledged | cleared

data
  AlarmAction
is
  ack | proc | cancel | clr

interaction
  ( Alarm in state   ( previousState: AlarmState  in  )
    accepts action   ( action:        AlarmAction in  )
                     ( accepts:       Boolean     out )
    and goes to      ( nextState:     AlarmState  out )
  ):Behaviour
is
  ( switch  ( (previousState), (action) ) : ( (nextState)    , (accepts)    )
    case    ( (incoming)     , (ack)    ) : ( (acknowledged) , (true)       )
    case    ( (incoming)     , (proc)   ) : ( (proceeding)   , (true)       )
    case    ( (proceeding)   , (ack)    ) : ( (acknowledged) , (true)       )
    case    ( (proceeding)   , (cancel) ) : ( (incoming)     , (true)       )
    case    ( (acknowledged) , (clr)    ) : ( (cleared)      , (true)       )
    default                               : ( (previousState), (false)      )
  )
```

Listing 6.5: LIDL implementation of the FSM of the alarm life cycle

Integrating the FSM as a mutation of AlarmState The FSM is entirely specified in one place, in Listing 6.5. However, this representation is not very convenient, because it is not really composable. In order to make it composable, we are going to create an interaction that represents the mutation of an `AlarmState` in reaction to an `AlarmAction`. Listing 6.6 presents the definition of a such an interaction. This interaction complies with the `MutationOfAlarmState` interface (Line 2). This means that this interaction will be composable inside the `MutationOfAlarm` interactions defined in Listing 6.4. This interaction is simply a mutation (Lines 5-8) which has a specific behaviour (Line 9). This specific behaviour is the behaviour specified in Listing 6.5, which represents the life cycle of alarms.

```
1  interaction
2    (reaction to (theAction: AlarmAction in)):MutationOfAlarmState
3  is
4    (
5      ( mutation
6        from (thePreviousState)
7        to (theNextState)
8      )
9      with behaviour
10     ( Alarm in state   (thePreviousState)
11       accepts action   (theAction) ()
12       and goes to      (theNextState)
13     )
14   )
```

Listing 6.6: Integration of the FSM to specify a mutation of Alarm objects

This mutation interaction will then be reusable easily in other parts of the code, since it complies with the mutation pattern. For example, here we reuse the mutation we just defined as part of a mutation of an `Alarm`:

```
1  (change state by (reaction to (ack)))
```

Another useful application of the FSM definition is described in Listing 6.7. It is an interaction that returns a boolean telling if a given `AlarmAction` can be accepted by the FSM in a given `AlarmState`.

```
1  interaction
2    ((theState: AlarmState) accepts (theAction: AlarmAction)): Boolean out
3  is
4    (
5      (acceptance)
6      with behaviour
7      ( Alarm in state (theState)
8        accepts action (theAction) (acceptance)
9        and goes to    ()
10     )
11   )
```

Listing 6.7: An interaction that tells if an action is accepted by the FSM in a given state

This interaction can be reused to test if a given `AlarmAction` can be performed on an alarm in a given `AlarmState`. For example, the following interaction outputs true only if the `currentState` is Acknowledged:

```
((currentState) accepts (clr))
```

6.2.2 The Model object

The `Model` object is a list of `Alarm` objects. The data type that represents the Model is described on Line 1 of Listing 6.8. This definition of a list data type automatically defines several interactions, whose signature is presented on Lines 3-16. Line 3-4 is the definition of a constructor for an empty list, Lines 6-7 presents the definition of an interaction that extracts an element of the list.

Then, Lines 9-16 presents several mutations of the List data type, which allow to add or remove objects to and from the list.

Lines 18-19 presents a mutation of the List that consists in modifying a specified element according to a given mutation.

Additionally, Lines 21-31 presents the definition of an additional mutation that consist in sorting the list of alarms. In this example, the precise sorting algorithm is defined in the target programming language, using a function called `sortListBySeverity`.

```
1  data Model is [ Alarm ]
2
3  interaction
4    (empty list):Model out
5
6  interaction
7    (element (index: Number in) of (model:Model in)): Alarm out
8
9  interaction
10   (add (alarm: Alarm in)): MutationOfModel
11
12 interaction
13   (remove (alarm: Alarm in) ): MutationOfModel
14
15 interaction
16   (remove element (index: Number in) ): MutationOfModel
17
18 interaction
19   (change element (index: Number in) by (mut:MutationOfAlarm)): MutationOfModel
20
21 interaction
22   (sort): MutationOfModel
23 is
24   (
25     ( mutation from (nonSortedModel)  to (sortedModel) )
26     with behaviour
27     ( (sortedModel) = (function sortListBySeverity) (nonSortedModel) )
28   )
```

Listing 6.8: The Model data type and some associated interactions

6.2.3 Conclusion

Other parts of the model can be described using LIDL, by following a process similar to the process shown above for the `Alarm` and `Model` objects. The basic idea is to define composite data types, and create simple interactions that act on these data types. The interactions do not mutate the objects, because it is not allowed by LIDL, but they take objects as inputs and return modified objects as outputs. This approach for domain modelling is similar to what is done in functional programming.

Use of the model Once the description of the model and related interactions is complete, LIDL can be used to describe operations on the model. Listing 6.9 presents a piece of LIDL code that performs several operations. It starts by initialising an empty model and performs several mutations on it before sending it to a variable called `resultingModel` (Line 1). The detail of modifications applied to the model is the following: First, adds two alarms to the model (Lines 3-4), then sort the model (Line 5), remove the second element (Line 6), and finally acknowledge the first `Alarm` of the list, using the interaction defined in Listing 6.6 (Line 6).

```
1 (modify (empty model) into (resultingModel) using
2   (compose mutations
3     (add (alarm of severity (2) with message ("GPS Loss") involving ([(R1)])) )
4     (add (alarm of severity (1) with message ("Collision")involving ([(R1),(R2)]))
5     (sort)
6     (remove element (1))
7     (change element (0) by (change state by (reaction to (ack))))
8   )
9 )
```

Listing 6.9: An example declarative description of manipulations of a model

As we see in Listing 6.9, LIDL provides a way to declaratively describe various operations and manipulations on the model. We saw earlier that LIDL allow to declaratively describe the structure of the model using data types.

Behaviour of the UI Model interactor At this point, we have everything needed to describe the behaviour of the whole "UI Model" interactor of our global architecture (Figure 6.2). Listing 6.10 presents the definition of this behaviour. This interaction has two interfaces: `userModifications` (Line 3), which represents the modifications performed on the model by the user (performing actions on alarms, clearing alarms...), and `roverModifications` (Line 4) which represents modifications performed by the rovers (adding new alarms to the list). The behaviour is simple: it holds a variable called `model` (Line 7), which is initially an empty list (Line 5), and at each instant, modifications performed by the user and modifications performed by the rovers are combined in order to update the model (Line 9).

```
interaction
  ( behaviour of the UI model interactor
    between (userModifications: MutationOfModel)
    and (roverModifications: MutationOfModel)
  ):Behaviour
is
  ( (model) is a flow initially (empty model)
    modified by (compose mutations (userModifications) (roverModifications))
  )
```

Listing 6.10: Definition of the behaviour of the UI Model interactor

Behaviour of other interactors of the same layer Actually, we can go further and also express the behaviour of the other interactors of the "Model" architecture layer. As shown in Figure 6.5, this layer contains 3 kinds of interactors: User, UI Model, and Rover. We just described the behaviour of the UI Model interactor.

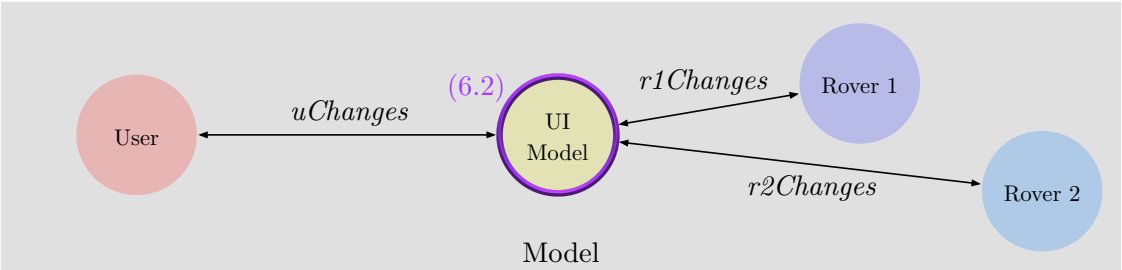


Figure 6.5: A view of the architecture model at this point in the development

Listing 6.11 presents a model of the interactions performed by the User. This can be seen as a high level task model of the user. Here, we model a simplistic task model where, at each instant, the user can either (Line 6) do nothing (Line 7), or perform any action (Line 9) on any alarm of the list (Line 8).

```
1  interaction
2    ( behaviour of the user performing (changes: MutationOfModel) ):Behaviour
3  with
4    interaction (list):Model out is (value before (mut))
5  is
6    ((changes) = (either
7      (change nothing)
8      (change element (any number between (0) and (length of (list)))
9        by (change state by (reaction to (either (ack) (proc) (clr)))))
10   ))
```

Listing 6.11: Definition of the behaviour of the user interactor in the model layer

The same way, Listing 6.12 presents a model of interactions performed by Rovers. At each instant, a rover can either change nothing (Line 7), or add an Alarm whose parameters are chosen non-deterministically.

```
interaction
  ( behaviour of a rover performing (changes: MutationOfModel) ):Behaviour
with
  interaction (list):Model out is (value before (mut))
is
  ((changes) = (either
    (change nothing)
    (add (alarm of severity (any number between (0) and (3))
          with message (either ("Collision") ("GPS Loss"))
          involving (any subset of ([(R1),(R2),(R3)])))))
  ))
```

Listing 6.12: Definition of the behaviour of the rover interactor in the model layer

Verifying the model layer Once the three kinds of interactors of the model layer are defined, we can execute the model layer. Listing 6.13 presents the behaviour of the whole model: it is the parallel composition of the behaviour of the four interactors. Three interactors have non-deterministic behaviours (the User and the Rovers), while one has a deterministic behaviour (the UI Model). The interaction model can be used to perform simulations or model checking.

```
interaction
  (model layer):Behaviour
is
  (all
    ( behaviour of the user performing (uChanges) )
    ( behaviour of the UI model interactor between (uChanges) and (roverChanges))
    ( behaviour of a rover performing (r1Changes) )
    ( behaviour of a rover performing (r2Changes) )
    ((roverChanges) = (compose mutations (r1Changes) (r2Changes)))
  )
```

Listing 6.13: The global interactions between interactors in the Model layer

The description of the Model Layer part of our UI is now complete, we can now go one layer down, and implement the abstract UI of our case study example.

6.3 Abstract interface

In this section, we go through the implementation of the abstract UI of the alarm management system. As most UIs, LIDL UIs are described in a hierarchical way. This leads to a tree of components, where simple components are composed in order to form a unique component at the top of the hierarchy. This large composite component represents the UI. This approach is valid for abstract UIs as well concrete UIs, the only difference being the nature of components and composition operators.

Top-down and bottom-up approaches could be used to design a compound system such as this one. In the top-down approach, one starts by describing high level components such as screens and containers, and then goes into the further details of components, sub-components and so on until the most basic widgets are reached. In the bottom-up approach, one starts by describing the most basic components, then proceeds to composing more complex components, and then composing composite components

until the whole system is described. In this section, we use the bottom-up approach, starting by describing basic components, and then assembling them to create the final UI.

Figure 6.6 shows a part of the component hierarchy of the UI. In this section, to keep things simple, we will not describe each and every element of the whole UI component hierarchy, but only a subset of it, which is framed in colour on Figure 6.6. We will start by describing a simple button, then we will increase complexity by describing an alarm item, then the alarm list, and finally the whole UI.

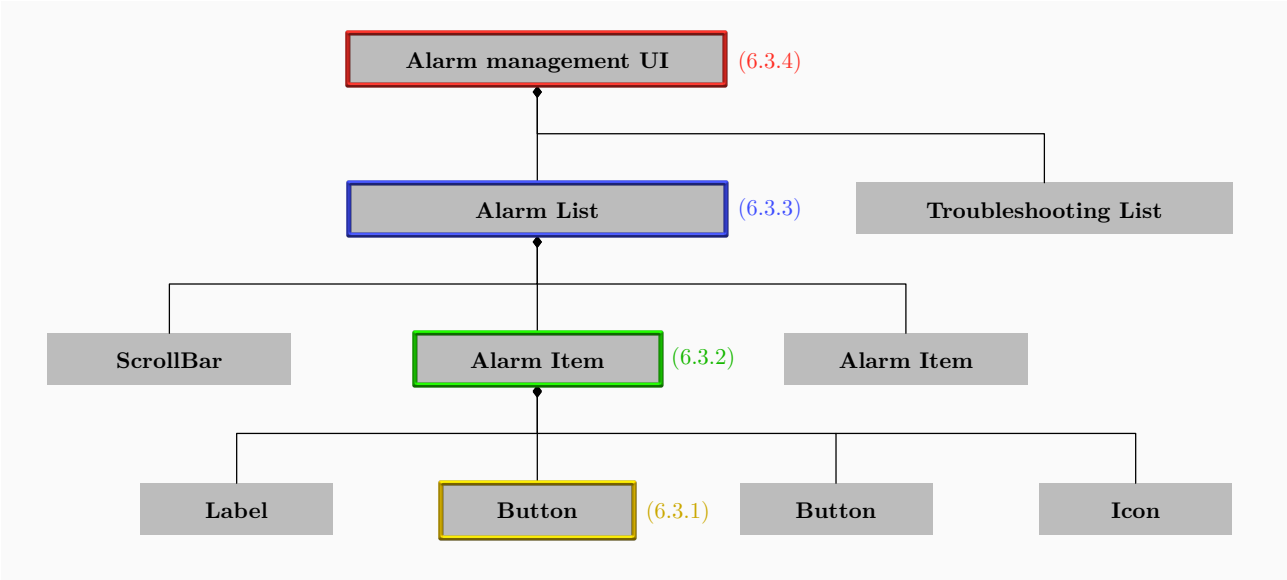


Figure 6.6: Component hierarchy of the UI.

6.3.1 Button

Buttons are a standard UI component, and as such, they are part of the LIDL standard library. The implementation of basic widgets in the standard library is explained in Subsection 5.2.4. As a consequence, as a developer using LIDL and its standard library, we do not have to implement the button component, and we can use it as is. However, for the sake of completeness, we will here go in further detail about the abstract button widget.

Interface The interface of the abstract button widget is described by the following piece of LIDL code. It is composed of two parts: `label` represents the text of the button shown to the user, and `click` represents clicks performed by the user on the button.

```
interface
  AUIButton
is
  {
    label: Text out,
    click: Activation in
  }
```

Interaction The interaction performed by the abstract button is described by the following piece of LIDL code. The interaction is made of a composition interaction (`{label:(text) click:(action)}`)

(Line 7-8), which is nested inside an `(if()then()else())` block which acts as a guard. When the `(enabled)` argument is *active*, then the button is enabled, the button's `label` is given by the `(text)` argument, and `clicks` coming from the user trigger the `(action)` argument. When `(enabled)` is *inactive*, then nothing happens: the button `label` is *inactive*, and user `clicks` are not taken into account.

```
1  interaction
2    ( button (enabled:Activation in)  displaying (text:Text in)
3      triggering (action:Activation out) ):AUIButton
4  is
5    ( if (enabled)
6      then ({
7        label: (text)
8        click: (action)
9      })
10     else ()
11   )
```

Execution Table 6.1 shows an example of the execution of the abstract button widget. Red text represents output values, while blue text represents input values. At instant 0, we see that the label of the button is equal to the value of the `(text)` argument. At instant 2, we see that the user clicks on the button, and this triggers the `(action)` argument. At instant 5, we see that the user tries to click on the button again, but since the `(enabled)` argument is *inactive* at this time, the `(action)` argument is not triggered. Finally at instant 8, we see that the button is active, but since no valid `(text)` is given for it to display, it does not display any label, but it is still be clickable.

Instant	(button (enabled) displaying (text) triggering (action))	(enabled)	(text)	(action)
0	{*label* : 'Ok', *click* : *inactive*}	*active*	'Ok'	*inactive*
1	{*label* : 'Ok', *click* : *inactive*}	*active*	'Ok'	*inactive*
2	{*label* : 'Ok', *click* : *active*}	*active*	'Ok'	*active*
3	{*label* : 'Ok', *click* : *inactive*}	*active*	'Ok'	*inactive*
4	{*label* : *inactive*, *click* : *inactive*}	*inactive*	'Ok'	*inactive*
5	{*label* : *inactive*, *click* : *active*}	*inactive*	'Ok'	*inactive*
6	{*label* : *inactive*, *click* : *inactive*}	*inactive*	'Ok'	*inactive*
7	{*label* : 'Ok', *click* : *inactive*}	*active*	'Ok'	*inactive*
8	{*label* : *inactive*, *click* : *active*}	*active*	*inactive*	*active*

Table 6.1: Timing diagram of an execution of the abstract button widget

6.3.2 Alarm Item

The alarm item component is a composite component that presents an individual `Alarm` object to the user, and allows the user to apply modifications to it. This component is not part of the standard library, and this subsection presents and explain how it is developed using LIDL.

Interface Listing 6.14 presents the interface of the alarm item abstract UI component. It is a composition of several abstract UI components. The interface is composed of an `icon` denoting the severity of the alarm, the `message` associated with the alarm, two booleans (`proceeding` and `acknowledged`) describing the state of the alarm, and the `proceed`, `acknowledge` and `clear` buttons.

```
1  interface
2    AUIAlarmItem
3  is
4    {
5      icon         : Text out,
6      message      : Label,
7      proceeding   : Boolean out,
8      acknowledged : Boolean out,
9      proceed      : AUIButton,
10     acknowledge  : AUIButton,
11     clear        : AUIButton
12   }
```

Listing 6.14: The Alarm Item component interface

Interaction The interaction performed by the alarm item component is more complex than the simple button. Listing 6.15 presents the code of the alarm item component.

Line 2, we see that this interaction complies with the `AUIAlarmItem` interface that we just defined: the main role of this interaction is indeed to present the abstract UI component to the user. We also see that this interaction has an argument called `mutation`, which represents the second role of this interaction: allowing the user to modify an `Alarm` object. It is important to realize that the interface `MutationOfAlarm` is a composite interface that contains two subinterfaces: an `Alarm in` (the `Alarm` to be displayed or modified) and an `Alarm out` (the `Alarm` after it has undergone modifications performed by the user).

Line 4-9 present several interaction definitions that are nothing more than shortcuts to clarify the expression of the interaction. For example, `(theAlarm)` is defined as a shortcut for `(value before (mutation))`: it is the `Alarm` object that has to be displayed by the component.

Line 12 states that the icon displayed to the user depends on the severity of `theAlarm`. The relationship between the icon and the severity of the alarm is defined prceisely in the definition of the interaction `(icon associated with ())`.

Line 13 states that the message shown to the user is displayed using a label widget which is always active, and that the text displayed is fullMesssage, as defined Line 9: the message of the alarm (Line 6), a space, and the list of rover names, separated by slashes (Line 8).

Line 15 states that the alarm item component is displayed as “acknowledged” (i.e. faded out in the concrete interface, like the two last items in Figure 6.1) if and only if the state of the alarm is acknowledged.

Lines 16-18 contain the definition of the "Proc" button. It uses the Button component defined in Subsection 6.3.1. Lines 16 states that the button is active only if `theState` is a state which accepts the *proc* action. This reuses the interaction defined in Listing 6.7. Line 17 states that the Text displayed on the button is always "Proc". Finally, Line 18 states that when clicked on by the user, the button has one effect: the alarm item component performs a modification which consists in changing the state of the alarm in reaction to the *proc* action. This reuses the mutation defined in Listing 6.6.

Lines 19-24 define the "Ack" and "Clr" buttons in the same way.

```
1  interaction
2    ( alarm item performing (mutation:MutationOfAlarm') ):AUIAlarmItem
3  with
4    interaction (theAlarm):   Alarm out      is (value before (mutation))
5    interaction (theState):   AlarmState out is ((theAlarm).state)
6    interaction (theMessage): Text out       is ((theAlarm).message)
7    interaction (theRovers):  [Rover] out    is ((theAlarm).rovers)
8    interaction (roverNames): Text out       is ((theRovers) separated by ("/"))
9    interaction (fullMessage):Text out       is ("(theMessage) (roverNames)")
10 is
11   ({
12     icon          : ( icon associated with ((theAlarm).severity) )
13     message       : ( label (active) displaying (fullMessage))
14     proceeding    : ( (theState) == (proceeding) )
15     acknowledged  : ( (theState) == (acknowledged) )
16     proceed       : ( button ((theState) accepts (proc))
17             displaying ("Proc")
18             triggering ((mutation) = (change state by (reaction to (proc)))) )
19     acknowledge   : ( button ((theState) accepts (ack))
20             displaying ("Ack")
21             triggering ((mutation) = (change state by (reaction to (ack)))) )
22     clear         : ( button ((theState) accepts (clr))
23             displaying ("Clr")
24             triggering ((mutation) = (change state by (reaction to (clr)))) )
25   })
```

Listing 6.15: The Alarm Item interaction

Execution Table 6.2 shows an execution of the alarm item. Red denote output while blue denote input of the interaction. The execution is compliant with the specification of the interaction in Listing 6.15. In particular we note that there are two interfaces. The first interface is `(mutation)`, it represents the modifications made to the alarm item by the user. The second interface, `(alarm item performing (mutation))`, is the main interface of this interaction, it represent the abstract UI component presented to the User.

- The state of the alarm after mutation (Columns 5-9) is different from the state of the alarm before mutation (Columns 1-4) only when the user performs an action by clicking on a button (columns called *click* which are active at instants 1, 3, 5, and 7). This behaviour is caused by the assignments of mutations as a consequence of user clicks, (Lines 18, 21 and 24 of Listing 6.15).
- The message displayed to the user (Column 11) is compliant with the process specified in Lines 8-9 of Listing 6.15. It is made of the message of `theAlarm` followed by the list of rovers.

Instant	(mutation)								(alarm item performing (mutation))									
	before				*after*				*icon*	*message*	*selected*	*proceed*		*acknowledge*		*clear*	
	severity	*message*	*state*	*rovers*	*severity*	*message*	*state*	*rovers*		*text*		*text*	*click*	*text*	*click*	*text*	*click*
0	1	'Collision course'	*acknowledged*	['Rover1', 'Rover2']	1	'Collision course'	*acknowledged*	['Rover1', 'Rover2']	'Critical.png'	'Collision course Rover1/Rover2 '	*true*					'Clr'	
1	1	'Collision course'	*acknowledged*	['Rover1', 'Rover2']	1	'Collision course'	*cleared*	['Rover1', 'Rover2']	'Critical.png'	'Collision course Rover1/Rover2 '	*true*					'Clr'	*active*
2	2	'GPS Loss'	*incoming*	['Rover1']	2	'GPS Loss'	*incoming*	['Rover1']	'Warning.png'	'GPS Loss Rover1'	*true*	'Prc'		'Ack'			
3	2	'GPS Loss'	*incoming*	['Rover1']	2	'GPS Loss'	*acknowledged*	['Rover1']	'Warning.png'	'GPS Loss Rover1'	*false*	'Prc'		'Ack'	*active*		
4	2	'GPS Loss'	*acknowledged*	['Rover1']	2	'GPS Loss'	*acknowledged*	['Rover1']	'Warning.png'	'GPS Loss Rover1'	*false*					'Clr'	
5	2	'GPS Loss'	*acknowledged*	['Rover1']	2	'GPS Loss'	*cleared*	['Rover1']	'Warning.png'	'GPS Loss Rover1'	*false*					'Clr'	*active*
6	3	'Low battery'	*incoming*	['Rover1']	3	'Low battery'	*incoming*	['Rover1']	'Info.png'	'Low battery Rover1'	*false*	'Prc'		'Ack'			
7	3	'Low battery'	*incoming*	['Rover1']	3	'Low battery'	*proceeding*	['Rover1']	'Info.png'	'Low battery Rover1'	*false*	'Prc'	*active*	'Ack'			
8	3	'Low battery'	*proceeding*	['Rover1']	3	'Low battery'	*proceeding*	['Rover1']	'Info.png'	'Low battery Rover1'	*true*	'Prc'		'Ack'			

Table 6.2: Execution of the alarm item interaction

6.3.3 Alarm List

The alarm list component displays a list of alarm items to the user, and allows him to perform actions on those. During the definition of the model (Section 6.2), we saw that the list of alarms has a variable length: alarms can be added by rovers and cleared by the user. However, a UI has limitations: the list of alarms can only display at most 4 alarms at a time, as shown in Figure 6.1. A scrollbar allows the user to navigate in the whole list and get access to all the alarms, but only 4 alarms can be visible at a time. This point is can have potential consequences in terms of usability, and is modelled at the abstract UI level, in this section.

Interface The `AUIAlarmList` interface is described in Listing 6.16. It is an Abstract UI component that contains a list of 4 visible alarm items (Lines 9-12), as well as additional widgets: a label displaying the title of the list (Line 5), a label displaying the number of items in the list (Line 6), and a scrollbar allowing to navigate in the list (Line 7).

```
1  interface
2    AUIAlarmList
3  is
4    {
5      title        : AUILabel,
6      itemCount    : AUILabel,
7      scrollBar    : AUIScrollBar,
8      items        : {
9        0 : AUIAlarmItem,
10       1 : AUIAlarmItem,
11       2 : AUIAlarmItem,
12       3 : AUIAlarmItem
13     }
14   }
```

Listing 6.16: The alarm list interface

Interaction Listing 6.17 presents the definition of the Alarm list interaction, which specifies the behaviour of the alarm list abstract UI component.

The Alarm List interaction uses several instances of the Alarm Item component and composes them in a list displayed to the user. It also describes precisely the behaviour of the title label, item count label and scroll bar.

Line 2, we see that the interaction complies with the `AUIAlarmList` interface that we just defined. The interaction also takes one argument called `mutation`, which complies with the `MutationOfModel` interface. This argument represents the fact that this abstract UI component allows the user to visualise a `Model` object (defined in Section 6.2), and to modify it (e.g. by changing the state of certain alarms of the alarm list).

Line 4, we defined `(list)` as a shortcut for the model received by the component, which is received through the `mutation` argument.

The interaction associated with the `AUIAlarmList` interface is expressed on Lines 7-19. The `title` label is always active, and displays a constant text (Line 8). The `itemCount` label is always active, and displays a text of the form “12 items” (Line 9). The `scrollBar` uses a widget which is predefined

in the LIDL standard library. The scroll bar is always active, and allows to scroll inside the list, and it performs a mutation on a number which represents the scroll position. This mutation is called changeOfScrollPosition.

Lines 14-17, we reuse the alarm item abstract UI component defined in Subsection 6.3.2, and instantiate it 4 times: once for each visible element in the list. Each of these 4 instances performs a modification on the alarm it represents.

Lines 21-30 specify additional behaviour of the alarm list component. Line 21 specifies that two behaviours are constantly activated. The first one (Line 22) specifies the behaviour of the scroll position. It is a number which is initially 0, and is modified at each instant according to the changeOfScrollPosition mutation, which is managed by the scroll bar component (Line 12). The constantly active behaviour is shown on Lines 23-29. This behaviour states that the mutation performed by the Alarm List component on the model is a complex modification of the alarm list. Five modifications are performed on the list: first, the four visible alarms are modified according to the user actions (Lines 24-27), and then the list is sorted. This mutation is performed at each instant, but most of the time, the mutation consists in not changing anything on the model.

```
1  interaction
2    (alarm list performing (mutation:MutationOfModel')):AUIAlarmList
3  with
4    interaction (list): Model out is (value before (mutation))
5  is
6  (
7    ({
8      title     : (label (active) displaying ("Global Alarms"))
9      itemCount : (label (active) displaying ("(length of (list)) items"))
10     scrollBar : (scrollbar (active)
11                   for a list of (length of (list)) elements
12                   performing (changeOfScrollPosition))
13     items     : ({
14       0 : (alarm item performing (changeOfAlarm0))
15       1 : (alarm item performing (changeOfAlarm1))
16       2 : (alarm item performing (changeOfAlarm2))
17       3 : (alarm item performing (changeOfAlarm3))
18     })
19   })
20   with behaviour
21   (all
22     ((scrollPosition)is a flow initially(0) modified by (changeOfScrollPosition))
23     ((mutation) = (compose mutations
24       (change element ((scrollPosition) + (0)) by (changeOfAlarm0))
25       (change element ((scrollPosition) + (1)) by (changeOfAlarm1))
26       (change element ((scrollPosition) + (2)) by (changeOfAlarm2))
27       (change element ((scrollPosition) + (3)) by (changeOfAlarm3))
28       (sort)
29     ))
30   )
31 )
```

Listing 6.17: The alarm list interaction

6.3.4 Alarm management UI

The Alarm management abstract UI component is the main component of the system, on top of the hierarchy. It represents the whole UI presented to the user.

Interface Listing 6.18 presents the interface of the Alarm management component. It is made of two components: the `AUIAlarmList` component we just defined in Subsection 6.3.3 (Line 5), and the `AUITroubleshootingList` component, which we did not present in this case study, but which displays the troubleshooting list to the user (Line 6) (Bottom part on the screenshot of Figure 6.1).

```
1 interface
2   AUIAlarmManagement
3 is
4   {
5     alarms          : AUIAlarmList,
6     troubleShooting : AUITroubleshootingList
7   }
```

Listing 6.18: The alarm management interface

Interaction Listing 6.19 presents the interaction defining the Alarm management abstract UI component. Line 2, we note that this interaction follows the usual pattern for abstract UI components: First, it complies with the `AUIAlarmManagement`, which means it expresses the behaviour of the abstract UI component; and second, it has an argument called `mutation` that complies with the `MutationOfModel` interface, which means that this interaction allows the user to perform changes on the `Model` object.

The interaction associated with the `AUIAlarmManagement` interface is expressed on Lines 9-12. Line 10 states that the `alarms` part of the interface is associated with the alarm list UI component we defined in Subsection 6.3.3. This component performs changes on the list of alarms; these changes are denoted by the `changeOfAlarmList` mutation. Line 11 states that the `troubleShooting` part of the interface is associated with the UI component called `(troubleshooting list performing ())` which is not specified in this case study. This component performs changes on a list of troubleshooting steps; these changes are denoted by the `changeOfSteps` mutation.

Lines 14-19 specify an additional behaviour for this interaction. This behaviour consists in specifying the `MutationOfModel` performed by the interaction. This mutation represents changes of a `Model` object. This mutation is a composition of two simpler mutations. First (Line 15), the component performs the changes specified by the alarm list component of Line 10, which can change state of alarms or remove them. Then, in a second time (Line 16-18), the component performs a change which consists in modifying the `Alarm` which is at a given position in the list (`indexOfSelectedAlarm`). This `Alarm` is modified by the mutation presented on Line 17, using a mutation of the `Alarm` data type presented in Listing 6.4, which changes the list of troubleshooting steps associated with the alarm. This change in the troubleshooting list of the selected alarm is denoted by `changeOfSteps`, which is specified by the troubleshooting list UI component (Line 11).

```
interaction
  (alarm management UI performing (mutation: MutationOfModel')):AUIAlarmManagement
with
  interaction (indexOfSelectedAlarm): Number out is
    ( index of the element whose state is (proceeding)
      in (value after (changeOfAlarmList)))
is
(
  ({
    alarms          : (alarm list performing (changeOfAlarmList))
    troubleShooting : (troubleshooting list performing (changeOfSteps))
  })
  with behaviour
  ((mutation) = (compose mutations
    (changeOfAlarmList)
    (change element (indexOfSelectedAlarm) by
      (change steps by (changeOfSteps))
    )
  ))
)
```

Listing 6.19: The interaction of the alarm management abstract UI

6.3.5 Conclusion

Behaviour of the Abstract UI interactor We can now finally describe the behaviour of the whole "Abstract UI" interactor of our global architecture (Figure 6.2). Listing 6.20 presents the definition of this behaviour. This interaction has two interfaces: `userUI` (Line 3), which represents the abstract UI presented to the user, and `changesOfModel` (Line 4), which represents changes performed on the model by the user (acknowledging alarms, clearing troubleshooting steps...). The behaviour is trivial: the UI presented to the user is the main UI component described in Subsection 6.3.4.

```
interaction
  ( behaviour of the abstract UI interactor
    where (userUI: AUIAlarmManagement)
    performs (changesOfModel: MutationOfModel)
  ):Behaviour
is
  ( (userUI) = (alarm management UI performing (changesOfModel)) )
```

Listing 6.20: Definition of the behaviour of the Abstract UI interactor

Verifying the abstract UI Figure 6.7 presents a geomorphic view where the Model layer and the Abstract UI layer are depicted. We just defined the behaviour of the Abstract UI interactor. We can also define the behaviour of the User interactor in the Abstract UI layer. The description of this interactor is out of the scope of this Chapter, but it is done using LIDL, in a way which is dual with the way we just described the Abstract UI. Indeed, while the Abstract UI interactor transforms actions on the UI into changes on the Model, the User interactor transforms changes on the (mental) Model into actions on the UI.

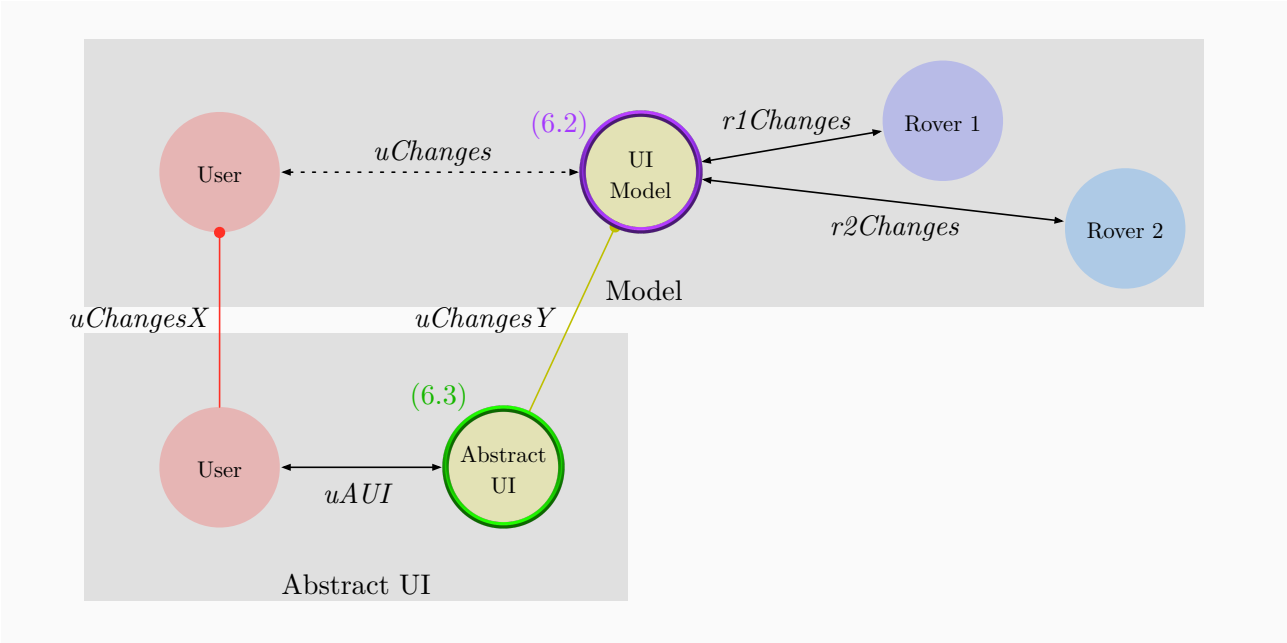


Figure 6.7: A view of the architecture model at this point in the development

Listing 6.21 presents the global behaviour of this model. We note its similarity with Listing 6.13. The difference is that this time, the User does not directly perform mutations on the Model, but he first translates it into actions to be performed on the Abstract UI (Line 7), and these actions of the User are translated by the Abstract UI into mutations on the Model (Line 8, using the interaction we defined in Listing 6.20). The interaction with the Rovers has not been impacted by the work done in this section. This corresponds to the changes between the models presented inFigure 6.7 and Figure 6.5.

```
1  interaction
2    (model layer):Behaviour
3  is
4    (all
5      ( behaviour of the user performing (uChangesX) )
6      ( behaviour of the user who wants to perform (uChangesX) using (uAUI))
7      ( behaviour of the abstract UI interactor where (uAUI) performs (uChangesY))
8      ( behaviour of the UI model interactor between (uChangesY) and (roverChanges))
9      ( behaviour of a rover performing (r1Changes) )
10     ( behaviour of a rover performing (r2Changes) )
11     ((roverChanges) = (compose mutations (r1Changes) (r2Changes)))
12   )
```

Listing 6.21: The global interaction between interactors in the Model and Abstract UI layers

This model can be used to perform simulations or model checking, in order to check that certain properties are verified. For example, we could check that this model is equivalent to the model defined in Section 6.2 where only the Model layer is present. If the abstract UI is specified correctly, it should be transparent: the user should be able to perform actions on the Model using the abstract UI as if he was performing actions directly on the Model, so the two models should be equivalent.

The abstract UI of our system is now completely defined, we can now add another layer of complexity to our UI: the concrete interface.

6.4 Concrete interface

In this section, we specify the concrete UI of our system, which includes notions of detailed graphics and concrete interactions devices. This concrete interface is implemented by the Concrete UI interactor presented in the architectural model of Figure 6.2.

In this section, we reuse abstract components defined in Section 6.3, and we add an additional layer of detail to them. As a consequence, we will follow the same path than previously: starting by describing a simple Button component (Subsection 6.4.1), we will then describe an alarm item component (Subsection 6.4.2), the alarm list (Subsection 6.4.3), and finally the complete UI (Subsection 6.4.4).

It is important to note that we will not duplicate code: the high level behaviour of the concrete widget is implemented by associating each concrete widget with a corresponding widget of the abstract UI defined in Section 6.3.

Another important point to notice is that concrete interface components all comply to the same interface. This is in contrast with the situation of abstract UI components presented in the Section 6.3, where each component had its own interface, which represented the abstract data flows between the user and the interface. Now, in the case of the concrete interfaces, the situation is different: the interface for all concrete widgets is the same. This interface is called `WimpUI`, and is part of the standard LIDL library, which we described in Listing 5.10 (The interfaces used in definitions of WIMP interfaces). The data flow between the user and the concrete UI is conceptually similar for each component: the user controls a mouse and keyboard, and sees a graphic representation. As a consequence, the definition of a concrete interface widget does not imply the definition of both an interface and an interaction, but only the definition of an interaction, which has to implement the fixed `WimpUI` interface.

6.4.1 Button

This component is a standard button, it is already defined in the standard LIDL concrete widget library, and a user of LIDL would not have to specify it. However, for the sake of completeness, we provide its definition in this section. Figure 6.8 presents in a simplified way how the concrete button interaction maps the abstract interface of the button (specified in Subsection 6.3.1) to the concrete user interface (represented graphically here).

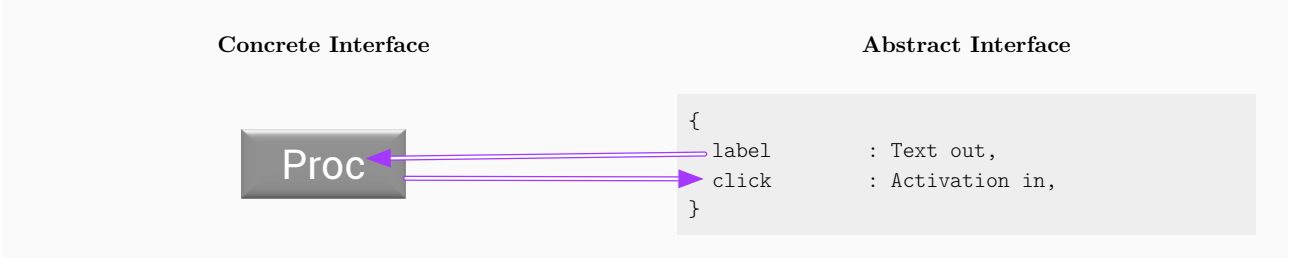


Figure 6.8: Mapping between concrete and abstract buttons

Listing 6.22 presents the LIDL definition of the concrete button. This definition is typical of most concrete UI components: It complies with the `WimpUi` interface (Line 2), and it is associated with the corresponding abstract interface component (Here, `abstractButton`, Line 2). The role of the interaction is then to output the graphical representation and manage the user input methods in order to concretise the abstract representation.

Lines 5-18 define the interaction associated with the `WimpUI` interface of the button. The interaction receives layout information and stores it in a variable called `layout` (Line 6). The interaction receives mouse information and stores it in a variable called `mouse` (Line 7). Lines 7-16 define the graphical representation of the button: It is a group of two graphical objects (Line 7). The graphics contains a text with a given font, at a given position (Lines 8-11). The graphics also contain a rectangle of a given size, at a given position, with a depth effect depending on the state of the button (pushed or not) (Lines 12-16).

Lines 19-30 define additional behaviour of the button. Lines 20-24 states that the button should be in a "pushed" state under certain conditions. These conditions are that the user presses the mouse button, and that the pointer was inside the button when the user started pushing the button. This behaviour specifies non-nominal interaction scenarios, like when the user starts clicking on the button, and then moves the pointer outside of the button, while keeping the mouse button pressed, which results in the button still being pressed, even though the pointer is outside of the button. Lines 25-29 specify that the actual "click" event received by the abstract interface is triggered when the mouse button is *released* by the user while the button is pushed.

This complex behaviour is expressed in 8 lines of code using LIDL, and allows to specify the precise semantics of the click on a UI button, and its translation into abstract click events. The specification of this behaviour uses several predefined interactions, such as `(()when last())`, `(() starts)` and `(() ends)`, which are defined in the "state related interactions" section of the standard library (Subsection 5.2.3).

```
interaction
  (standard button (abstract: AUIButton)): WimpUi
is
(
  ({
    layout    :  ( layout )
    mouse     :  ( mouse )
    graphics  :  ( group containing
        ( text ((abstract).text)
          centred at ((layout).center)
          with (font (Helvetica) (15) (Normal))
          filled (white) )
        ( ( rectangle
            centred on ((layout).center)
            of size ((layout).size)
            filled (grey) )
          with depth (if (buttonPushed) then (-1) else (1)) ) )
  })
  with behaviour (all
    ( when (
        ((mouse) click) and
        (((mouse) inside (layout)) when last (((mouse) click) starts)) )
      then (buttonPushed)
    )
    ( when (
        ((mouse) inside (layout)) and
        ((buttonPushed) ends))
      then ((abstract).click)
    )
  )
)
```

Listing 6.22: The definition of a concrete button widget

6.4.2 Alarm item

Figure 6.9 presents a simplified view of how the concrete Alarm item component maps the Abstract interface of the alarm item (specified in Subsection 6.3.2) to the concrete user interface (represented graphically here).

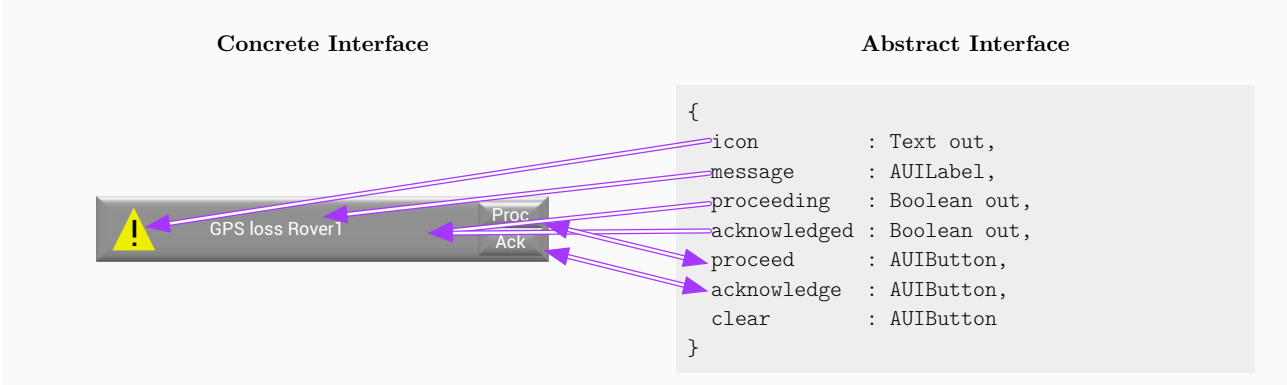


Figure 6.9: Mapping between concrete and abstract interface of a alarm list item

Listing 6.23 presents the definition of the concrete alarm item component using LIDL. This component follows the usual pattern of concrete UI components: it implements the `WimpUi` interface, and it takes an argument (called `abstract`), which represents the abstract UI component associated with the concrete UI component (Line 2).

Lines 5-13 specify the structure of this component: This component is a rectangle which has a specific graphic styling (called `itemStyle`), and which contains three other components which are laid out in a row (Line 5). The first component is the image which displays the icon associated with the severity of the alarm (Line 6). The second component is the label that displays the message of the alarm (Line 7). The last component is the column of buttons which allows the user to either proceed, acknowledge or clear the alarm.

Lines 15-19 specify additional behaviour of this component. This additional behaviour consists in modifying the `itemStyle` variable using a switch statement. Depending on the values of the `proceeding` and `acknowledged` boolean output by the abstract UI, the style applied to the component can take various values. This style is then used (Line 5) in order to style the background and the text colour of the component. Indeed, alarms are highlighted if they are being proceeded, displayed normally if the are not being proceeded nor acknowledged, and faded if they are acknowledged.

```
1  interaction
2    (concrete alarm item (abstract:AUIAlarmItem)):WimpUi
3  is
4  (
5    (row layout with (itemStyle) containing
6      (image displaying ((abstract).icon))
7      (label for ((abstract).message))
8      (column layout with (styleTransparent) containing
9        (standard button ((abstract).proceed))
10       (standard button ((acknowledge).proceed))
11       (standard button ((clear).proceed))
12     )
13   )
14   with behaviour
15   ( switch (((abstract).proceeding),((abstract).acknowledged)) : (itemStyle)
16     case   (       (true)          ,       (false)           ) : (lightGrayRaised)
17     case   (       (false)         ,       (false)           ) : (grayWhiteText)
18     case   (       (false)         ,       (true)            ) : (grayDarkText)
19     default                                                    : (inactive)
20   )
21 )
```

Listing 6.23: Definition of the concrete alarm item component

6.4.3 Alarm List

Figure 6.10 presents a simplified view of how the concrete Alarm list component maps the Abstract interface of the alarm list (specified in Subsection 6.3.3) to the concrete user interface (represented graphically here).

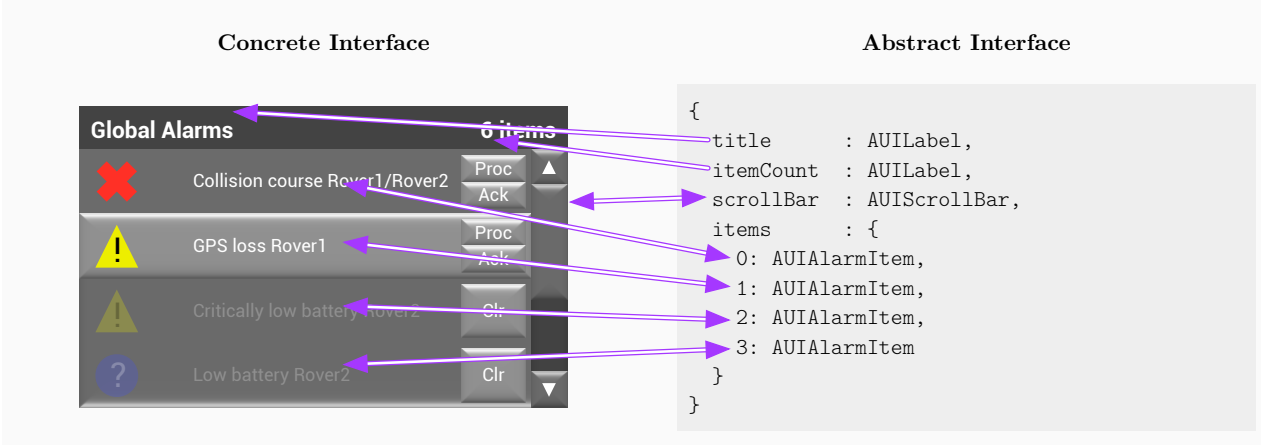


Figure 6.10: Mapping between concrete and abstract interface of a alarm list

Listing 6.24 presents the interaction performed by the concrete alarm item component. Like most concrete interfaces, this interaction is a `WimpUi` associated with an `abstract` UI (Line 2). Lines 4-16 presents the structure of the component. The component is a complex arrangement of `column layouts` and `row layouts` which contain existing widgets. The top container contains the title label and item count label (Lines 6-7). The bottom container contains the alarm list and the scroll bar (Lines 9-14). The alarm list is laid out as a column (Lines 9-13).

```
interaction
  (concrete alarm list (abstract:AUIAlarmItem)):WimpUi
is
  (column layout with (styleNone) containing
    (row layout with (styleWhiteOnTextDarkBG) containing
      (label for ((abstract).title))
      (label for ((abstract).itemCount)) )
    (row layout with (styleNone) containing
      (column layout with (styleDarkBG) containing
        (concrete alarm item ((abstract).items.0))
        (concrete alarm item ((abstract).items.1))
        (concrete alarm item ((abstract).items.2))
        (concrete alarm item ((abstract).items.3)))
      (vertical scrollbar for ((abstract).scrollBar))
    )
  )
```

Listing 6.24: Definition of the concrete alarm list component

6.4.4 Complete UI

Figure 6.11 presents a simplified view of how the concrete alarm management UI component maps the abstract interface (specified in Subsection 6.3.4) to the concrete user interface (represented graphically here).

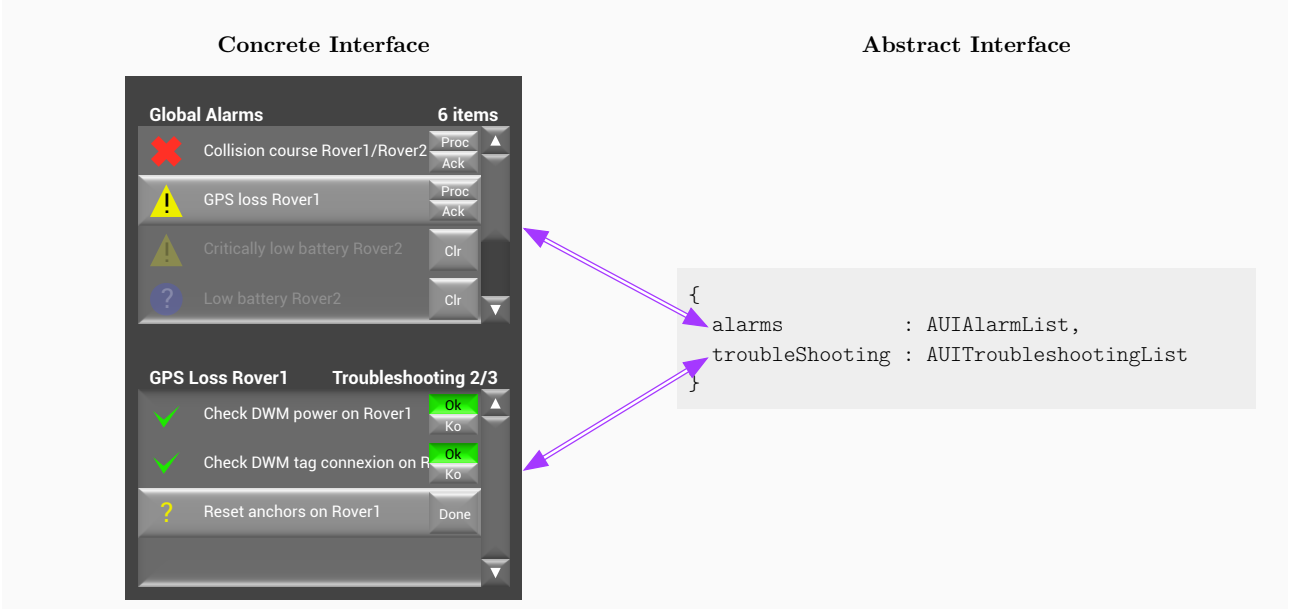


Figure 6.11: Mapping between concrete and abstract interface of the complete UI

Listing 6.25 presents the interaction performed by the concrete alarm management UI. Like most concrete interfaces, this interaction is a `WimpUi` associated with an `abstract` UI (Line 2). The definition of this high-level component is very simple: It is made of two components: the alarm list, and the troubleshooting list, laid out in a column.

```
1 interaction
2   (concrete alarm management (abstract:AUIAlarmManagement)):WimpUi
3 is
4   (column layout with (styleDarkBG) containing
5     (concrete alarm list           ((abstract).alarms))
6     (concrete troubleshooting list ((abstract).troubleShooting))
7   )
```

Listing 6.25: Definition of the concrete alarm list component

6.4.5 Conclusion

Behaviour of the Concrete UI interactor We can now describe the behaviour of the whole "Concrete UI" interactor of our global architecture (Figure 6.2). Listing 6.26 presents the definition of this behaviour. This interaction has two interfaces: `concreteUI` (Line 3), which represents the concrete UI presented to the user, and `abstractUI` (Line 4), which represents the abstract UI to be presented to the user. The behaviour is trivial: the concrete UI presented to the user is the main UI component described in Subsection 6.4.4.

```
1 interaction
2   ( behaviour of the concrete UI interactor
3     where (concreteUI: WimpUi)
4     concretises (abstractUI: AUIAlarmManagement)
5   ):Behaviour
6 is
7   ( (concreteUI) = (concrete alarm management (abstractUI)) )
```

Listing 6.26: Definition of the behaviour of the Abstract UI interactor

Verifying the abstract UI Figure 6.12 presents a geomorphic view where the Model layer, Abstract UI layer and Concrete UI layer are depicted. We just defined the behaviour of the Concrete UI interactor.

We could also define the behaviour of the User interactor in the Abstract UI layer. The description of this interactor would be quite difficult to perform. Indeed, this interactor represents how a Human user would make the link between the concrete UI (an graphic image) and the abstract UI. This would imply modelling complex aspects such as perception of graphical features, text recognition and other complex aspects. It would also imply modelling user interaction using an actual mouse and keyboard, including pointing errors, delays and imprecisions.

If this interactor was modelled, we could perform simulations of the whole end to end architecture of the UI, as presented in Figure 6.12. Listing 6.27 presents the code which would allow such simulation. This code was written by adding Lines 7 and 8 to Listing 6.21. These lines add new intermediates between the user and the UI: instead of communicating directly using the abstract UI, they now communicate through the concrete UI, which adds another layer of complexity between them.

Simulating such a system would allow to answer interesting questions such as "What happens if the User is imprecise in his use of the mouse", "What happens if the User is too slow", "What happens if the UI is less visible because of sun reflexions on the screen". In conclusion, implementing this model would be quite difficult, but the payoff would be very interesting.

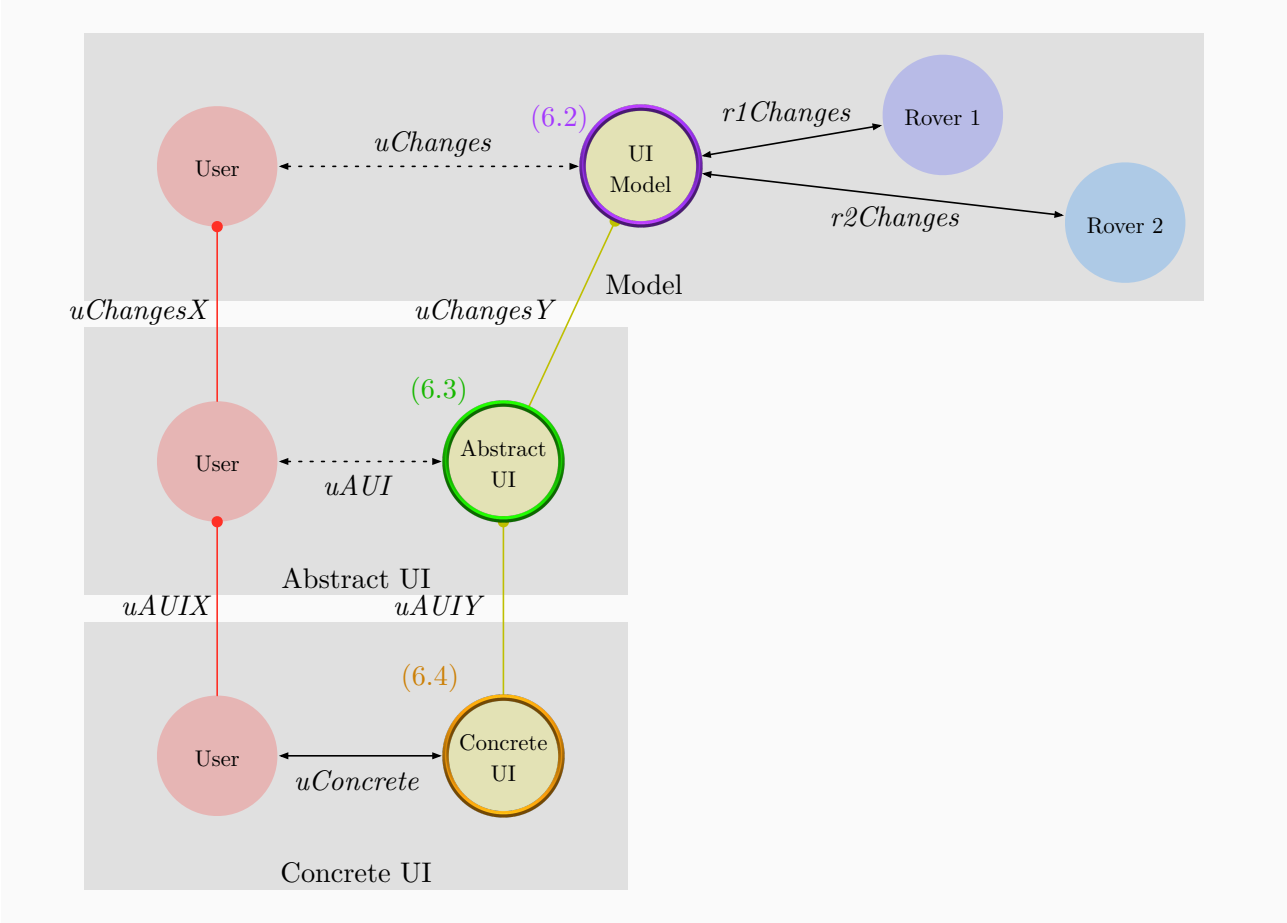


Figure 6.12: A view of the architecture model at this point in the development

```
interaction
  (model layer):Behaviour
is
  (all
    ( behaviour of the user performing (uChangesX) )
    ( behaviour of the user who wants to perform (uChangesX) using (uAUIX))
    ( behaviour of the user who wants to control (uAUIX) using (uConcrete))
    ( behaviour of the concrete UI interactor where (uConcrete) concretises
↪   (uAUIY))
    ( behaviour of the abstract UI interactor where (uAUIY) performs (uChangesY))
    ( behaviour of the UI model interactor between (uChangesY) and (roverChanges))
    ( behaviour of a rover performing (r1Changes) )
    ( behaviour of a rover performing (r2Changes) )
    ((roverChanges) = (compose mutations (r1Changes) (r2Changes)))
  )
```

Listing 6.27: The global interaction between the user and the UI, taking into account 3 layers

6.5 Conclusion

6.5.1 Overview of the model

As stated in Chapter 1 and confirmed in Chapter 2, understanding correctly the architecture of interactive systems is a complex task. In this thesis, we presented several tools whose objective is to alleviate this difficulty. In this case study, we presented an application of these tools on a concrete case.

We started by presenting the global architecture of the model (Subsection 6.1.2). This architecture is decomposed in several interactors which have well-defined roles. These roles are determined according to the geomorphic model (Chapter 3): Each interactor belong to a specific Abstraction layer (Model, abstract UI, concrete UI...), and a specific interactive system (User, UI, Rovers).

We then used LIDL to define the interactions performed by these interactors. Specifically, we chose to detail three interactors in particular: The UI Model interactor (Section 6.2), the Abstract UI interactor (Section 6.3), and the Concrete UI interactor (Section 6.4). The interactions performed by these interactors are complex. As a consequence, for each interactor, we specified a hierarchy of interactions, from basic interactions to complex composite interactions.

This means that the global architecture model is a composition of interactors, where each interactor executes a composition of interactions. Figure 6.13 depicts this structure of the model, where interactors contain compositions of interactions.

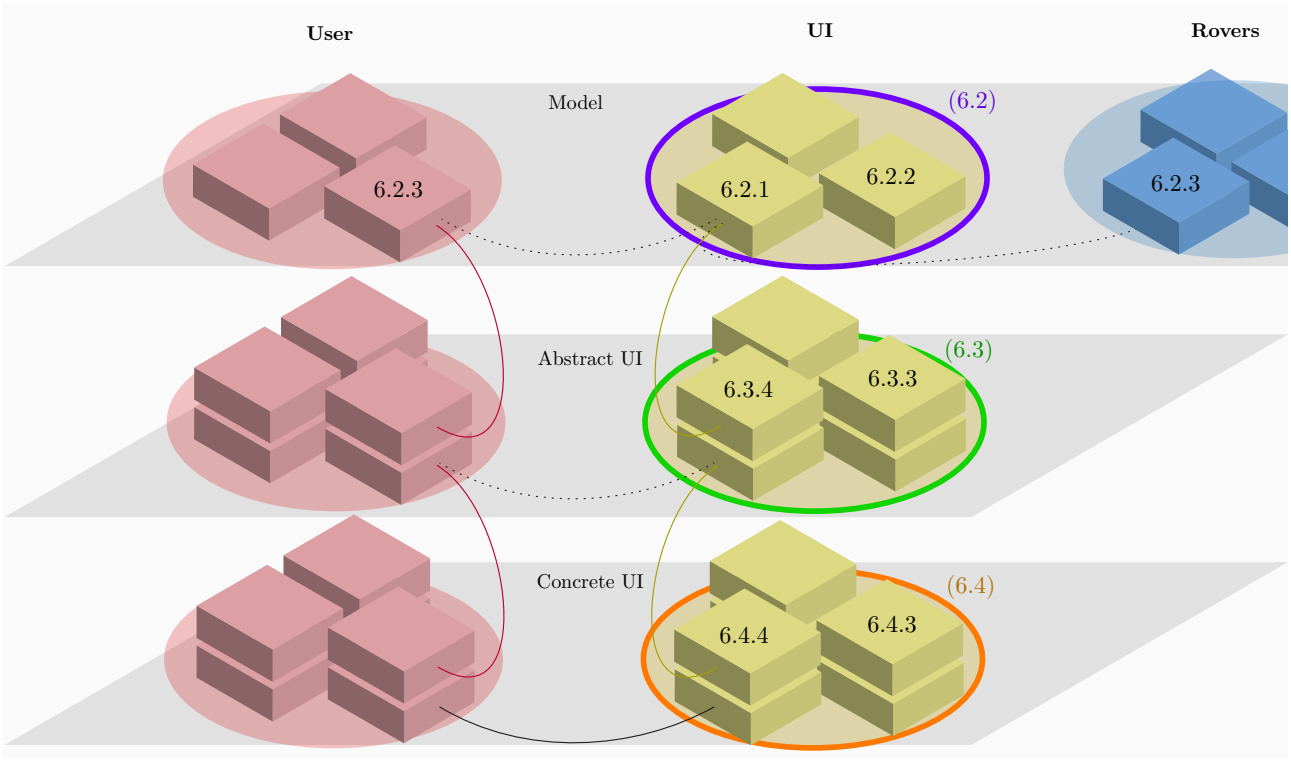
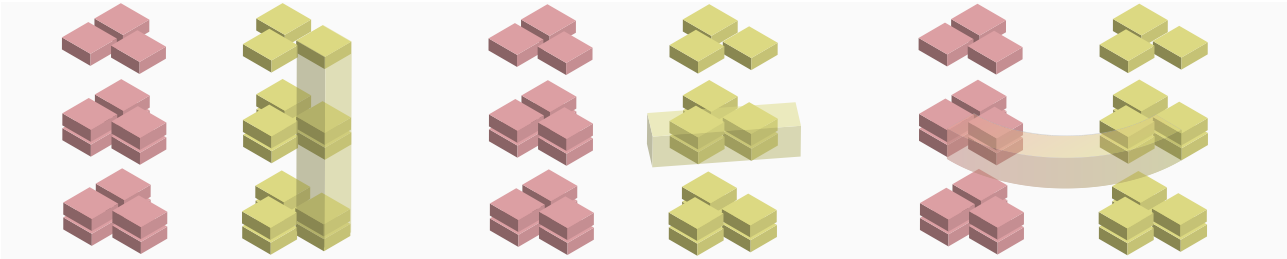


Figure 6.13: Final overview of the architecture of the system

Looking at Figure 6.13, we see that the methods presented in this thesis lead us to create a highly structured model of the interactive system. In particular, this model presents several dimensions where regular patterns can be observed. We can identify three dimensions which are linked with three important aspects of UI systems architecture:



(a) Abstraction Level (b) Composition (c) Sender/Receiver

Figure 6.14: Three dimensions of UI architecture

1. The notion of abstraction layer (As explained in Chapter 3)
2. The notion of composition of interactions (As explained in Chapter 4)
3. The duality between sender and receiver (Of which the duality between human and machine is a special case)

Each interaction we defined in this case study is associated with other elements which are its "symmetries" according to these three dimensions. To illustrate this, let us consider a particular interaction, and see how it is related to other interactions of the model. Consider the Abstract Alarm List interaction, which was defined in Subsection 6.3.3. Figure 6.14 present how this interaction is translated along these three dimensions in the model we developed in this case study. This translations are the following:

1. Translation along the abstraction layer dimension (Figure 6.14a):
   - Upwards, more abstract: The Model object (Subsection 6.2.2)
   - Downwards, less abstract: The Alarm list concrete interface (Subsection 6.4.3)
2. Translation along the composition dimension (Figure 6.14b):
   - Upwards, larger: The Complete Alarm Management abstract interface (Subsection 6.3.4)
   - Downwards, smaller: The Alarm Item abstract interface (Subsection 6.3.2)
3. Symmetry according to the sender/receiver duality (Figure 6.14c):
   - Human side: The task model describing how to use the abstract UI of the alarm item

6.5.2 Assessment of requirements

Using LIDL and the geomorphic model in this case study gives several benefits as compared to more traditional approaches. Table 6.3 presents these benefits, and puts them in perspective with the requirements identified in Chapter 1. This shows that the approaches presented in this thesis do provide significant answers to the requirements.

Requirement	Impact of the solution on the case study
1 Formal	Every interaction defined in the case study denotes a state machine
2 Collaborative	Decomposition of the architecture as a set of independent interactors with clear interfaces allows several teams to work on different parts without interferences
3 Simple	The code is readable as compared to code written in more traditional programming languages. The same paradigm and language is used throughout the case study to describe every aspect of the system, where other approach would implicate a mix of various languages and formalisms.
4 Projection	Interactions defined in the case study denote Directed Acyclic Graph (DAG) that represent the transition function of a state machine, providing a sound starting point for code generation
5 Traceability	All interfaces between components of the system are defined formally, which allows to relate various parts of the specification unambiguously.
6 Composition	Interactions defined in the case study are created by composing interactions, with minimal glue code.
7 Modularity	The specification of reusable interactions allows to change individual parts of the model easily if the need change, without impacting other aspects.
8 Abstraction	The case study uses several useful abstractions such as Mutations, Abstract UI, Task models... All these abstractions are special cases of interactions.
9 Manage	The geomorphic model of the system gives an overview of the global architecture, and abstraction levels are not mixed up.
10 Prototyping	Definition of the concrete UI can start before the abstract UI is completed: it is possible to prototype the appearance of the UI even if not all the high level behaviour is implemented. Conversely, it is possible to define the abstract UI first, and prototype the high level interaction patterns before freezing the design of the concrete appearance of the UI.
11 Verification	The high-level models (UI Model, abstract UI) are prone to model checking (verifying properties that hold on the whole state space), and the more complex models (abstract UI, concrete UI) are prone to simulation (verifying on a subset of the state space)
12 Flexibility	The global architecture model is not fixed, and LIDL allows very different approaches. LIDL does not enforce any particular architecture.

Table 6.3: A synopsis of the requirements and what the case study tells us about them

Chapter 7

Conclusion

If opportunity does not knock, build a door.

Milton Berle

7.1 Overview

In this thesis, we presented the challenges faced during the development of embedded critical Human-Machine Interfaces (HMIs). The main objective of this thesis was then to analyse the problem and find solutions to these challenges. In Chapter 1, after an analysis of the problem, we expressed a set of requirements that should be met by approaches in order for them to offer appropriate answers to the challenges.

Then, in Chapter 2, we examined various approaches that provide partial solutions to the challenges. We deliberately chose to inspect a wide spectrum of approaches, in order to gather as many innovative ideas as possible. We were forced to note that no existing approach provides a fully satisfying answer to all the requirements. However, we remarked that many approaches provide excellent answers to some of the requirements (Table 2.13), and we tried to extract the best ideas of these approaches.

From there, we devised two complimentary approaches, in order to provide a satisfying answer to our requirements.

The first approach is only a conceptual and informal model that allows to describe the global architecture of complex interactive systems. This approach is called the Geomorphic view of interactive systems, and is presented in Chapter 3. It was inspired by several works presented in Chapter 2. We conclude this chapter by explaining how this approach should theoretically meet the requirements.

The second approach is the main contribution of this thesis. It has been developed much more thoroughly in three chapters, and it presents a language called LIDL. This language is based on the paradigm of interaction programming, named so because the goal of this language is to describe interactions between interactive systems.

Basic concepts of LIDL are presented as simply as possible in Chapter 4. Among these basic concepts, we find the notion of composable interface, which is a way to specify the static aspects of interactive systems. Then the concept of interaction is presented. Interactions are a simple concept, but also the most important aspect of LIDL, and what gives it its expressivity. They are a novel concept, even though they can be compared to specific concepts such as destructuring assignments in Javascript,

bidirectional predicates in Prolog, Lustre nodes or Reactive functions in Elm. To my understanding, LIDL interactions are a syntactically simple way to express complex hierarchical Directed Acyclic Graphs of data flow nodes. We conclude this chapter by explaining how LIDL should theoretically meet the requirements.

Implementation of LIDL into a full-featured language with appropriate Integrated Development Environment is a long process. 5 presents work that has been done in this direction. Four building blocks have been developed: A compiler that compile LIDL to JS, a standard library of LIDL interactions, a component that allows to run LIDL programs as HTML Web applications (LIDL Canvas), and an IDE (LIDL Sandbox).

Finally, in Chapter 6, we present a case study which puts to use LIDL and the geomorphic view. This case study is a UI that allows to manage alarms created by a fleet of rovers. We use the geomorphic model to describe the global architecture of this application, and we use LIDL to describe four parts of this architecture, including the task model performed by the human operator. We use the result of this case study to confront LIDL and the geomorphic view with the requirements in a practical setting, and note that the approach provides non-negligible benefits as compared to the state of the art.

7.2 Perspectives

The work presented in this thesis is still at an early experimental stage. Many limitations exist, both in the theoretical ideas and in the practical implementations presented in the thesis. Thankfully, much of these limitations are not insurmountable, and offer perspectives for further improvement.

7.2.1 Geomorphic model

The geomorphic model of architecture of UI is presented in Chapter 3. Its main limitation is its lack of formality, particularly in terms of link with LIDL interactions and interfaces. We used the example of the case study Chapter 6 to presented how LIDL code is related to the architecture model. However, this is only an illustration on a particular example. The general theory that formalises this aspect does not exist yet.

Another limitation of the model is its lack of dynamism. The model does not present any way to represent the creation or destruction of links between interactions. Theses dynamic events happen all the time in real use cases, and are not represented in the geomorphic model, which is a static representation of the structure of systems. However, situations where links between interactors are created and removed can be modelled in combination with LIDL, and especially the notion of Data activation Subsection 4.2.2.

The original idea behind the geomorphic view of interactive systems is the geomorphic view of network systems [49]. This provide interesting perspectives, because the geomorphic view of network system has been formalised and successfully used to verify certain high-level properties of network systems architectures. Further work could lead to a unified model, where interactive systems and network systems are modelled in the same way. This would allow to benefit of the formal tools developed for the geomorphic view of network systems. The verification of high level properties of architectures of interactive systems could be performed.

7.2.2 LIDL

LIDL is a language that is based on several ideas implemented in other languages and tools (Lustre, Elm, React...). Making all these interesting ideas into a consistent and simple core language was a difficult task. At this stage, LIDL still has theoretical limitations, and its implementation (compiler, standard library...) is even more limited. However, the basic idea of LIDL is very simple: to declare interactions, which are composed using interfaces. This simple but well-defined paradigm allows to have a clear vision of the limitations and perspectives of LIDL, in regards to the theoretical *ideal* language.

First, the current implementation of LIDL could be enhanced in several ways:

- The compiler has still to support the whole LIDL language specficiation (especially declaration of reusable interfaces).
- The compiler is slow, it should be possible to increase its speed a great deal.
- The compiler is limited to relatively simple systems. Compilation of complex systems is either too slow or encounters bugs.
- The compiler error messages should be improved a lot in order to provide useful feedback to the programmer.
- The LIDL Sandbox application has great room for improvement in terms of usability.
- The LIDL Sandbox should allow to access elements of the LIDL standard library in an easy way, either through code completion or a graphical list of available interactions.

Finally, the LIDL language definition itself still offers lots of room for improvement. In particular, the following potential improvements have been identified:

- A more complete type system (Arrays, Algebraic data types...). In particular Chapter A presents how algebraic data types could be integrated in LIDL.
- Automatically create interactions based on data type definitions. This point is related with the previous one. When a new data type is added, LIDL should automatically define several interactions related to the data type. We identified four fundamental interactions that would be sufficient to this purpose: construction, matching, mutation, and destruction. This is detailed further in Chapter A.
- Add support for polymorphism. LIDL type system is relatively restricted at the moment. The addition of parametric polymorphism would allow to define generic interactions that are valid for a several data types, reducing a large part of the programmers work.
- Integrate built-in support for documentation. Other languages benefit from tools such as JavaDoc, Doxygen and many others. LIDL does not support any documentation yet. Integration documentation in a meaningful way could greatly improve the experience of programmers.
- Add support for asynchronous function calls. At the moment, function application interactions (4.2.6.5) are synchronous: they must return a value at each instant. However, asynchronous function calls are an important part of UI software engineering. In the future, LIDL should allow to call asynchronous functions, using a built-in language operator, or an addition to the standard library.
- Add other resolving operators (See Subsection 4.2.4). Resolving operators are used to determine the output values of a variable, based on their input values. For example, in a LIDL program when two different signals are sent to a single variable (`x`), an invariant must be verified: at each instant, at most one of the input signal can be *active*, and all the others signals must be *inactive*. At any instant, if more than one signal send actual values to the variable, then an error is raised. This behaviour is the only resolving operator built in LIDL. However, other operators could be used.
- Add support for higher-order interactions, or at least a way to perform interactions on lists. Listing 6.17 is a good example were code is repeated because LIDL does not handle list of interfaces. Chapter A describes a potential solution to this problem.

Chapter A presents several concepts or perspectives that were considered interesting enough to detail

as appendices of this thesis.

Bibliography

[1] *DO178C. Software Consideration in Airborn Systems and Equipment Certification, release c*, RTCA,Inc, 2012.

[2] F. O'Brien, *The Apollo Guidance Computer: Architecture and Operation*. Springer Science & Business Media, 2010.

[3] "Best practices for the design and development of critical information systems", 2008. [Online]. Available: `http://www.prologism.fr/uploads/pdf/CWA_CriticalInformationSystems_Prologism.pdf`.

[4] N. R. Storey, *Safety critical computer systems*. Addison-Wesley Longman Publishing Co., Inc., 1996.

[5] *Specification 661*, Supplement 5, Draft 1, ARINC, Mar. 2012. [Online]. Available: `http://www.aviation-ia.com/aeec/projects/cds/index.html`.

[6] J. Rushby, "Formal methods and critical systems in the real world", 1989.

[7] J. Bowen and V. Stavridou, "Safety-critical systems, formal methods and standards", *Software Engineering Journal*, vol. 8, no. 4, pp. 189–209, 1993.

[8] D. Delmas and J. Souyris, "Astrée: From research to industry", English, in *Static Analysis*, ser. Lecture Notes in Computer Science, H. Nielson and G. Filé, Eds., vol. 4634, Springer Berlin Heidelberg, 2007, pp. 437–451, ISBN: 978-3-540-74060-5. DOI: `10.1007/978-3-540-74061-2_27`. [Online]. Available: `http://dx.doi.org/10.1007/978-3-540-74061-2_27`.

[9] J. Bicarregui, D. Clutterbuck, G. Finnie, H. Haughton, K. Lano, H. Lesan, D. Marsh, B. Matthews, M. R. Moulding, A. R. Newton, *et al.*, "Formal methods into practice: Case studies in the application of the b method", *IEE Proceedings-Software*, vol. 144, no. 2, pp. 119–133, 1997.

[10] J.-R. Abrial, *The B-book: Assigning programs to meanings*. New York, NY, USA: Cambridge University Press, 1996, ISBN: 0-521-49619-5.

[11] D. Bjørner and C. B. Jones, "The vienna development method: The meta-language", *Lecture notes in computer science*, vol. 61, 1978.

[12] J. Nielsen, *Usability 101: Introduction to usability*, 2003.

[13] F.-X. Dormoy, "Scade 6: A model based solution for safety critical software development", in *Proceedings of the 4th European Congress on Embedded Real Time Software (ERTS'08)*, 2008, pp. 1–9.

[14] S. P. Miller, M. W. Whalen, and D. D. Cofer, "Software model checking takes off", *Communications of the ACM*, vol. 53, no. 2, pp. 58–64, 2010.

[15] E. Technologies, *Scade display*, 2011. [Online]. Available: `http://www.esterel-technologies.com/products/scade-display`.

[16] T. Ball and J. Daniel, "Deconstructing dynamic symbolic execution",

[17] A. Hidayat, "Phantomjs", *Computer software. PhantomJS. Vers*, vol. 1, no. 7, 2013.

[18] A. Gimblett and H. Thimbleby, "Applying theorem discovery to automatically find and check usability heuristics", in *Proceedings of the 5th ACM SIGCHI symposium on Engineering interactive computing systems*, ser. EICS '13, London, United Kingdom: ACM, 2013, pp. 101–106, ISBN: 978-1-4503-2138-9. DOI: 10.1145/2480296.2480320. [Online]. Available: http://doi.acm.org/10.1145/2480296.2480320.

[19] G. J. Holzmann, "The model checker spin", *IEEE Trans. Softw. Eng.*, vol. 23, no. 5, pp. 279–295, May 1997, ISSN: 0098-5589. DOI: 10.1109/32.588521. [Online]. Available: http://dx.doi.org/10.1109/32.588521.

[20] J. C. Campos, M. Sousa, M. C. B. Alves, and M. D. Harrison, "Formal verification of a space system's user interface with the ivy workbench", *IEEE TRANSACTIONS ON HUMAN-MACHINE SYSTEMS*, 2015.

[21] E. L. Wiener, *Human factors of advanced technology (" glass cockpit") transport aircraft*. 1989, vol. 177528.

[22] M. L. Bolton, R. Siminiceanu, E. J. Bass, *et al.*, "A systematic approach to model checking human–automation interaction using task analytic models", *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, vol. 41, no. 5, pp. 961–976, 2011.

[23] F. Dehais, C. Tessier, and L. Chaudron, "Ghost: Experimenting countermeasures for conflicts in the pilot's activity", 2003.

[24] S. Combéfis, D. Giannakopoulou, C. Pecheur, and M. Feary, "A formal framework for design and analysis of human-machine interaction", in *Systems, Man, and Cybernetics (SMC), 2011 IEEE International Conference on*, IEEE, 2011, pp. 1801–1808.

[25] T. Mens, M. Claes, P. Grosjean, and A. Serebrenik, *Studying evolving software ecosystems based on ecological models*. Springer, 2014.

[26] L. Bass, P. Clements, and R. Kazman, "Software architecture in practice", 1997.

[27] M. W. Maier, D. Emery, and R. Hilliard, "Software architecture: Introducing ieee standard 1471", *Computer*, vol. 34, no. 4, pp. 107–109, 2001.

[28] J.-D. Fekete, "Un modele multicouche pour la construction d'applications graphiques interactives", 1996PA112010, PhD thesis, 1996, 225 P. [Online]. Available: http://www.theses.fr/1996PA112010.

[29] L. Bass, R. Little, R. Pellegrino, S. Reed, R. Seacord, S. Shepard, and M. Szezur, "The arch model: Seeheim revisited", in *Proceedings of the 1991 User Interface Developers' Workshop*, Seeheim: ACM, 1991.

[30] T. M. H. Reenskaug, "The original mvc reports", 1979.

[31] G. E. Krasner, S. T. Pope, *et al.*, "A description of the model-view-controller user interface paradigm in the smalltalk-80 system", *Journal of object oriented programming*, vol. 1, no. 3, pp. 26–49, 1988.

[32] J. Coutaz, "Pac, an object oriented model for dialog design", 1987.

[33] ——, "Software architecture modeling for user interfaces", *Encyclopedia of software engineering*, 1993.

[34] L. Nigay and J. COUTAZ, *Building user interfaces: Organizing software agents*, 1991.

[35] T. Duval and L. Nigay, "Implémentation d'une application de simulation selon le modèle pac-amodeus", *IHM'99*, pp. 86–93, 1999.

[36] M. Potel, "Mvp: Model-view-presenter the taligent programming model for c++ and java", *Taligent Inc*, 1996.

[37] "Mvc as a compound design pattern", Apple, Inc, Tech. Rep. [Online]. Available: https://developer.apple.com/library/mac/documentation/General/Conceptual/CocoaEncyclopedia/CocoaEncyclopedia.pdf.

[38] A. Fowler, *A swing architecture overview*, 2005. [Online]. Available: http://www.oracle.com/technetwork/java/architecture-142923.html (visited on 2015).

[39] A. Cortier, "Contribution à la validation formelle de systèmes interactifs java", PhD thesis, Universite Paul Sabatier - Institut Supérieur de l'Aéronautique et de l'Espace (ISAE-Supaero), 2008. [Online]. Available: http://alexandre.cortier.free.fr.

[40] M. Fowler, *Presentation model*, 2004. [Online]. Available: http://martinfowler.com/eaaDev/PresentationModel.html.

[41] J. Gossman, *Introduction to model/view/viewmodel pattern for building wpf apps*, 2005. [Online]. Available: https://msdn.microsoft.com/en-us/magazine/dd419663.aspx.

[42] *Qt*. [Online]. Available: http://www.qt.io.

[43] *The flux architecture*, Facebook, 2014. [Online]. Available: https://facebook.github.io/flux/.

[44] *React native, a framework for building native apps using react*. [Online]. Available: https://facebook.github.io/react-native/.

[45] C. Bachman, "Provisional model of open-systems architecture", *SIGCOMM Computer Communication Review*, vol. 8, no. 3, pp. 49–61, Jul. 1978, ISSN: 0146-4833. DOI: 10.1145/1015850.1015854. [Online]. Available: http://doi.acm.org/10.1145/1015850.1015854.

[46] H. Zimmermann, "Osi reference model–the iso model of architecture for open systems interconnection", *Communications, IEEE Transactions on*, vol. 28, no. 4, pp. 425–432, 1980.

[47] M. Handley, "Why the internet only just works", *BT Technology Journal*, vol. 24, no. 3, pp. 119–129, 2006.

[48] T. Roscoe, "The end of internet architecture", in *Proceedings of the 5th Workshop on Hot Topics in Networks*, 2006.

[49] P. Zave and J. Rexford, "The geomorphic view of networking: A network model and its uses", in *Proceedings of the 7th Workshop on Middleware for Next Generation Internet Computing*, ser. MW4NG '12, Montreal, Quebec, Canada: ACM, 2012, 1:1–1:6, ISBN: 978-1-4503-1607-1. DOI: 10.1145/2405178.2405179. [Online]. Available: http://doi.acm.org/10.1145/2405178.2405179.

[50] G. P. Faconti and F. Paternò, "An approach to the formal specification of the components of an interaction", 1990.

[51] F. Paterno', "A theory of user-interaction objects", *Journal of Visual Languages Computing*, vol. 5, no. 3, pp. 227–249, 1994, ISSN: 1045-926X. DOI: http://dx.doi.org/10.1006/jvlc.1994.1012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1045926X84710123.

[52] D. Duke and M. Harrison, "Abstract interaction objects", *Computer Graphics Forum*, vol. 12, no. 3, pp. 25–36, 1993, ISSN: 1467-8659. DOI: 10.1111/1467-8659.1230025. [Online]. Available: http://dx.doi.org/10.1111/1467-8659.1230025.

[53] J. M. Spivey and J. Abrial, *The Z notation*. Prentice Hall Hemel Hempstead, 1992.

[54] D. Duke and M. Harrison, "Event model of human-system interaction", *Software Engineering Journal*, vol. 10, no. 1, pp. 3–12, 1995.

[55] D. Duke, G. Faconti, M. Harrison, and F. Paternó, "Unifying views of interactors", in *Proceedings of the workshop on Advanced visual interfaces*, ACM, 1994, pp. 143–152.

[56] P. Roche, "Modélisation et validation d'interface homme-machine", PhD thesis, Ecole nationale supérieure de l'aéronautique et de l'espace, 1998.

[57] D. Pilaud, N. Halbwachs, and J. Plaice, "Lustre: A declarative language for programming synchronous systems", in *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages (14th POPL 1987). ACM, New York, NY*, vol. 178, 1987, p. 188.

[58] J. I. Hsia, E. Simpson, D. Smith, and R. Cartwright, "Taming java for the classroom", in *ACM SIGCSE Bulletin*, ACM, vol. 37, 2005, pp. 327–331.

[59] J. Brooks F.P., "No silver bullet essence and accidents of software engineering", *Computer*, vol. 20, no. 4, pp. 10–19, Apr. 1987, ISSN: 0018-9162. DOI: 10.1109/MC.1987.1663532.

[60] S. Blackheath and A. Jones, *Functional reactive programming*. Manning, 2014.

[61] M. Fowler, *Domain-specific languages*. Pearson Education, 2010.

[62] B. Meyers, R. Deshayes, L. Lucio, E. Syriani, H. Vangheluwe, and M. Wimmer, "Promobox: A framework for generating domain-specific property languages", English, in *Software Language Engineering*, ser. Lecture Notes in Computer Science, B. Combemale, D. Pearce, O. Barais, and J. Vinju, Eds., vol. 8706, Springer International Publishing, 2014, pp. 1–20, ISBN: 978-3-319-11244-2. DOI: 10.1007/978-3-319-11245-9_1. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-11245-9_1.

[63] *Qml*. [Online]. Available: http://doc.qt.io/qt-5/qmlapplications.html.

[64] *The json data interchange format*, ECMA, Oct. 2013. [Online]. Available: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf.

[65] *Java fx*. [Online]. Available: http://docs.oracle.com/javase/8/javafx/get-started-tutorial/jfx-overview.htm#JFXST784.

[66] *Scala fx*. [Online]. Available: http://www.scalafx.org.

[67] *Groovy fx*. [Online]. Available: http://groovyfx.org.

[68] *Clojure fx*. [Online]. Available: https://bitbucket.org/zilti/clojurefx.

[69] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, "Extensible markup language (xml)", *World Wide Web Consortium Recommendation REC-xml-19980210. http://www.w3. org/TR/1998/REC-xml-19980210*, vol. 16, 1998.

[70] M. Abrams, C. Phanouriou, A. L. Batongbacal, S. M. Williams, and J. E. Shuster, "Uiml: An appliance-independent xml user interface language", *Computer Networks*, vol. 31, no. 11, pp. 1695–1708, 1999.

[71] Q. Limbourg, J. Vanderdonckt, B. Michotte, L. Bouillon, and V. López-Jaquero, "Usixml: A language supporting multi-path development of user interfaces", *EHCI/DS-VIS*, vol. 3425, pp. 200–220, 2004.

[72] L. MacVittie, *XAML in a Nutshell*. " O'Reilly Media, Inc.", 2006.

[73] D. Hyatt, B. Goodger, I. Hickson, and C. Waterson, "Xml user interface language (xul) 1.0", *Mozilla. org*, 2001.

[74] C. Coenraets, "An overview of mxml: The flex markup language", *Adobe Systems. March*, 2004.

[75] Q. Limbourg, J. Vanderdonckt, B. Michotte, L. Bouillon, M. Florins, and D. Trevisan, "Usixml: A user interface description language for context-sensitive user interfaces", in *Proceedings of the ACM AVI'2004 Workshop" Developing User Interfaces with XML: Advances on User Interface Description Languages*, 2004, pp. 55–62.

[76] *W3c document object model (dom)*. [Online]. Available: http://www.w3.org/DOM/.

[77] *Jquery*. [Online]. Available: https://jquery.com.

[78] T. J. Parr, "Enforcing strict model-view separation in template engines", in *Proceedings of the 13th international conference on World Wide Web*, ACM, 2004, pp. 224–233.

[79] *Angular js*. [Online]. Available: https://angularjs.org.

[80] *React - a javascript library for building user interfaces*, Facebook, 2013. [Online]. Available: http://facebook.github.io/react/.

[81] A. Courtney and C. Elliott, "Genuinely functional user interfaces", in *Haskell workshop*, 2001, pp. 41–69.

[82] Z. Wan and P. Hudak, "Functional reactive programming from first principles", in *ACM SIGPLAN Notices*, ACM, vol. 35, 2000, pp. 242–252.

[83] S. L. P. Jones, *Haskell 98 language and libraries: The revised report*. Cambridge University Press, 2003.

[84] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler, "A history of haskell: Being lazy with class", in *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, ACM, 2007, pp. 12–1.

[85] H. Apfelmus, *Reactive-banana*, 2012. [Online]. Available: http://www.haskell.org/haskellwiki/Reactive-banana.

[86] E. Czaplicki, "Elm: Concurrent frp for functional guis", PhD thesis, 2012.

[87] ——, *The elm architecture*. [Online]. Available: https://github.com/evancz/elm-architecture-tutorial/.

[88] P. Palanque and R. Bastide, "Interactive cooperative objects : An object-oriented formalism based on petri nets for user interface design", in *Proceedings of the IEEE Conference on System Man and Cybernetics*, Elsevier Science Publisher, Oct. 1993.

[89] A. Hamon-Keromen, "Définition d'un langage et d'une méthode pour la description et la spécification d'ihm post-wimp pour les cockpits interactifs", PhD thesis, Université de Toulouse, Université Toulouse III-Paul Sabatier, 2014.

[90] D. Navarre, P. Palanque, J.-F. Ladry, and E. Barboni, "Icos: A model-based user interface description technique dedicated to interactive systems addressing usability, reliability and scalability", *ACM Trans. Comput.-Hum. Interact.*, vol. 16, 18:1–18:56, 4 Nov. 2009, ISSN: 1073-0516. DOI: http://doi.acm.org/10.1145/1614390.1614393. [Online]. Available: http://doi.acm.org/10.1145/1614390.1614393.

[91] B. Weyers, J. Bowen, A. Dix, and P. Palanque, "Workshop on formal methods in human computer interaction", in *Proceedings of the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, ACM, 2015, pp. 294–295.

[92] *Max/msp user manual*, Cycling '74. [Online]. Available: https://docs.cycling74.com/max7.

[93] S. Oschatz, *Feedback loops in vvvv*, 2010. [Online]. Available: http://vvvv.org/documentation/creating-feedback-loops (visited on 2010).

[94] *Storyboards user manual*, Apple Incorporated. [Online]. Available: https://developer.apple.com/library/ios/documentation/General/Conceptual/Devpedia-CocoaApp/Storyboard.html.

[95] F. Paternò, C. Mancini, and S. Meniconi, "Concurtasktrees: A diagrammatic notation for specifying task models", in *Human-Computer Interaction INTERACT'97*, Springer, 1997, pp. 362–369.

[96] F. Racim, M. Célia, and P. Philippe, "Hamsters: Un environnement d'édition et de simulation de modèles de tâches", in *IHM'14, 26e conférence francophone sur l'Interaction Homme-Machine*, 2014, pp. 5–6.

[97] F. Jourde, Y. Laurillau, and L. Nigay, "E-comm, un éditeur pour spécifier l'interaction multimodale et multiutilisateur", in *Conference Internationale Francophone sur I'Interaction Homme-Machine*, ACM, 2010, pp. 225–228.

[98] M. L. Bolton and E. J. Bass, "Enhanced operator function model: A generic human task behavior modeling language", in *Systems, Man and Cybernetics, 2009. SMC 2009. IEEE International Conference on*, IEEE, 2009, pp. 2904–2911.

[99] ——, "Model checking human-automation interaction with enhanced operator function model", *Human-Machine Interaction (Formal H)*, p. 34, 2012.

[100] M. Bolton and E. Bass, "Evaluating human-automation interaction using task analytic behavior models, strategic knowledge-based erroneous human behavior generation, and model checking", in *Systems, Man, and Cybernetics (SMC), 2011 IEEE International Conference on*, Oct. 2011, pp. 1788–1794. DOI: 10.1109/ICSMC.2011.6083931.

[101] C. A. Petri, "Fundamentals of a theory of asynchronous information flow", in *Proceedings of IFIP Congress*, München, Germany, 1962, pp. 386–390.

[102] W. M. Van der Aalst, "The application of petri nets to workflow management", *Journal of circuits, systems, and computers*, vol. 8, no. 01, pp. 21–66, 1998.

[103] F. de Rosis, S. Pizzutilo, and B. De Carolis, "Formal description and evaluation of user-adapted interfaces", *International Journal of Human-Computer Studies*, vol. 49, no. 2, pp. 95–120, 1998.

[104] H. Ezzedine and C. Kolski, "Modelling of cognitive activity during normal and abnormal situations using object petri nets, application to a supervision system", *Cognition, Technology & Work*, vol. 7, no. 3, pp. 167–181, 2005.

[105] C. Sibertin-Blanc, "High level petri nets with data structure", in *6th european workshop on Application and Theory of Petri Nets*, 1985, pp. 141–168.

[106] B. d'Ausbourg, "Rapport de synthèse en vue de candidater à l'habilitation à diriger des recherches.", ONERA, Tech. Rep., 2001.

[107] ——, *Using model checking for the automatic validation of user interfaces systems*. Springer, 1998.

[108] I. Sessitskaia, "Apport des techniques d'abstraction pour la véri cation des interfaces homme-machine", PhD thesis, PhD thesis, ONERA, Toulouse, France, 2002.

[109] J. C. Campos, G. Doherty, and M. D. Harrison, "Analysing interactive devices based on information resource constraints", *International Journal of Human-Computer Studies*, vol. 72, pp. 284–297, 2014. DOI: 10.1016/j.ijhcs.2013.10.005.

[110] J. C. Campos and M. D. Harrison, "Modelling and analysing the interactive behaviour of an infusion pump", in *Proceedings of the Fourth International Workshop on Formal Methods for Interactive Systems (FMIS 2011)*, 2011.

[111] M. Ryan, J. Fiadeiro, and T. Maibaum, "Sharing actions and attributes in modal action logic", in *Theoretical Aspects of Computer Software*, Springer, 1991, pp. 569–593.

[112] J. C. Campos and M. D. Harrison, "Model checking interactor specifications", *Automated Software Engineering*, vol. 8, no. 3-4, pp. 275–310, 2001.

[113] J. C. Campos, D. Gonçalves, N. Guerreiro, S. Mendes, V. Pinheiro, J. C. Campos, *et al.*, "Animal-a user interface prototyper and animator for mal interactor models", in *Interac-International Conference on Human-Computer Interaction*, Grupo Português de Computação Gráfica, 2008.

[114] A. Gimblett, "Structural usability techniques for dependable HCI", PhD thesis, Swansea University, 2014.

[115] Y. Aït-Ameur, I. Aït-Sadoune, and M. Baron, "Modélisation et validation formelles d'ihm: Lot 1 (lisi/ensma)", *Délivrable pour le projet RNRT-VERBATIM*, p. 73, 2005.

[116] J.-R. Abrial, *Modeling in Event-B: System and software engineering*. Cambridge University Press, 2010.

[117] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-oriented software architecture, volume 1: A system of patterns*, 1996.

[118] K. Simonyan and B. Horwitz, "Laryngeal motor cortex and control of speech in humans", *The Neuroscientist*, p. 1 073 858 410 386 727, 2011.

[119] A. P. Conn, "Time affordances: The time factor in diagnostic usability heuristics", in *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM Press/Addison-Wesley Publishing Co., 1995, pp. 186–193.

[120] J. Nielsen, "Usability inspection methods", in *Conference companion on Human factors in computing systems*, ACM, 1994, pp. 413–414.

[121] R. T. Fielding, "Architectural styles and the design of network-based software architectures", PhD thesis, University of California, Irvine, 2000.

[122] G. H. Mealy, "A method for synthesizing sequential circuits", *Bell System Technical Journal*, vol. 34, no. 5, pp. 1045–1079, 1955.

[123] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous dataflow programming language lustre", in *Proceedings of IEEE*, ser. 79, Sep. 1991, pp. 1305–1320.

[124] A. Agrawal, G. Karsai, and F. Shi, "Graph transformations on domain-specific models",

[125] R. Tarjan, "Edge-disjoint spanning trees and depth-first search", English, *Acta Informatica*, vol. 6, no. 2, pp. 171–185, 1976, ISSN: 0001-5903. DOI: 10.1007/BF00268499. [Online]. Available: http://dx.doi.org/10.1007/BF00268499.

[126] L. Hunt, *Html5 reference, the syntax, vocabulary and apis of html5*, 2010. (visited on 2010).

[127] *Html canvas 2d context*. [Online]. Available: https://www.w3.org/TR/2dcontext/ (visited on 11/19/2015).

[128] *Node package manager*. [Online]. Available: https://www.npmjs.com.

[129] E. Meijer, M. Fokkinga, and R. Paterson, "Functional programming with bananas, lenses, envelopes and barbed wire", in *Functional Programming Languages and Computer Architecture*, Springer, 1991, pp. 124–144.

Part III

Appendix

Appendix A

Emerging concepts in LIDL

Computer science is no more about computers than astronomy is about telescopes

Edsger Wybe Dijkstra

This chapter presents several interesting concepts and reflexions that are part of LIDL at the present day, or that could provide powerful extensions to the language.

A.1 Interaction hierarchy rotation

Behaviours are the canonic form of LIDL interactions. Thanks to the flexibility of LIDL, any interaction can be put in the form of a behaviour. For example, Listing A.1 shows how the `(sin(x))` interaction (whose interface is `Number out`) can be made into the `((y)=sin(x))` interaction (whose interface is `Behaviour`).

```
1 interaction
2   (sin (x:Number in)):Number out
3 is
4   ...
5
6 interaction
7   ((y:Number out)=sin(x:Number in)):Behaviour
8 is
9   ( (y) = (sin(x)) )
```

Listing A.1: Definition of a behaviour associated with the `(sin(x))` interaction

As a matter of fact, most basic interactions built in LIDL itself (Function application, previous, affectation...) are `Behaviours` (See Subsection 4.2.6).

However, behaviours are not always a convenient way to compose interactions. Depending on the aspects being modelled, various composition operators are preferable. For example, when describing numeric computations, the `Number out` interface is much more convenient than the `Behaviour` interface, since it allows to compose mathematical expressions naturally. Listing A.2 presents two expressions describing the same computation: The first one (Line 1) uses the `Number out` interface,

while the second one (Lines 3-7) uses the `Behaviour` interface, which is not ideal for the description of mathematical expressions, and ends up in much longer code.

```
1 ( (c) = ( sin(( (a) + (b) ) / (2)) )
2
3 (all
4   ( (x) = (a) + (b) )
5   ( (y) = (x) / (2) )
6   ( (c) = sin (y) )
7 )
```

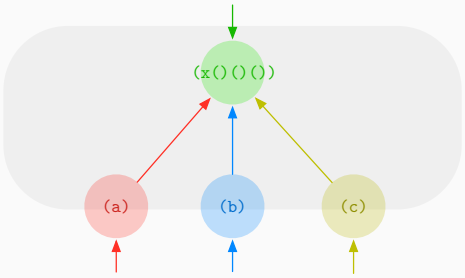
Listing A.2: Two expressions describing the same computation.

As we see, choosing the right interface, the interface that will allow to conveniently express the composition of interactions of a certain nature, is a very important aspect of LIDL programming, since it has a large impact on the syntactic simplicity of programs. An important part of the design of the LIDL standard library (Section 5.2) is an adequate choice of interfaces, which allows to conveniently compose concepts such as Concrete and Abstract Widgets, Tasks, or Data of certain types.

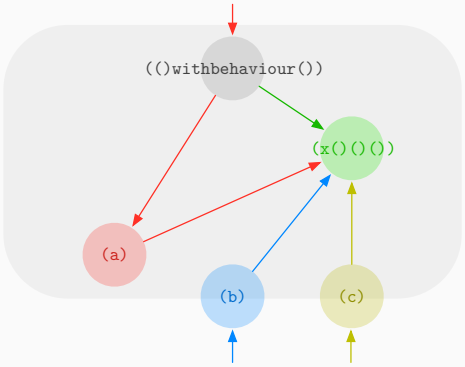
Figure A.1 presents a general example that describes this concept of *Interaction hierarchy rotation*. An interaction is first expressed in terms of a Behaviour with 3 arguments, and then presented in three other forms, one for each argument of the `Behaviour`. These four interactions perform the same thing, but syntactically, they will be very different to use.

Finally, the choice of the form to use to describe a particular interaction is sometimes a matter of taste or arbitrary orientation by a LIDL library designer. For example let us consider the case of a "button to increment a value". Depending on the programmer perspective and experience, this interaction can be seen in at least 3 different ways. Listing A.3 presents the LIDL code associated with these three design options:

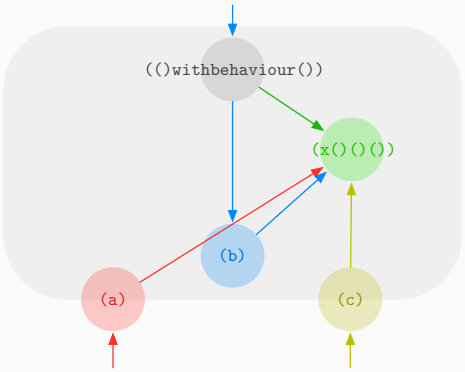
1. A `WimpUi` Widget representing a button, which happens to increment a number when clicked (Lines 1-3). This orientation is very convenient when the main focus is to describe the composition of a UI as a hierarchy of widgets.
2. A `Behaviour` performed by the system. The behaviour consists in displaying a button widget to the user, and incrementing a value when the button is clicked (Lines 5-8). This orientation is very convenient when the main focus is to describe the behaviour of an interactor, in a way similar to imperative programming.
3. A data operator acting on a `Number`. This operator consists in conditionally incrementing the number or not, depending on whether the button is clicked or not (Lines 10-12). This orientation is very convenient when the main focus is to describe the data flow through a program.



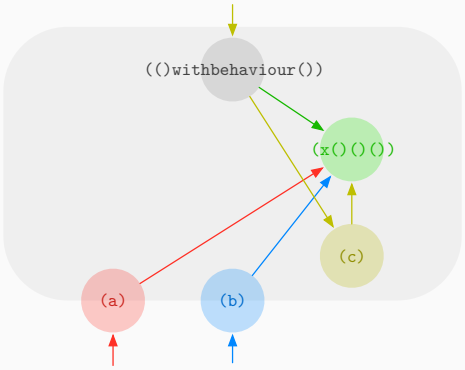

```
interaction
  (x (a:A) (b:B) (c:C)): Behaviour
```




```
interaction
  (xA (b:B) (c:C)):A
is
  (
    (a)
    with behaviour
    (x (a) (b) (c))
  )
```




```
interaction
  (xB (a:A) (c:C)):B
is
  (
    (b)
    with behaviour
    (x (a) (b) (c))
  )
```




```
interaction
  (xC (a:A) (b:B)):C
is
  (
    (c)
    with behaviour
    (x (a) (b) (c))
  )
```

Figure A.1: A behaviour with three arguments, and its three rotations

```
interaction
  ( button that increments (x: Number in)
    and send the result to (y: Number out) ): WimpUi

interaction
  ( display a button       (b:WimpUi)
    that increments        (x: Number in)
    and send the result to (y: Number out) ): Behaviour

interaction
  ( (x:Number in)
    incremented when the button (b:WimpUi) is clicked ): Number out
```

Listing A.3: Three valid ways to see a "button to increment a value"

A.2 Variables resolving operators

Variables (See 4.2.6.2) allow to send and receive data across LIDL programs, between different branches of the interaction composition tree. Without variables, the data flow in LIDL programs would be restricted to vertices of the interaction tree, but variables allow to create convenient "bridges" between branches.

Each variable can be used in two dual contexts: sending values to variables, or receiving values from variables. For example, in `((x)=(5))`, the value 5 is sent to the variable `(x)`, and in `(label displaying (x))`, the value is received from the variable `(x)` in order to be displayed. Each variable can be used several times in both contexts (send and receive) within a LIDL program. This leads to a question: what is the value of a variable when several interactions set its value ?

For example, consider the code of Listing A.4. This program consists of four behaviours that are executed continuously in parallel, since LIDL is a synchronous language. This program involves a variable called `(x)`. `(x)` can receive two different values, `(a)` or `(b)`, depending on the values of `(condition 1)` and `(condition 2)` (Lines 2-3). The value of `(x)` is sent to `(y)` and `(z)` (Lines 4-5).

```
1 (all
2   (when (condition 1) then ((x)=(a)) )
3   (when (condition 2) then ((x)=(b)) )
4   ((y)=(x))
5   ((z)=(x))
6 )
```

Listing A.4: A simple LIDL program involving the variable `(x)`

Figure A.2 presents the same LIDL program graphically, as a composition tree, where each node is an interaction. The figure also represent the direction of the data flow between each node. We see that `(x)` is used twice in both contexts (send and receive). This leads to a situation where a "resolving operator" is used.

The resolving operator formally defines the semantics of LIDL programs where variables are written or read several times. A resolving operator is defined by a single function, whose input is the list of values received by the variable at each step, and whose output is the list of values sent by the variable at each step. In our example, the resolving operator is defined as a function $resolve : (in_1, in_2) \mapsto (out_1, out_2)$.

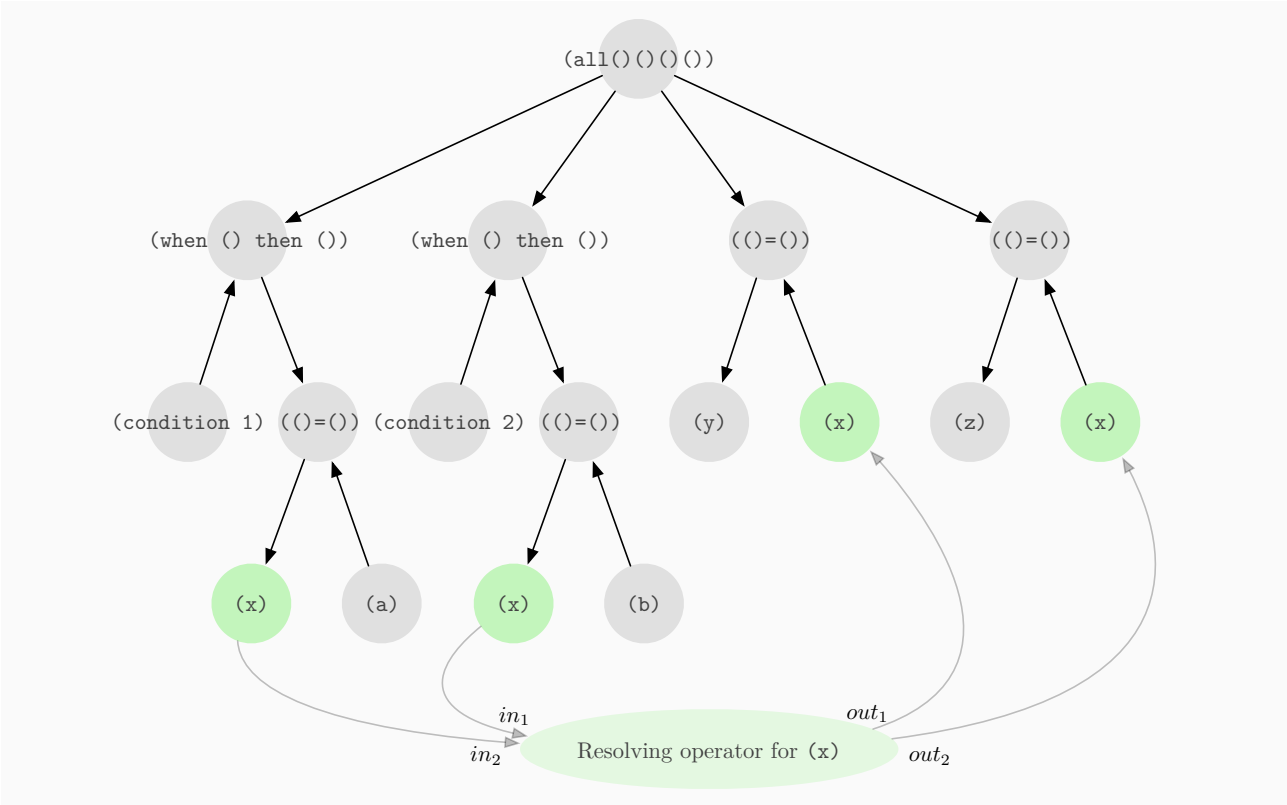


Figure A.2: The interaction composition tree and data flow of Listing A.4

The current definition of LIDL supports only one particular resolving operator, which is used by default for every variable in every LIDL program. This operator is relatively straightforward and relies on the notion of data activation. Informally, the resolving operator behaves by following three simple cases: If all the input values are *inactive* then all the output values are *inactive*, If one and only one input value is not *inactive* then all the output values are equal to this value, and if more than one input values are not *inactive* then an error is raised. Formally, the definition of the resolving operator is the following:

$$\forall i, out_i = \begin{cases} inactive & \text{if } \forall j, in_j = inactive \\ in_j & \text{if } \exists! j, in_j \neq inactive \\ error & \text{otherwise} \end{cases}$$

Table A.1 shows an example execution of the program presented in Listing A.4. We note that `(y)` and `(z)` can take their values either from `(a)` or from `(b)` depending on conditions, in accordance with the resolving function presented above.

LIDL supports only one resolving operator at the moment. However, many other interesting resolving operators can come to mind. A development perspective for LIDL would be to offer a way to specify and implement other resolving operators, such as:

- *appendAllToList*: Several values can be sent to a variable, and the variable output is a list containing all the values received by the variable:

$$\forall i, out_i = (in_0, in_1, ...in_j)$$

- *someMustBeActiveAndAgree*: Several values can be sent to a variable, if several of them are active

Instant	(a)	(condition1)	(b)	(condition2)	(y)	(z)
0	1	*active*	10	*inactive*	1	1
1	2	*active*	10	*inactive*	2	2
2	2	*inactive*	10	*inactive*	*inactive*	*inactive*
3	2	*inactive*	10	*active*	10	10
4	2	*inactive*	11	*active*	11	11
5	2	*inactive*	*inactive*	*active*	*inactive*	*inactive*
6	2	*active*	*inactive*	*active*	2	2

Table A.1: Timing diagram of an execution of the program of Listing A.4

and they are equal, then their common value is output:

$$\forall i, out_i = \begin{cases} inactive & \text{if } \forall j, in_j = inactive \\ v & \text{if } \exists! v, \forall j, in_j \neq inactive \implies in_j = v \\ error & \text{otherwise} \end{cases}$$

- *allMustBeActiveAndAgree*: All input values must be active and agree on the same value:

$$\forall i, out_i = \begin{cases} v & \text{if } \exists! v, \forall j, in_j = v \\ inactive & \text{otherwise} \end{cases}$$

- *onlyOneActiveDefaultToPrevious*: If all input are inactive, then the variable defaults to its previous value instead of *inactive*:

$$\forall i, out_i = \begin{cases} previous(out_i) & \text{if } \forall j, in_j = inactive \\ in_j & \text{if } \exists! j, in_j \neq inactive \\ error & \text{otherwise} \end{cases}$$

A.3 Generics

A.4 Data type interactions

Many software systems offer four kinds of fundamental operations on data. These operations are often called the Create, Read, Update, Delete (CRUD) operations. During the development of LIDL, we found that for each data type, four kind of interactions allow to conveniently express most useful interactions involving this data type. As a consequence, we believe that a way forward for LIDL would be to create these interactions automatically whenever a data type is defined. For example, consider the following algebraic data type:

Each data type is associated with 4 kinds of interactions, the following list present similar concept of other paradigms: Databases, Object-Oriented Programming (OOP) and FP.

- Construction: Create (Databases), Constructor (OOP), Constructor (FP)
- Comparison: Read (Databases), Getter (OOP), Pattern matching (FP)
- Mutation: Update (Databases), Setter (OOP), Setter lens (FP)

```
data
  MyType
is
  (foo (x: Number) (y: Number)) | (bar)
```

Listing A.5

- Destruction: Delete (Databases), Destructor (OOP),

These four kinds of interactions are the LIDL equivalent of the four CRUD operations.

A.4.1 Construction

This data type create several interactions:

```
interface Construction X is X out

interaction (foo (x: Number in) (y: Number in)): Construction Toto
interaction (bar): Construction Toto
```

Listing A.6

A.4.2 Comparison

```
interface Comparison X is {value: X in, result: Boolean out}

interaction (foo (x: Comparison Number) (y:Comparison Number)): Comparison Toto
interaction (bar): Comparison Toto
interaction (is (t: Toto in)): Comparison Toto
interaction (to (t: Toto out)): Comparison Toto
interaction ((a: Comparison Toto) and (b: Comparison Toto)): Comparison Toto
interaction ((a: Comparison Toto) or (b: Comparison Toto)): Comparison Toto
```

Listing A.7

A.4.3 Mutation

```
interface Mutation X is {before: X in, after: X out}

interaction (foo (x: Comparison Number) (y:Comparison Number)): Comparison Toto
interaction (bar): Comparison Toto
interaction (is (t: Toto in)): Comparison Toto
interaction (to (t: Toto out)): Comparison Toto
interaction ((a: Comparison Toto) and (b: Comparison Toto)): Comparison Toto
interaction ((a: Comparison Toto) or (b: Comparison Toto)): Comparison Toto
```

Listing A.8

Appendix B

Compiler graph transformations

Trees sprout up just about everywhere in computer science

Donald Ervin Knuth

In this section, we detail the Graph transformations that are used in the current implementation of the LIDL compiler. Once the parser has returned an AST, the AST transformed into a graph. From there a sequence of 30 graph transformations are performed. As explained in Section 5.1.2, these graph transformations belong to two main groups: The first group deals with the expansion of LIDL definitions in order to get a single "expanded" interaction, which is a composition of basic LIDL interactions. The second group deals with the transformation of this single interaction into a DAG of data flow operators.

The two following sections will explain these two graph transformation groups.

B.1 Definition expansion

This phase takes the AST as input. The AST of a LIDL program is a hierarchy of definitions. Each of these definitions is made of three elements: A signature (Declaration of the name, the arguments, and the interface of the interaction being defined), an interaction expression ()

B.2 Base interaction

B.2.1 Compilation process

1. Parse the source code: A few actions are taken in order to end up with a clean AST.
2. Data Type Definition Expansion: We expand the data type definitions that are present in the code in order to only have basic data types and no type aliases.
3. Interface Definition Expansion: We expand the interface definitions that are present in the code in order to only have base interfaces and not interface aliases.
4. Interaction Type Inference: The main interaction expression tree is visited in order to infer the interface of all interaction used.
5. Interaction Expansion: Once the Interface of all interactions is known, we expand the definitions of these interactions in order to end up with only base interactions.

6. Referential transparency: We merge together identical subexpressions that happen all over the code, so they are computed only once.
7. Identifier elimination: We eliminated variable names by directly linking all interactions that output to a variable to interactions that take the same variable as input.
8. Behaviour elimination: We separated the two parts of behaviours in order to remove the virtual dependancy that exist between them. Indeed, the second operator of a behaviour interaction always receive the active value, so it does not depend on the first argument. We introduce code snippets in the graph in order to generate these active values.
9. Function literal elimination: We replace Function literals with snippets which contains references to the actual implemented function.
10. Function call elimination: We replace Functions calls with actual code snippets.
11. Composition ordering: We transform Composition interactions in order for their arguments to follow the lexical order, so we can match them more easily later.
12. Main Interface instantiation: We replace the interface of the main interaction with an actual composition interaction.
13. Composition pairing: We find couples made of a composition interactions linked with a matching decomposition interactions, and removes them so that their components are linked directly.
14. Composition instantiation: At this point, grouping interaction are either full composition or full decomposition. They cannot be interfaces. We replace the remaining composition and decomposition interactions with code snippets that perform this composition or decomposition.
15. Input Output Ports instantiation: We replace Input and output ports of the main interface with code snippets that access to the correct reference.
16. Graph ordering: We order the resulting directed graph by annotating its node with their order.
17. Variable naming: We give unique names to variables that are present in the code snippets.
18. Code generation: We concatenate the code snippets in the right order.

B.2.1.1 Parser

The parser is developed using PEGjs, which is a parser generator for JS and Parsing Expression Grammars (PEGs).

B.2.1.2 Referential transparency

A nice property of LIDL is referential transparency. This means that two identical expressions always have identical values. This prevents a lot of confusion that can happen in languages that do not have this nice property. In terms of graph transformations, it means that when two subtrees of the expression tree are identical, then they can be considered to be the same subtree.

For example, in the following expression, the interaction `((x)+(1))` is present in three different places, but it has the same meaning everywhere, so it actually represent one and only one interaction. During execution, `((x)+(1))` will only be computed once on each execution step, because it has the same meaning everywhere.

```
(... ... ((x)+(1)) ...
  (... ... ((x)+(1)) ... ...)
  ... ((x)+(1)) ...
)
```

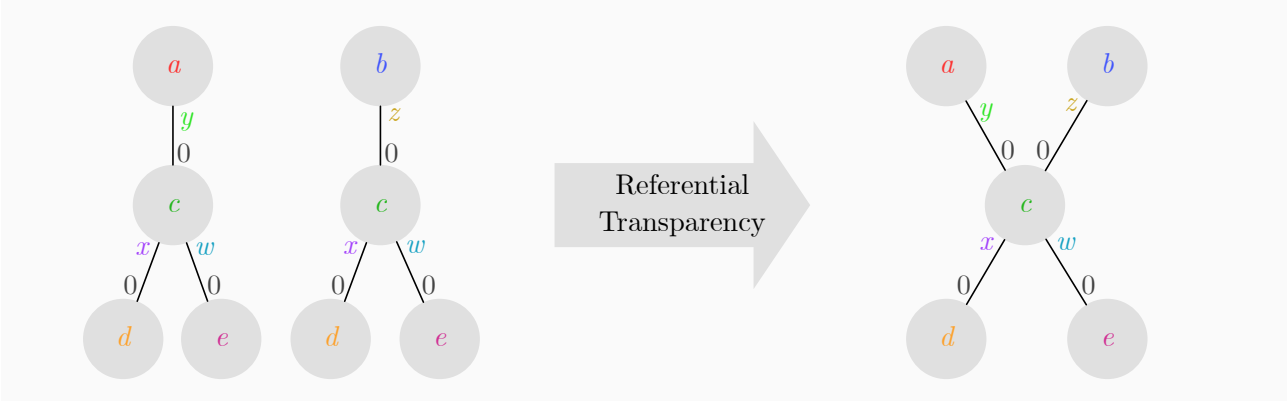


Figure B.1: The referential transparency rule

B.2.1.3 Identifier elimination

The identifier elimination rule is the rule that gives the semantics of the identifier base interaction. In terms of graph rewriting, it means that when different expressions refer to the same identifier interaction, then the identifier can be removed, and these different expressions can directly be linked together.

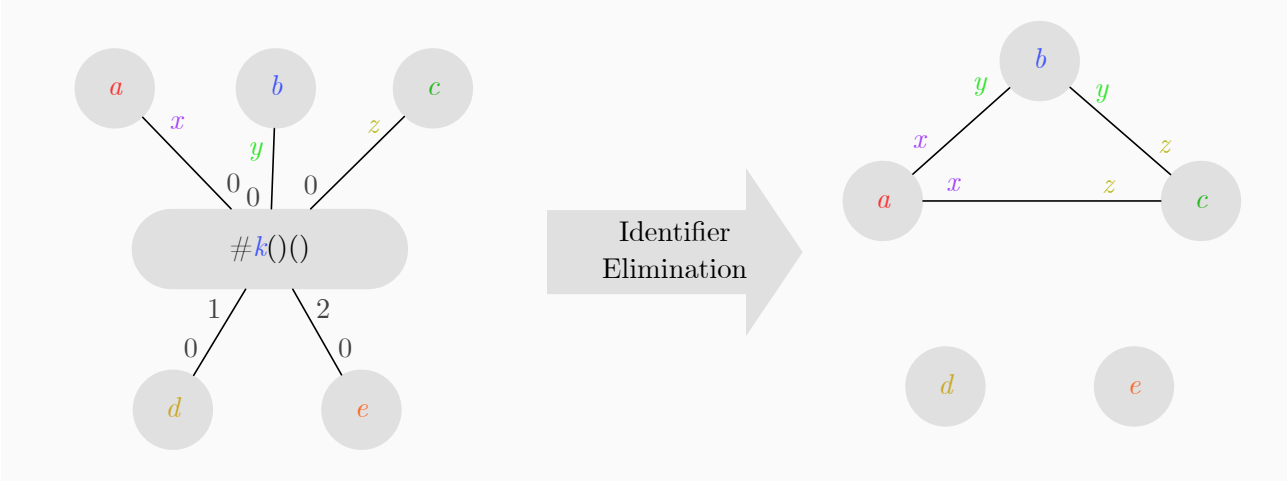


Figure B.2: The identifier elimination rule

For example, in the following expression, the interaction `(#1)` is present in three different places. The identifier elimination rules states that these three expressions will be linked together, and the identifier will be removed. The resulting interaction does not have a textual representation. As explained in Subsection C.5.2, the whole point of the identifier interaction is to enable textual representation, in the form of an expression tree, of structures that are not tree-like.

```
(... ... ((x)=(#1)) ...
  (... ((#1)=(y)) ... )
  ((z)=(#1)) ...
)
```

B.2.1.4 Behaviour separation

The behaviour separation rule is the rule that gives the semantics of the behaviour base interaction. In terms of graph rewriting, it means that the behaviour nodes are split in two different parts. The first part identifies the behaviour interaction with its first argument. The second part consists in sending the *active* value to the second argument.

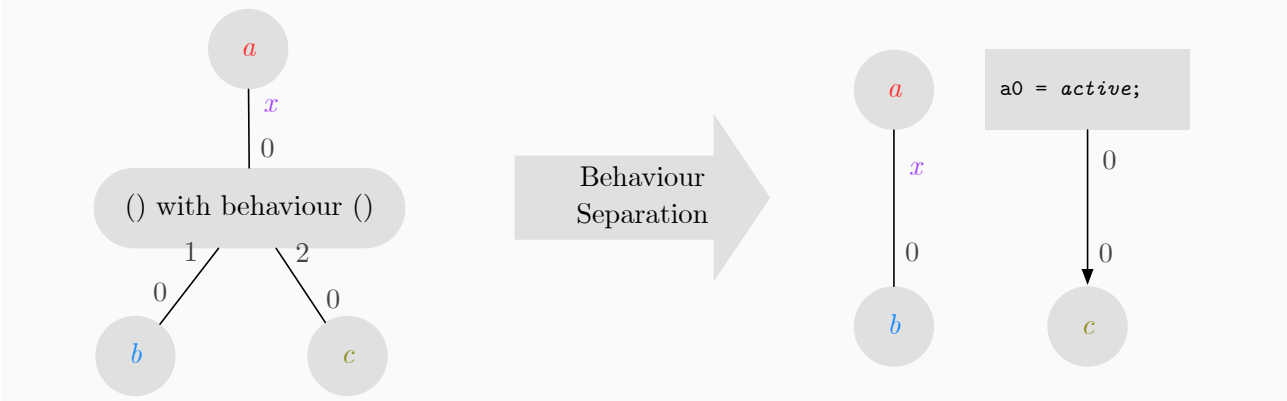


Figure B.3: The behaviour separation rule

B.2.1.5 Function literal linking

The function literal linking rule is the rule that gives the semantics of the function base interaction. In terms of graph rewriting, it means that the function literal nodes are replaced by a link to the appropriate function in the target programming language.

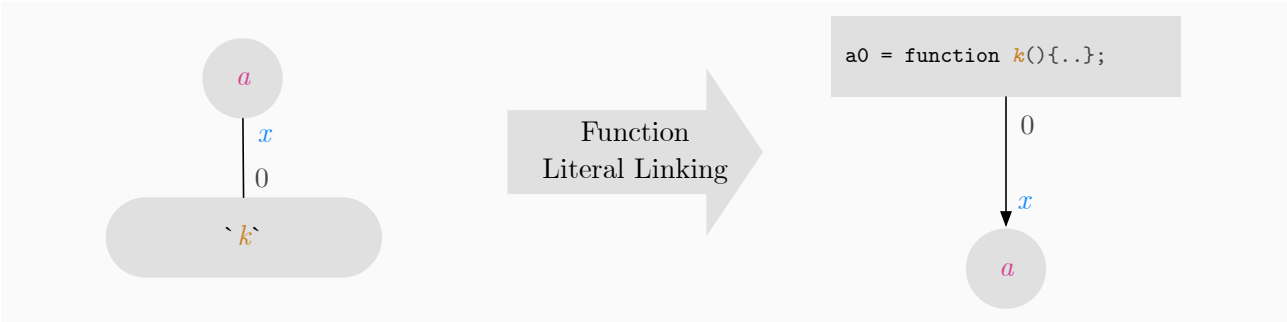


Figure B.4: The function literal linking rule

B.2.1.6 Synchronous function call transformation

The synchronous function call transformation rule gives the semantics of the sychronous function call interactions. It transforms synchronous function call interactions into nodes that call a given function if they receive an activation.

B.2.1.7 Asynchronous function call transformation

The asynchronous function call transformation rule gives the semantics of the asychronous function call interactions. It transforms asynchronous function call interactions into a pair made of an input port and a matching output port.

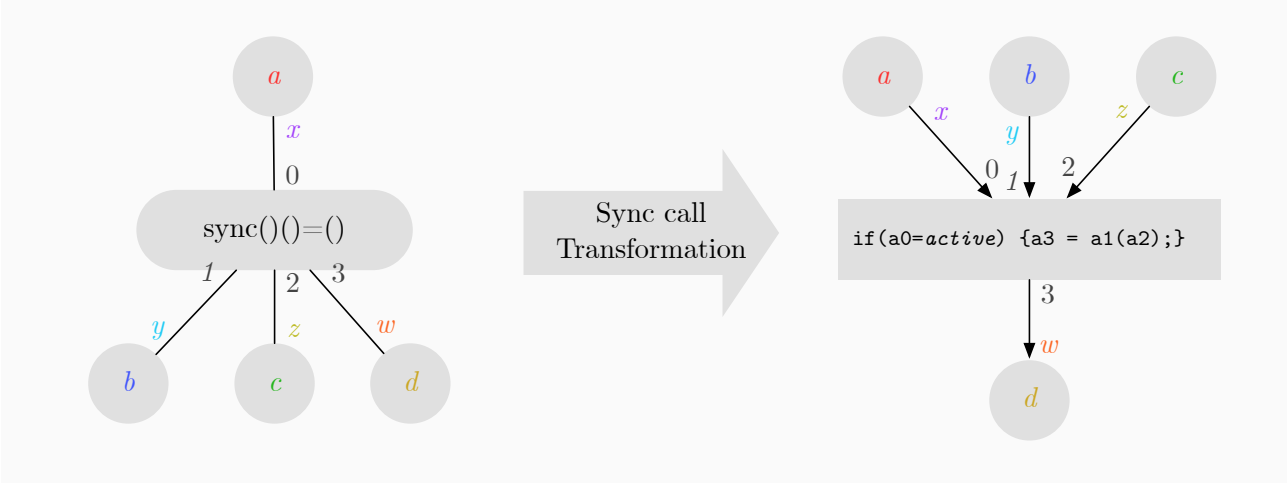


Figure B.5: The synchronous function call transformation rule

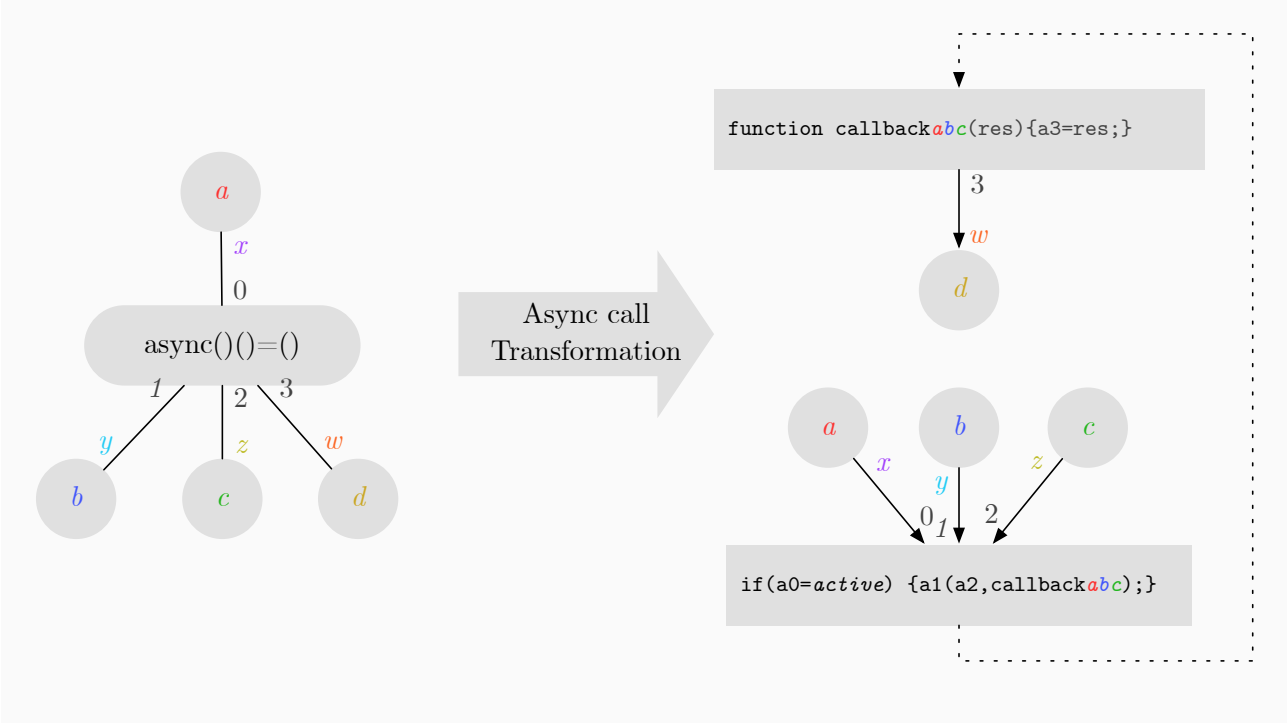


Figure B.6: The asynchronous function call transformation rule

B.2.1.8 Composition ordering

The composition ordering rule simply states that the order of fields in a composition interaction is not significant. Only labels of fields are important. This rule replaces all composition interactions by ordering their labels in lexical order.

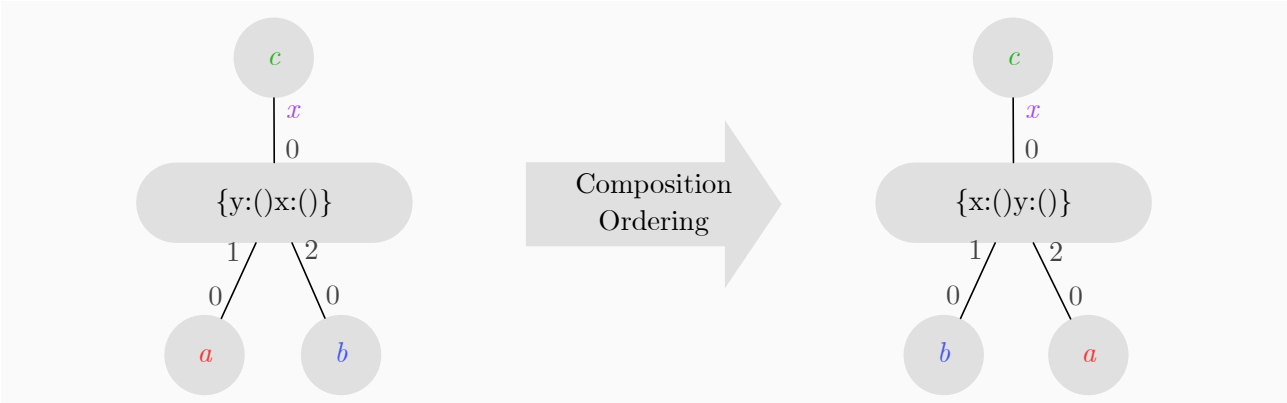


Figure B.7: The composition ordering rule

For example, the composition ordering rule states that the interaction `({ b:(y), a:(x) })` is equivalent to the interaction `({ a:(x), b:(y) })`.

B.2.1.9 Composition matching

The composition matching rule is a rule that gives semantics to composition interactions. It means that when there is a link between a composition interaction and a matching decomposition interaction, then the two matching interactions are removed, and their components are directly linked together instead.

The important thing to notice here is that the composition matching rule allows to remove dependencies between elements that were previously linked together. For example in Figure B.8, before application of the rule, a and d are indirectly linked together, seemingly denoting a dependency relationship between these two nodes. But after application of the rule, it becomes clear that a and d are not in a dependency relationship, which was an illusion caused by their use in the same interface.

B.2.1.10 Composition Transformation

Sometimes, composition (resp. decomposition) interaction are not directly paired with their reciprocal decomposition (resp. composition) interaction. This is the case for example when a composition (resp. decomposition) interaction is used as the input (resp. output) of a function call. In this case, the composition (resp. decomposition) interactions are replaced by a node that actually performs the composition(resp. decomposition) of several pieces of data to (resp. from) a composite piece of data.

B.2.1.11 Multiple resolving

The multiple resolving rule deals with instances where several different data sources try to send signals to a unique target. The multiple resolving rule deals with these situations by adding a resolving unit. At every execution step, there are are three scenarios that the resolving unit can have to deal with:

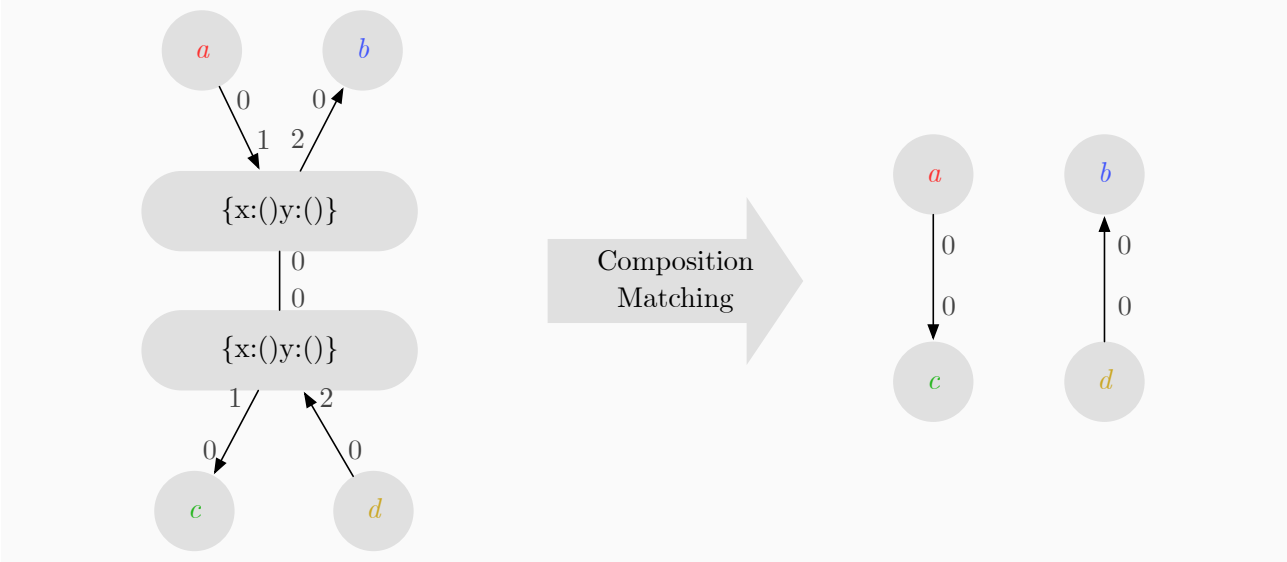


Figure B.8: The composition matching rule

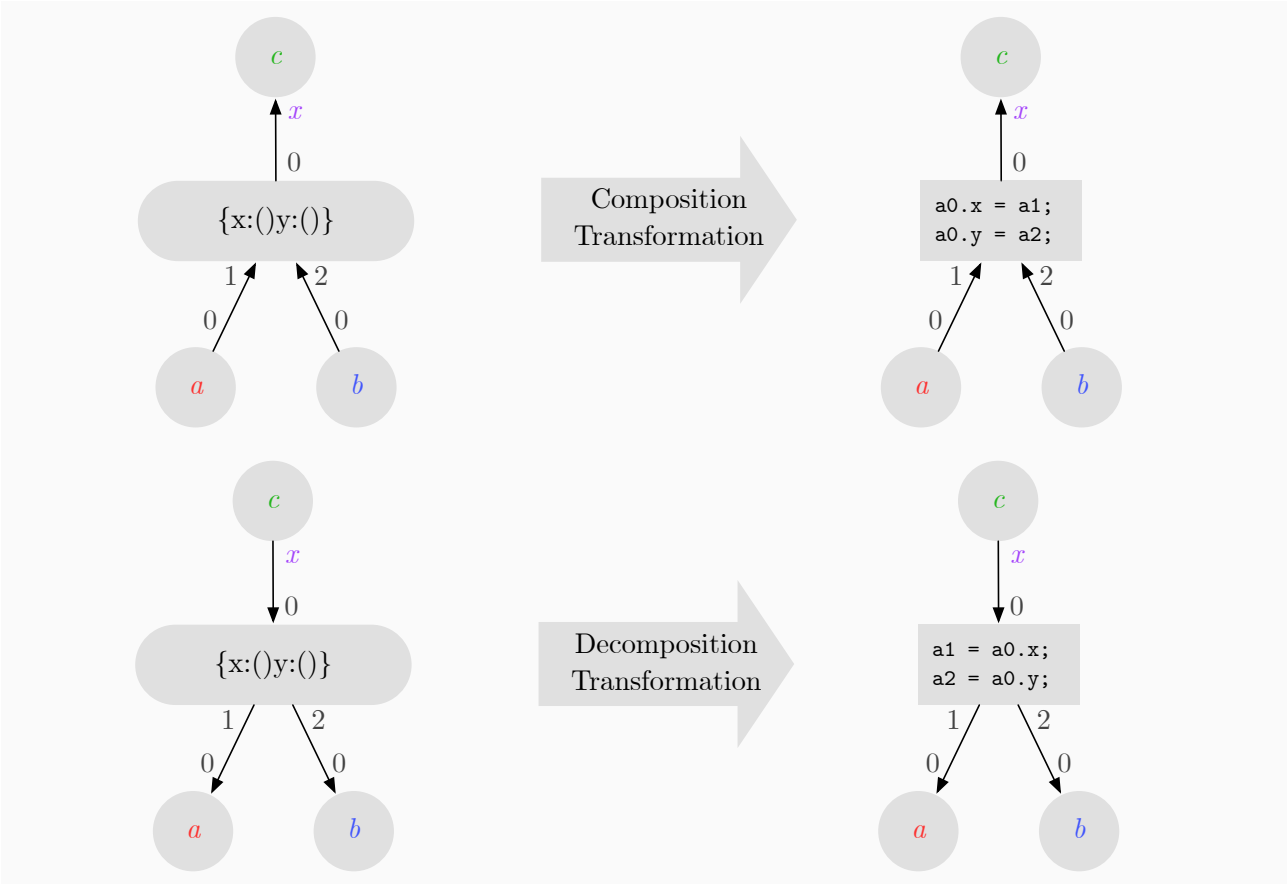


Figure B.9: The composition transformation rule

- All the data sources send the *inactive* value. Then the *inactive* value is received. If no one gives anything to a recipient, then the recipient does not get anything.
- Only one data source sends an active value, all the other being *inactive*. Then the only active value is received. If one person gives something to a recipient, then the recipient get this thing.
- More than one data source send an active value. Then there is a run-time problem. If one person can only receive one thing at a time but is sent more than two at the same time, then there is a problem.

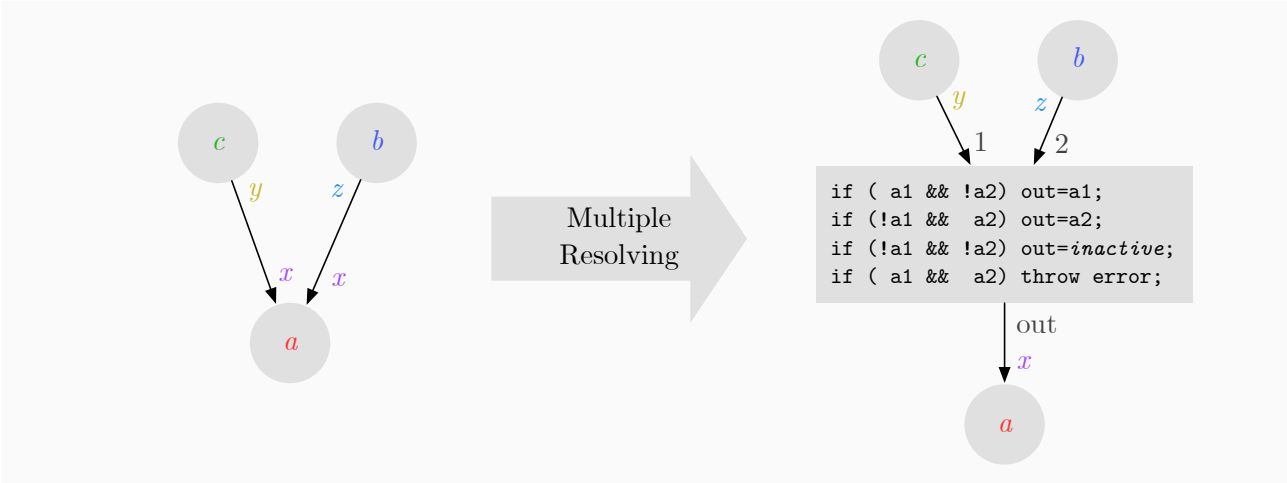


Figure B.10: The multiple resolving rule

For example, in the following example, `((x)=(y))` and `((x)=(z))` both send a signal to the same interaction `(x)`. If they are not both activated at the same time, then everything is fine. But what happens when they are both activated at the same time ? Which value does `(x)` take, `(y)` or `(z)` ? The answer is neither : this case is a runtime error, and we should check that such a situation cannot happen during execution.

```
(... ...
  (... ((x)=(y)) ... )
  (... ((x)=(z)) ... )
)
```

Appendix C

LIDL reference manual

C.1 Definitions

LIDL systems are described by means of definitions. Actually a LIDL system is defined by one and only one definition. This definition can refer to other sub definitions. During compilation, the main definition is expanded using the other definitions, until only base interactions remain.

C.1.1 Syntax

Extended Backus-Naur Form (EBNF) grammar of the definition language:

$$
\begin{aligned}
\langle \text{definition} \rangle &::= \langle \text{definitionSimple} \rangle \mid \langle \text{definitionComplex} \rangle \\
\langle \text{definitionSimple} \rangle &::= \langle \text{signature} \rangle \text{ `is' } \langle \text{expression} \rangle \\
\langle \text{definitionComplex} \rangle &::= \langle \text{signature} \rangle \text{ `with' } \langle \text{definition} \rangle * \text{ `is' } \langle \text{expression} \rangle \\
\langle \text{signature} \rangle &::= \langle \text{datatypeSignature} \rangle \mid \langle \text{interfaceSignature} \rangle \mid \langle \text{interactionSignature} \rangle \\
\langle \text{expression} \rangle &::= \langle \text{datatype} \rangle \mid \langle \text{interface} \rangle \mid \langle \text{interaction} \rangle
\end{aligned}
$$

C.1.2 Simple definitions

LIDL definitions use the `is` keyword. Here is the general form of a simple definition in LIDL:

```
1 |$A$| is |$a$|
```

This definition means that symbols that matches the signature A should be replaced with the expression a. Table C.1 shows some example simple definitions.

C.1.3 Complex definitions

LIDL definitions can be nested using the `with` keyword. For example, in the definition structure expressed in Listing C.1, we say that definitions B and E are children of definition A. Figure C.1 represent this structure graphically.

Listing C.2 presents some actual example of complex definitions in LIDL.

```
|$A$|
with
  |$B$|
  with
    |$C$| is |$c$|
    |$D$| is |$d$|
  is |$b$|
  |$E$| is |$e$|
is |$a$|
```

Listing C.1: An example of nested complex definitions

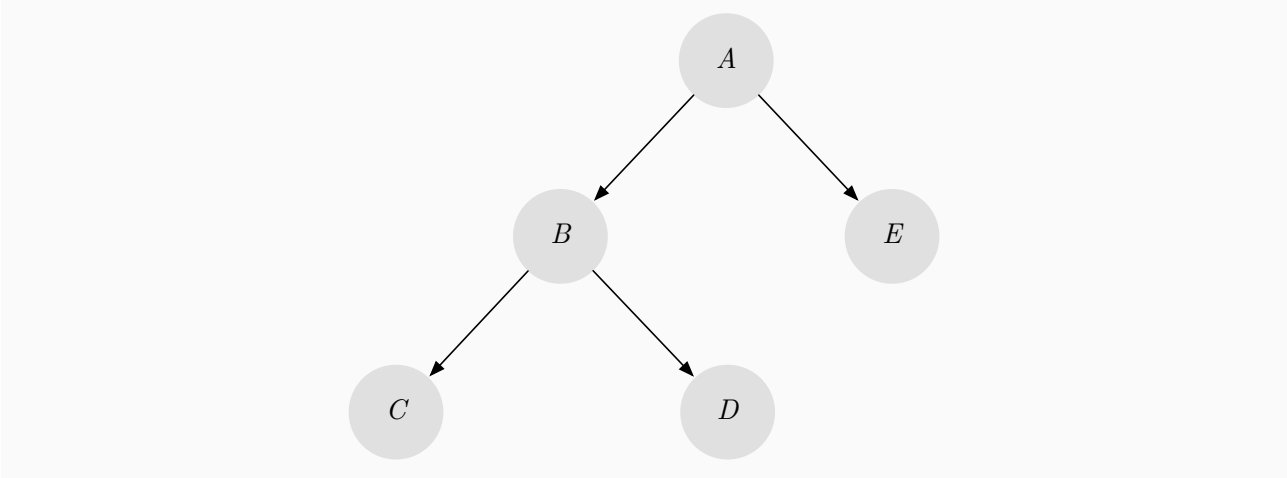


Figure C.1: Graphical representation of an example definition tree.

Definition	Description
`data Point is {x:Number, y:number}`	The data type to represent a point
`interface Mouse is {position:Point in, click:Boolean in }`	An interface to simplistically represent a mouse
`interaction (Hello):Text out is ("Hello")`	An interaction that outputs the 'Hello' string

Table C.1: Example simple definitions

C.1.4 Definitions scope

In order to prevent unexpected behaviour caused by matching unexpected definitions, definitions have a scope: they are not exposed to the whole definition tree. Instead, there is one and only one simple scoping rule to know in LIDL:

Definition C.1 (Definition scope)**.** *A definition can only refer to any of its ancestor definitions including itself, and their children definitions. See Figure C.2.*

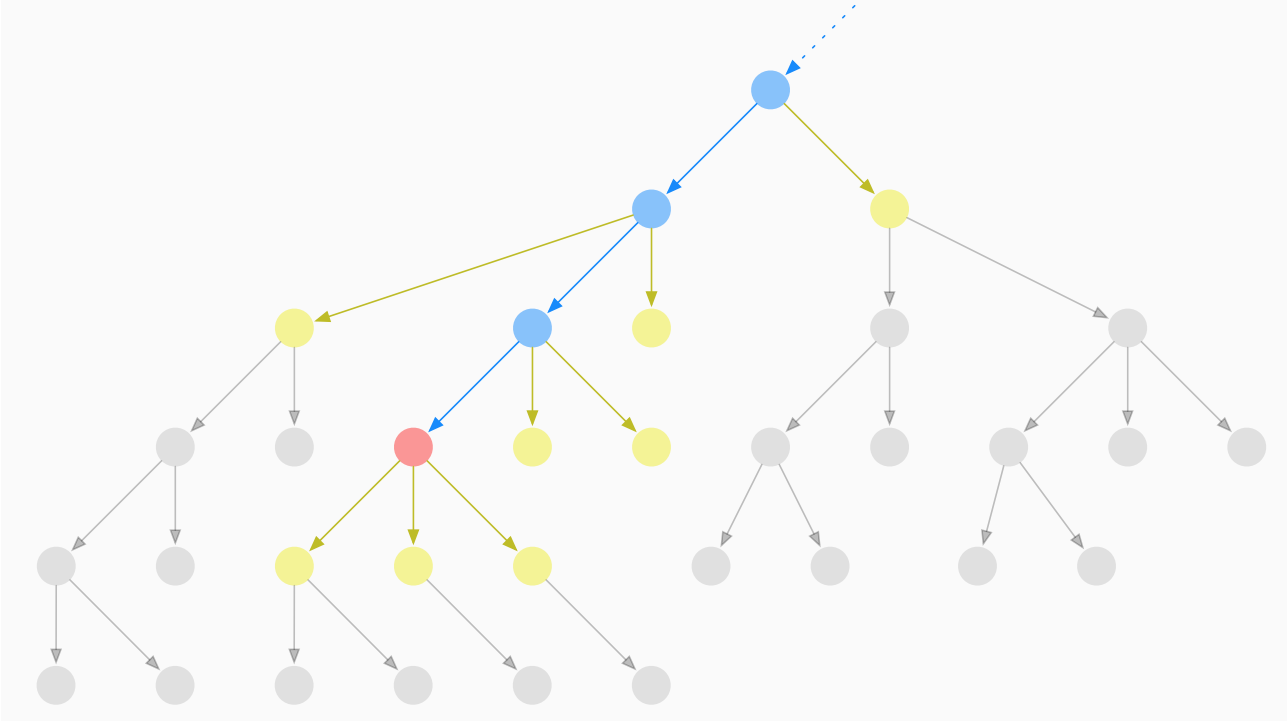


Figure C.2: Graphical representation of definitions scoping in LIDL. The red definition can refer to blue definitions because they are its ancestors and to yellow definitions because they are direct children of its ancestors, but not to itself. See Definition C.1

The reference relation being the reciprocal of the exposure relation, this definition has a corollary which is also useful when reasoning about the code:

```
data
  Person
with
  data
    Date
  is
    {day:Number, month:Number, year: Number}
is
  {
    firstName:Text,
    lastName:Text,
    birthDate:Date
  }

interaction
  (hello twice):Text out
with
  interaction (hello):Text out is ("Hello !")
is
  ((hello)(hello))
```

Listing C.2: Actual examples of complex definitions in LIDL

Corollary C.1 (Definition scope reciprocal)**.** *A definition is exposed to any of its descendant, and any descendent of its parent. See Figure C.3.*

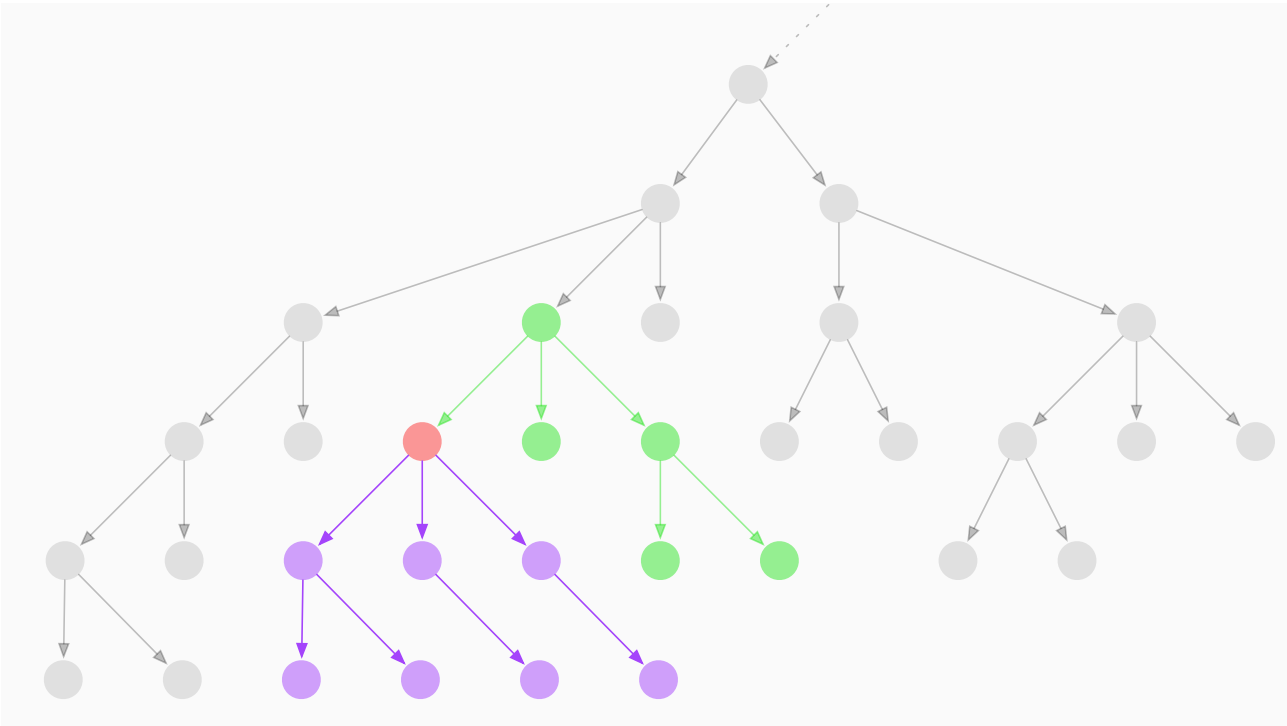


Figure C.3: Graphical representation of definitions scoping in LIDL. The red definition is exposed to purple definitions because they are its descendants, and to green definitions because they are the descendants of its parent, but not to itself. See Corollary C.1

In the example of Listing C.1, we have the following scoping:

- The expression a can refer to the definitions B and E. However, definitions C and D are masked because they are not direct children of A.
- The expression b can refer to the definitions A, C, D and E.
- The expression c can refer to the definitions A, B, D and E.
- The expression d can refer to the definitions A, B, C and E.
- The expression e can refer to the definitions A and B, However, definitions C and D are masked because they are not direct descendants of A.

Conversely, it is equivalent to say that:

- The definition A is exposed and can be used in expressions b, c, d, e.
- The definition B is exposed and can be used in expressions a, c, d, e.
- The definition C is exposed and can be used in expressions b and d, but not in a and e because it is masked by B.
- The definition D is exposed and can be used in expressions b and c, but not in a and e because it is masked by B.
- The definition E is exposed and can be used in expressions a, b, c and d.

C.1.5 Definition validity

There are three categories of definitions in LIDL: Definitions of data types, definitions of interfaces, and definitions of interactions. Listing C.3 shows LIDL syntax for these three categories respectively.

Definitions of different categories can be mixed together. For example an interaction definition can have children data type definitions. The same scoping rule applies for all.

```
data
  MyData
is
  ...

interface
  MyInterface
is
  ...

interaction
  (MyInteraction):MyInterface
is
  ...
```

Listing C.3

Definition C.2 (Valid definition)**.** *A LIDL definition is valid if and only if its signature matches its expression and its expression is valid.*

C.1.6 LIDL System

A simple definition of an interaction, which only use basic data types, basic interfaces and basic interactions, is called a LIDL system. Such systems are self contained, and can be interpreted as-is,

without needing to refer to other definitions.

Definition C.3 (LIDL sytem)**.** *Any valid, self-contained, simple definition of an interaction is called a LIDL system.*

Theorem C.1 (LIDL sytem)**.** *LIDL systems are interactors as defined in Definition 3.1.*

Proof. A LIDL system x is a simple definition. The signature of x defines several interfaces that are implemented by the system (1). The interaction expression of x is the interaction that is performed by the system (2). From (1) and (2), we find that x is an interactor as defined in Definition 3.1. □

C.2 Data

Interactive systems are systems that exchange data. Hence, LIDL has a way to define data types. LIDL type system is a very basic structural type system. Data types are divided into three categories: Atomic data types, Composite data types, and Function types.

C.2.1 Syntax

EBNF grammar of the data type language:

⟨datatype⟩ ::= ⟨atomic⟩ | ⟨composite⟩ | ⟨function⟩
⟨atomic⟩ ::= *UpperCamelCasedName*
⟨function⟩ ::= ⟨datatype⟩ ‘->’ ⟨datatype⟩
⟨composite⟩ ::= ‘{’ ⟨element⟩ ∗ ‘}’
⟨element⟩ ::= ⟨label⟩ ‘:’ ⟨datatype⟩
⟨label⟩ ::= *lowerCamelCasedName*

C.2.2 Atomic data types

Atomic data types represent the smallest unit of data, they cannot be broken into smaller pieces. LIDL is not aware of the fact that these data types might be further divided in smaller pieces.

There are 4 atomic data types built in LIDL : `Activation`, `Boolean`, `Number` and `Text`. However, any other atomic data type can be added by LIDL users, as long as they provide functions to construct and operate on instances of these types.

Here are some perfectly valid atomic data types in LIDL : `Activation`, `Boolean`, `Number`, `Text`, `Blob`, `Image`, `MyCustomType`, `Array`.

LIDL Atomic data types can be used to represent data types which are actually composite types in the target programming language. For example, in the example above, `Image` might be implemented in a target programming language by an array of colors, which is a composite data type. But LIDL sees instances of these types as atoms, and is not aware of any further division of these types.

The main consequence of this is that in LIDL, the idea of a *piece* of an atomic type does make sense. Hence, a *piece* of an atomic type instance cannot depend on another *piece* of the same instance. For

example, in the `Image` atomic data type, one could not state in LIDL that the color of a certain pixel of one image depends on another the color of another pixel of the same image. This statement would contain a self-reference to the image, which LIDL forbids. See Figure C.6.

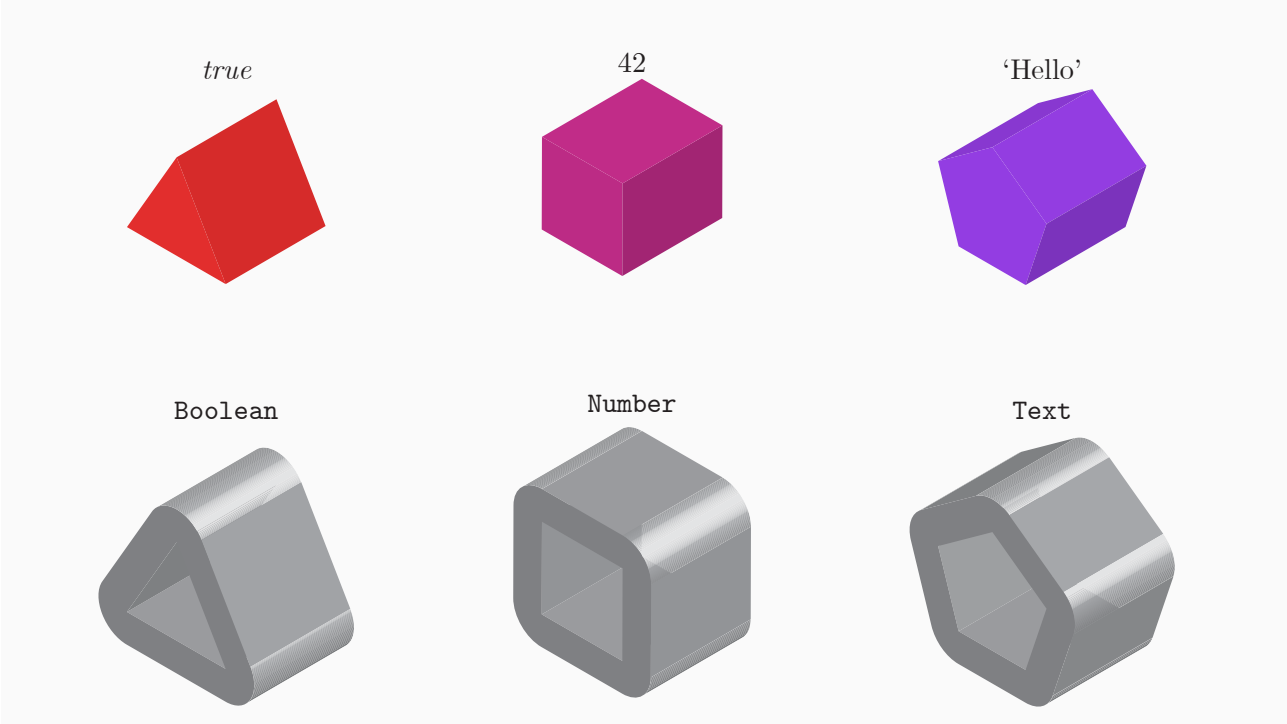


Figure C.4: A metaphore of atomic data types. Pieces of data are represented on top, and their associated data types are represented as "molds" that allow to create them and check wheter a piece of data is of a certain type.

C.2.3 Composite data types

There is only one category of composite data types in LIDL. This only category of composite data type is what is commonly named records. These types compose a fixed set of labeled fields of other types. This is similar to C structs, JS records, for instance. Here are some composite data types:

Data type	Description
`{x:Number, y:Number}`	2D Point with coordinates labelled x and y
`{name:Text, surname:Text, birthDate:Number}`	Person

Table C.2: Composite data types examples

Rationale The importance of composite data types in LIDL is that LIDL is aware that these types are made of smaller *pieces*. Hence, for composite data types, it makes sense to state that one piece of a datum depends on another piece of the same datum. See Figure C.6 for an example.

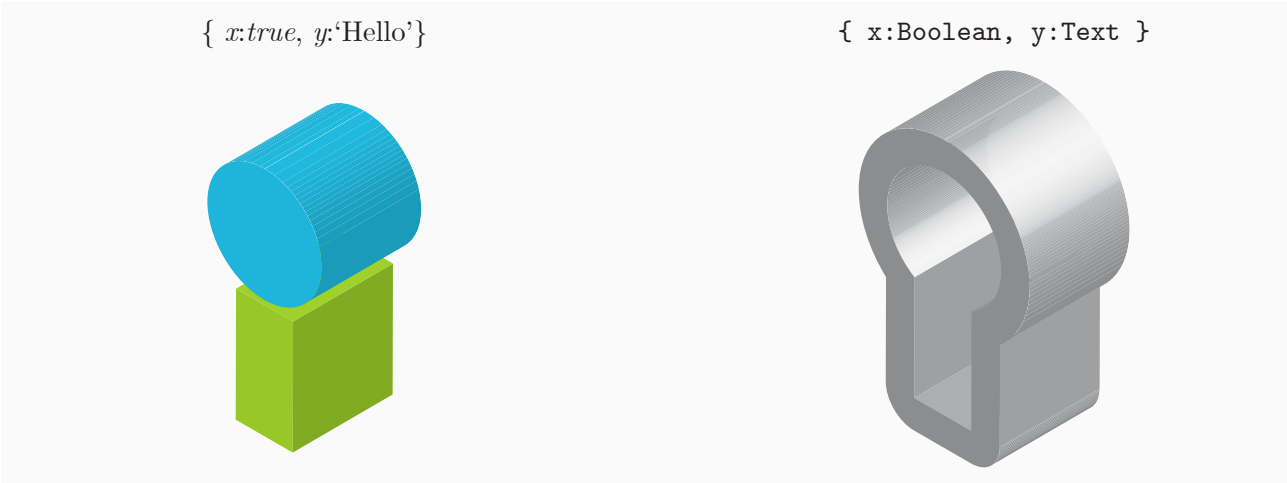


Figure C.5: A metaphore of composite data types. Composite data types (on the right) are data types that fit composed data (left)

C.2.4 Function types

The function data type is important in LIDL. It represents a function that can be passed as any other kind of data. Functions are first class citizens in LIDL, so they can be combined using LIDL. However, functions are not defined in the LIDL language, but have to be defined in target programming languages. LIDL functions take only one argument and return only one value. However both the argument and the return value can be composite data types.

Rationale LIDL is a language to describe interactions, not computations. Functions are the basic unit of computation, hence their definition is outside of the scope of LIDL. But most interactions involve computations, this is why functions exist in LIDL. Functions take only one argument and return only one value for the sake of simplicity.

C.2.5 Structural typing

Typing in LIDL is structural. This means that two composite data types defined with two different names can be used interchangeably if they have the same structure, i.e. the same set of labels associated with the same data types. The order of the labels is not significant in composite types.

This structural typing allows anonymous data types. For example, consider the following definitions:

```
1 data Point2D is { x: Number, y: Number }
2 data Vector2D is { x: Number, y: Number }
3 data AnOtherType is { b: Number, a: Number }
```

With these definitions, `Point2D`, `Vector2D` and `{x: Number, y:Number}` are strictly equivalent data types, these three names can be used interchangeably. However, they are not equivalent to `AnOtherType`, because this one is equivalent to `{a: Number, b: Number}` and not `{x: Number, y:Number}`.

For atomic data types, the story is a bit different. LIDL knows nothing about atomic data types. Apart from their names, everything about them is delegated to the target programming language. Hence LIDL has no information but the name of atomic data types. This is why, for atomic data

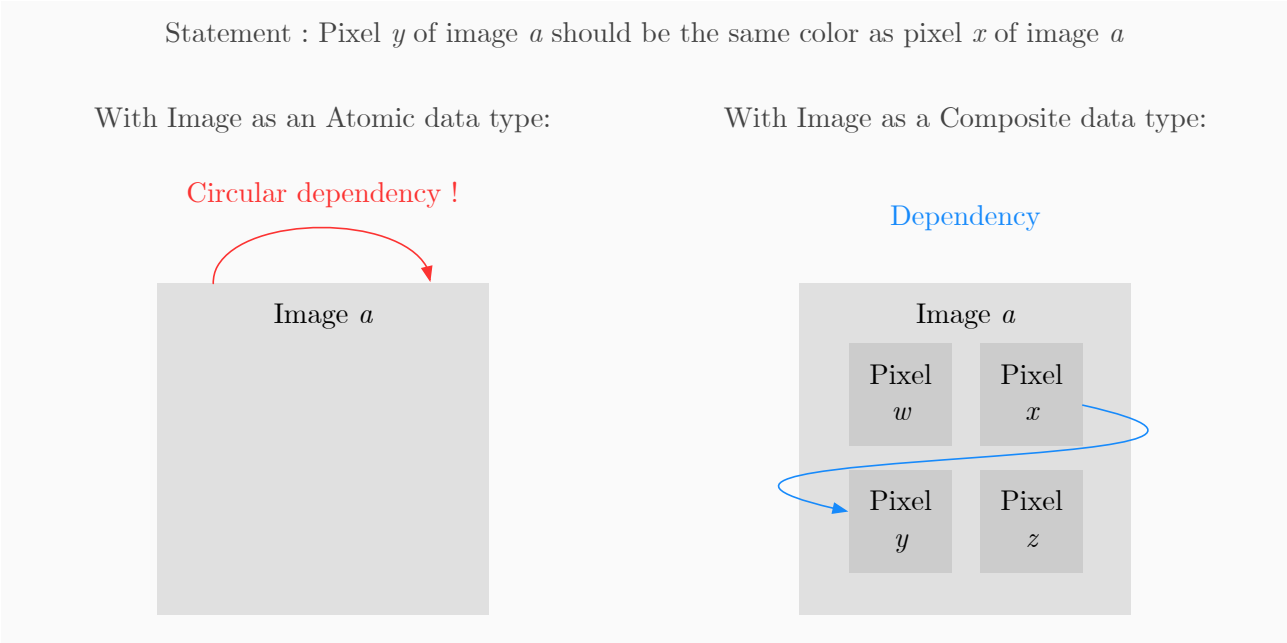


Figure C.6: The main point for composite data types in LIDL: Resolving circular dependencies by going deeper in the data structure.

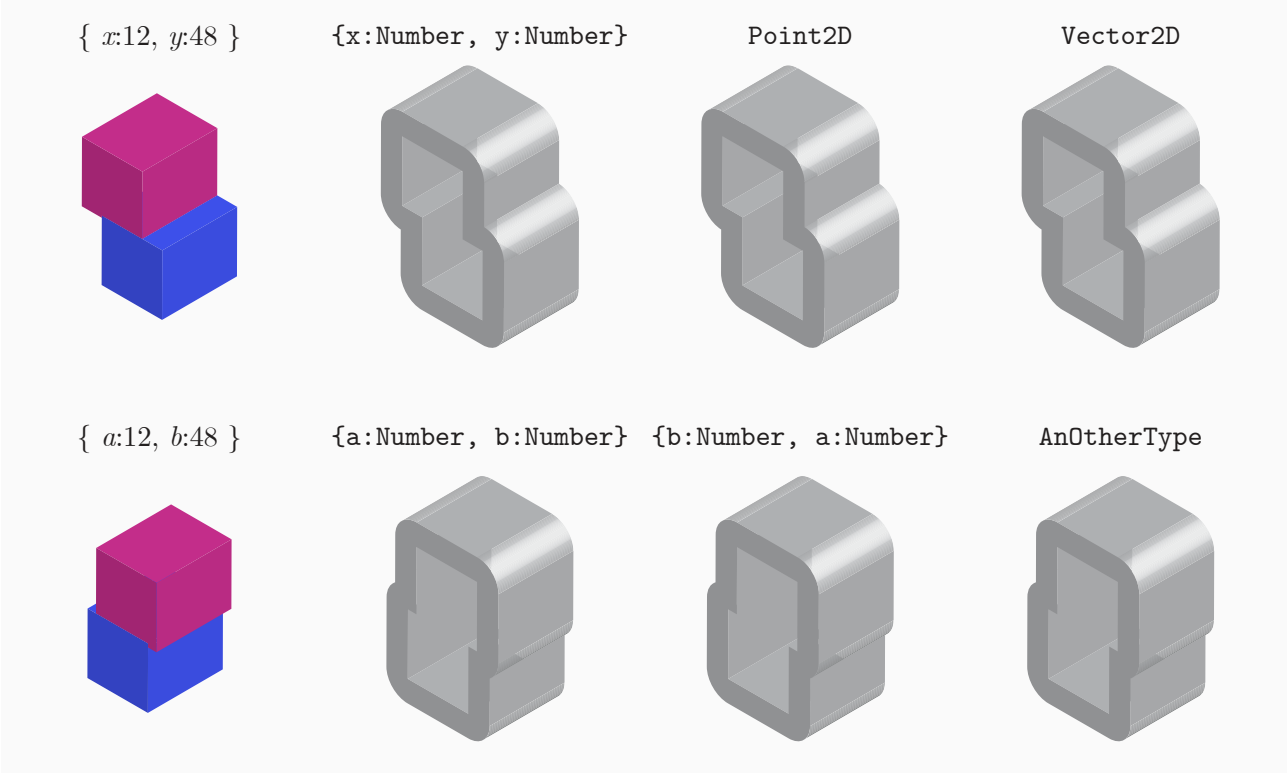


Figure C.7: A metaphore of structural typing. Data types with different names but the same structure are equivalent.

types, the typing judgment is nominative. Two atomic data types with different names are considered different even if they are somewhat compatible in the target programming language.

C.2.6 Data activation

LIDL data types have an extra value called `inactive`. At any point in time, a signal can either be `inactive`, or have a value. This concept is called activation in LIDL. The activation of a piece of data represents its presence.

This is similar to option types in other programming languages, like the `Maybe` type in Haskell or the `Optional` class templates in object-oriented programming languages such as C++ or Java. This is also somewhat similar to the `null` pointer in C, or `null` in Javascript.

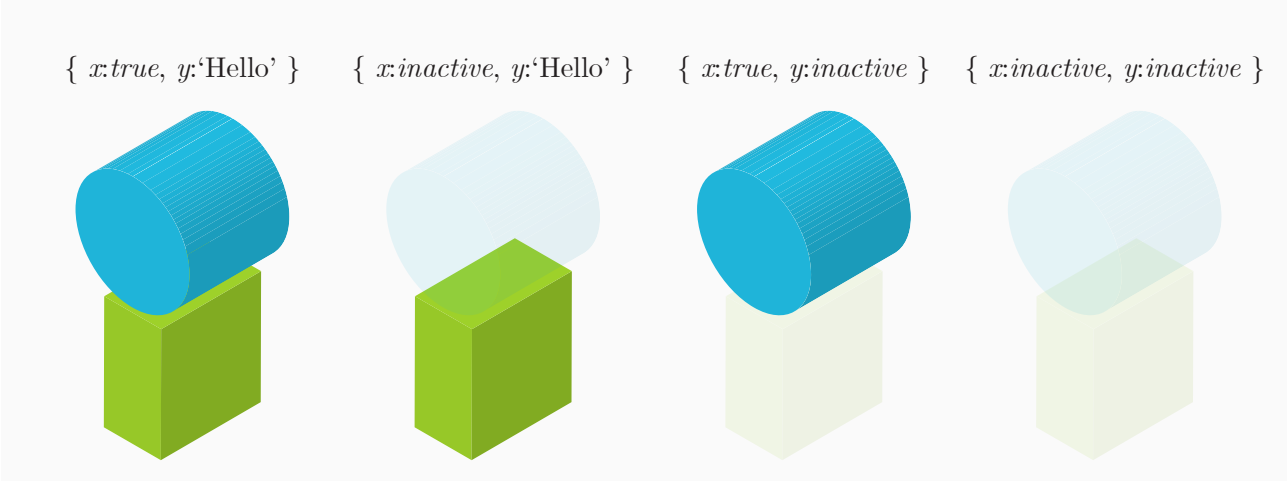


Figure C.8: A metaphore of data activation. Some pieces of data might not be present but the data is still a valid instance of its data type.

C.3 Interfaces

Section C.2 explained how data is represented in LIDL. LIDL is a language about interaction, and interaction is the exchange of data. So how is the exchange of data represented ? This is what LIDL interfaces are for. LIDL interfaces are a way to describe the structure of inputs and outputs of a system.

Interfaces in LIDL are divided in two categories : atomic interfaces, and composite interfaces.

C.3.1 Syntax

EBNF grammar of the interfaces language:

$$
\begin{aligned}
\langle \text{interface} \rangle &::= \langle \text{atomic} \rangle \mid \langle \text{composite} \rangle \\
\langle \text{atomic} \rangle &::= \langle \text{datatype} \rangle \ \langle \text{direction} \rangle \\
\langle \text{composite} \rangle &::= \text{`\{'} \ \langle \text{element} \rangle * \ \text{`\}'} \\
\langle \text{element} \rangle &::= \langle \text{label} \rangle \ \text{`:'} \ \langle \text{interface} \rangle \\
\langle \text{label} \rangle &::= \textit{lowerCamelCasedName} \\
\langle \text{direction} \rangle &::= \texttt{in} \mid \texttt{out}
\end{aligned}
$$

Here are some examples of LIDL interfaces:

Interface	Remark
`Image out`	Atomic
`Text out`	Atomic
`Text in`	Atomic

`{`
`  mouse:{`
`  position: { x: Number, y:Number},`
`  buttons: {left: Boolean, right: Boolean},`
`  wheel: {x: Number, y:Number}`
`  } in,`
`  screen: Image out`
`}`
	Composite

Table C.3: Interfaces examples

C.3.2 Atomic interfaces

Atomic interfaces are composed of a single data type, and a single direction. Here are some examples of atomic interfaces:

Atomic Interface	Description
`Number in`	An input port for numbers
`Text out`	An output port for text
`{x:Number, y:Number} in`	An input port for 2D points
`{a:{name: Text, surname: Text},` `b:{name: Text, surname: Text}} out`	A port that outputs two people, labelled a and b

Table C.4: Atomic interfaces examples

These interface are atomic because they all only contain one indication of direction. The last two are atomic interfaces of composite data types.

C.3.3 Composite interfaces

The same way LIDL offers a single way to compose data types into composite data types, LIDL offers a single way to compose interfaces into composite interfaces, with a similar syntax and semantics.

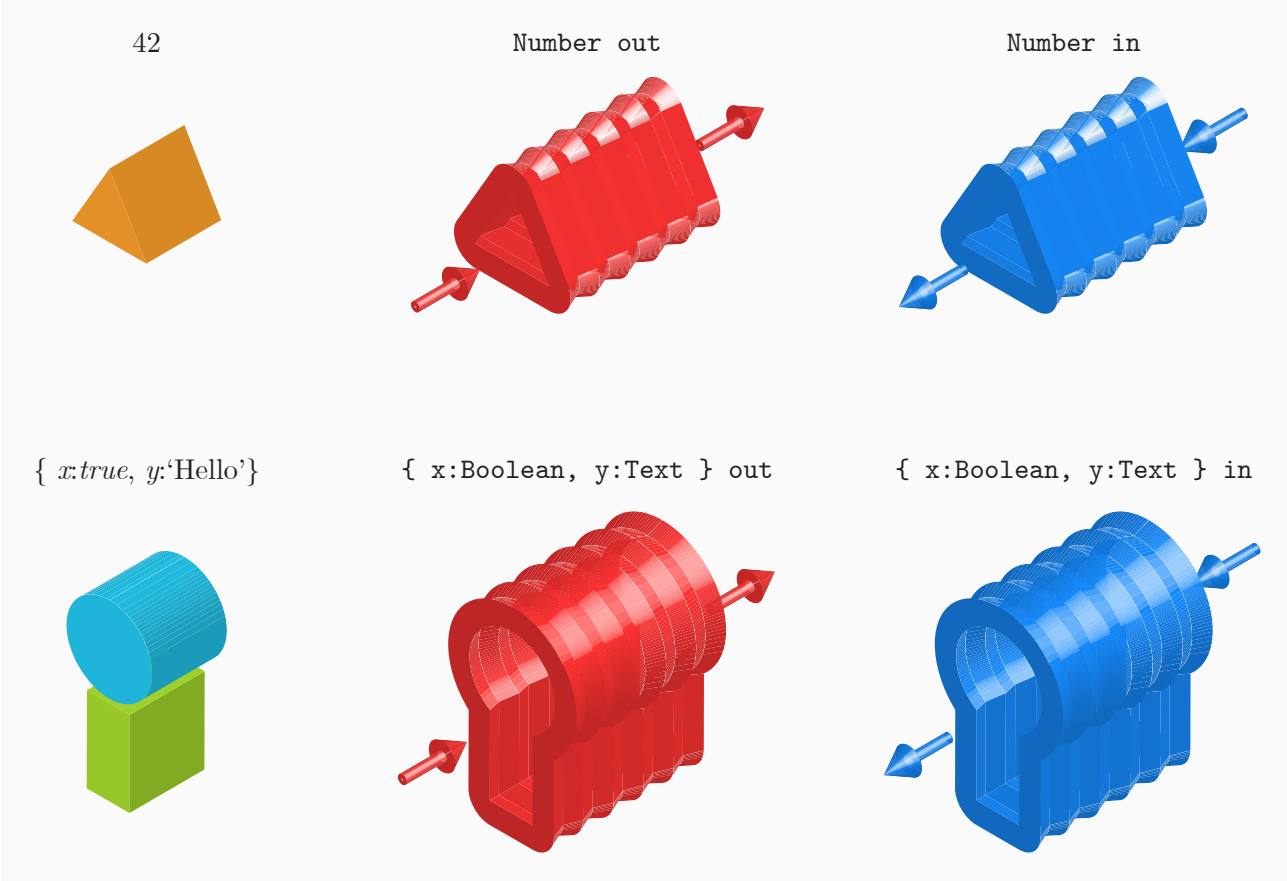


Figure C.9: A metaphore of atomic interfaces. Interfaces are pipes that let data (left) go in one direction: either out (red) our in (blue). The data can be either atomic (top) or composite (bottom)

Composite interfaces are a fixed set of labeled fields of other interfaces. here are some example composite interfaces:

Interface
`{click:Activation in, numberOfClicks: Number out}`
`{mouse: Mouse in, keyboard: Keyboard in, graphics: Graphics out}`
`{userId: Text in, password: Text in, authorization: Activation out}`

Table C.5: Interfaces examples

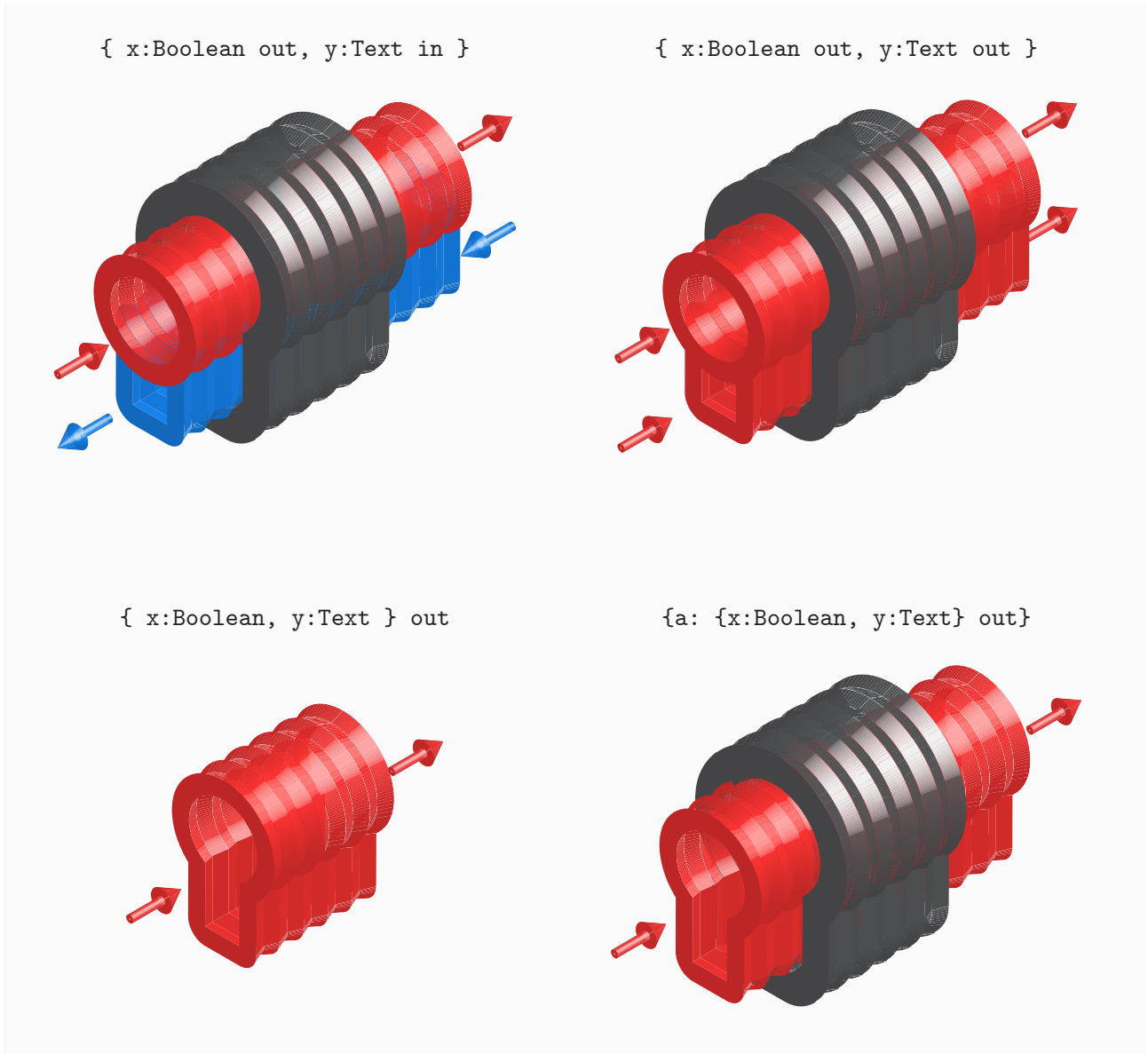


Figure C.10: A metaphore of composite interfaces. Composite interfaces are (black) pipes that contain other pipes. In this figure, one of the four interfaces is an atomic interface.

C.3.4 Interface operations

A set of operations can be applied to interfaces, such as conjugation, union, intersection, globalization, localization, toDataType.

toDataType Every LIDL interface has an associated data type. The function *toDataType* gives the data type associated with an interface. Basically, *toDataType* works by removing all directions information (`ins` and `outs`) of an interface. Here is the definition of the *toDataType* function:

Definition C.4 (*toDataType*)**.**

$$\begin{array}{rrcl} toDataType: & interfaces & \rightarrow & datatypes \\ & x\ \texttt{in} & \mapsto & x \\ & x\ \texttt{out} & \mapsto & x \\ & \{l_1 : i_1,\ \cdots,\ l_n : i_n\} & \mapsto & \{l_1 : toDataType(i_1),\ \cdots,\ l_n : toDataType(i_n)\} \end{array}$$

Theorem C.2 (Properties of toDataType)**.** *toDataType is surjective and not injective.*

Proof. $\forall x \in datatypes, (x\ \texttt{in}) \in interfaces \wedge (x\ \texttt{out}) \in interfaces$ □

Table C.6 shows some examples of interfaces with their associated data type.

Interface	Associated data type
`Number in`	`Number`
`{click:Activation in, numberOfClicks:Number out}`	`{click:Activation, numberOfClicks:Number}`
`(Number -> Number) in`	`(Number -> Number)`
`{position:{x:Number, y:Number}in, target:{x:Number, y:Number}out }`	`{position:{x:Number, y:Number}, target:{x:Number, y:Number}}`

Table C.6: Some example of toDataType.

Conjugation The conjugation of an interface transforms all its in into outs.

Definition C.5 (conjugate)**.**

$$\begin{array}{rrcl} conjugate: & interfaces & \rightarrow & interfaces \\ & x\ \texttt{in} & \mapsto & x\ \texttt{out} \\ & x\ \texttt{out} & \mapsto & x\ \texttt{in} \\ & \{l_1 : i_1,\ \cdots,\ l_n : i_n\} & \mapsto & \{l_1 : conjugate(i_1),\ \cdots,\ l_n : conjugate(i_n)\} \end{array}$$

Table C.7 shows some examples of conjugation.

Interface	Conjugate interface
`Number in`	`Number out`
`{click:Activation in, numberOfClicks: Number out}`	`{click:Activation out, numberOfClicks: Number in}`
`Number -> Number in`	`Number -> Number out`

Table C.7: Some example of conjugation.

Union and intersection The union and intersection operators perform respectively an union and an intersection of two interface fields. In case of duplicate fields with identical labels but different content the first operand get the priority.

Definition C.6 (interface union)**.**

$$\begin{array}{rcl} union: & (compositeinterfaces)^2 & \rightarrow compositeinterfaces \\ \begin{pmatrix} \{l_1^1 : i_1^1,\ \cdots,\ l_m^1 : i_m^1,\ x_1 : y_1^1,\ \cdots,\ x_l : y_l^1\} \\ \{l_1^2 : i_1^2,\ \cdots,\ l_n^2 : i_n^2,\ x_1 : y_1^2,\ \cdots,\ x_l : y_l^2\} \end{pmatrix} & \mapsto & \begin{Bmatrix} l_1^1 : i_1^1,\ \cdots,\ l_m^1 : i_m^1, \\ l_1^2 : i_1^2,\ \cdots,\ l_n^2 : i_n^2, \\ x_1 : y_1^1,\ \cdots,\ x_l : y_l^1 \end{Bmatrix} \end{array}$$

Definition C.7 (interface intersection)**.**

$$\begin{array}{rcl} intersection: & (compositeinterfaces)^2 & \rightarrow compositeinterfaces \\ \begin{pmatrix} \{l_1^1 : i_1^1,\ \cdots,\ l_m^1 : i_m^1,\ x_1 : y_1^1,\ \cdots,\ x_l : y_l^1\} \\ \{l_1^2 : i_1^2,\ \cdots,\ l_n^2 : i_n^2,\ x_1 : y_2^1,\ \cdots,\ x_l : y_l^2\} \end{pmatrix} & \mapsto & \left\{x_1 : y_1^1,\ \cdots,\ x_l : y_l^1\right\} \end{array}$$

Interface A	Interface B	Union(A,B)
`{ x: Number in }`	`{ y:Number out }`	`{ x:Number in, y:Number out }`
`{ x:Number in, y:Number out }`	`{ x:Text in, z:Number out }`	`{ x:Number in, y:Number out, z:Number out }`

Table C.8: Some example of Interface union.

Globalization and localization Globalization of an interface tries to get all the direction indication as shallowly as possible in the hierarchy

```
globalization { x: Number in, y: Number in}
=
{ x: Number, y: Number } in
```

Localization is the opposite, it pushes direction indication as deep as possible in the hierarchy, actually going to the leaves.

```
localization { x: Number, y: Number} in
=
{ x: Number in, y: Number in }
```

C.3.5 Remarks and rationales

C.3.5.1 Terminology

Interfaces are called interfaces in the sense accepted in UI design: a set of input and output components. But they are also similar to Java interfaces: Java interfaces can be implemented by several classes, LIDL interfaces can be implemented by different interactions. Also Java interface present a structured set of methods that can be used to send data (Setters) or receive data (Getters).

C.3.5.2 Rationale

Interfaces allow to present the static aspects of HMIs

C.4 Interactions

Interactions are the most important part of LIDL. They actually describe what LIDL systems do.

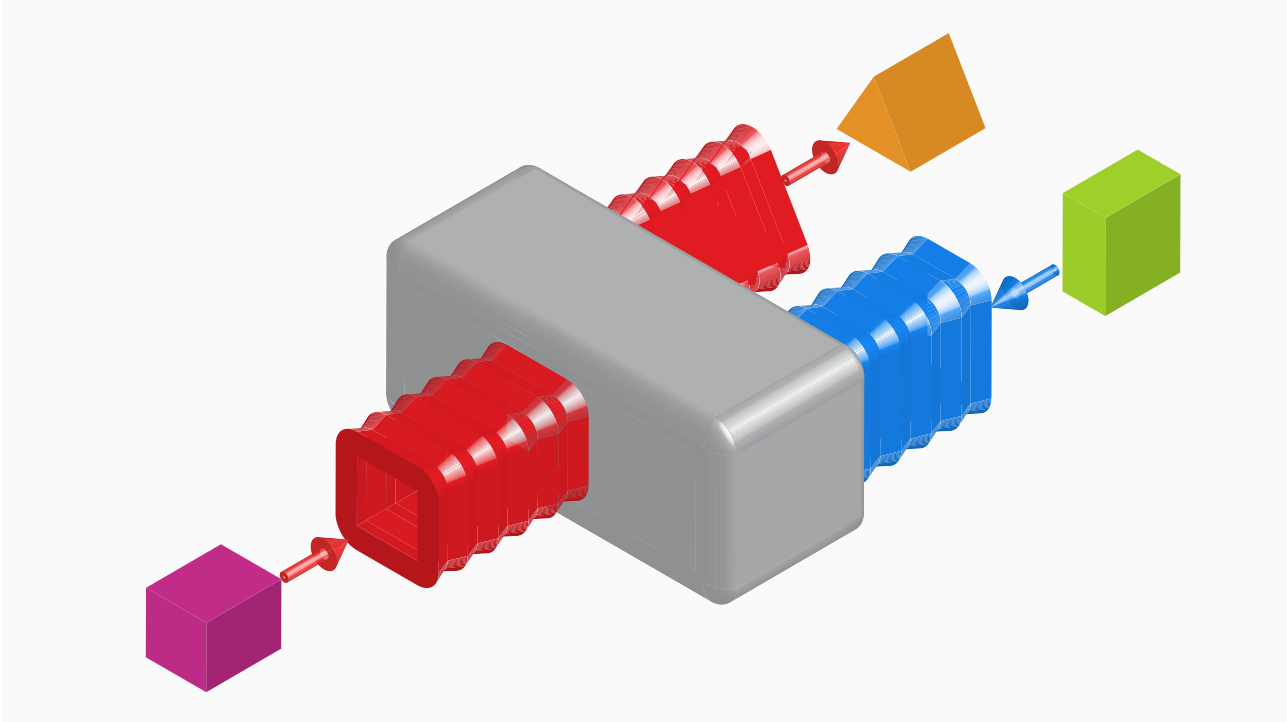


Figure C.11: A metaphore of interactions. Interactions are components which have several interfaces and can be composed in order to make more complex systems.

C.4.1 Syntax

The interaction language has a simple EBNF grammar:

$$
\begin{aligned}
\langle \text{interaction} \rangle &::= \ (\ \langle \text{element} \rangle * \ ) \\
\langle \text{element} \rangle &::= \ \langle \text{text} \rangle \ | \ \langle \text{interaction} \rangle \\
\langle \text{text} \rangle &::= \ \textit{any text not containing parentheses}
\end{aligned}
$$

Interaction LIDL parenthesis-heavy syntax has one goal: make interactions appear clearly. Indeed, each pair of matching parentheses in a LIDL program represent exactly one interaction.

The list of elements that compose an interaction (text or interactions) allows to derive two important attributes of the interaction: its operator (See C.4.3) and its operands (See C.4.4).

As we will see, the LIDL interaction syntax is a trade-off that has one disadvantage: A parenthesis-heavy textual syntax, and several advantages, such as a very easy way to specify embedded DSLs, a simple grammar, and self-explanatory operators.

C.4.2 Interaction expression

An interaction expression is a composition of interactions.

Interactions are compositional: they can have children. For example, the interaction `(do ((this) or (that)) and then (thisOther))` has two children: `((this) or (that))` and `(thisOther)`. The whole structure of this interaction is explained in figure C.12.

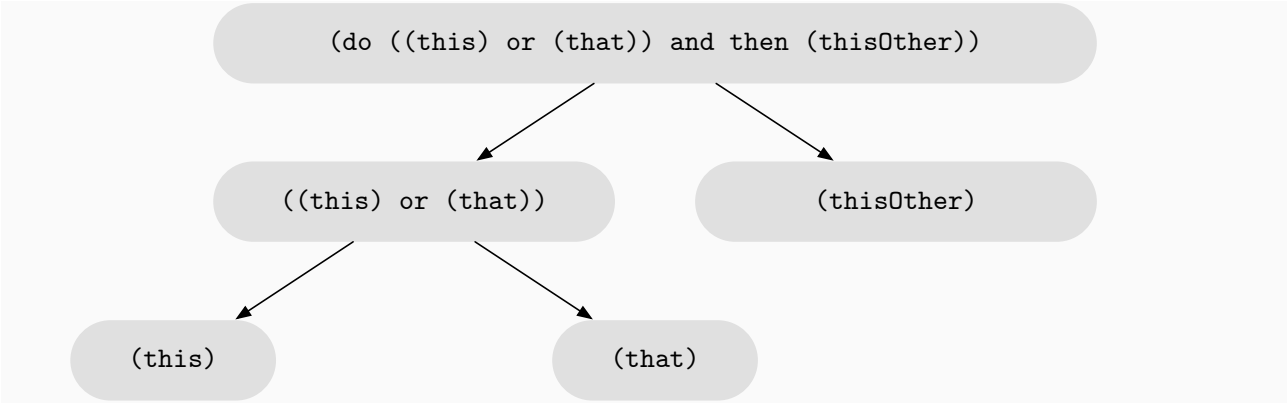


Figure C.12: Graphical view of an exemple interaction expression made of 5 interactions.

C.4.3 Interaction operator

The operator of an interaction is the equivalent of a function name in other programming languages. However, LIDL interaction syntax makes interaction operators a little bit trickier to derive. Indeed LIDL interaction operators do not specifically have a prefix or infix notation. LIDL allows all kind of notations.

For example, consider the following interaction expression: `(do ((this) or (that)) and then (thisOther))`. This expression is made of 5 interactions, whose operators are 'do()and then()', '()or()', 'that', 'this', and 'thisOther'. In most other languages, an equivalent expression might have looked more like `doandthen( or( this(), that() ), thisOther() )`.

More formally, given a list $\{e_1, e_2, \ldots, e_n\}$ of elements of an interaction resulting from the parsing explained in C.4.1, we derive the interaction operator using the function *operator* defined by:

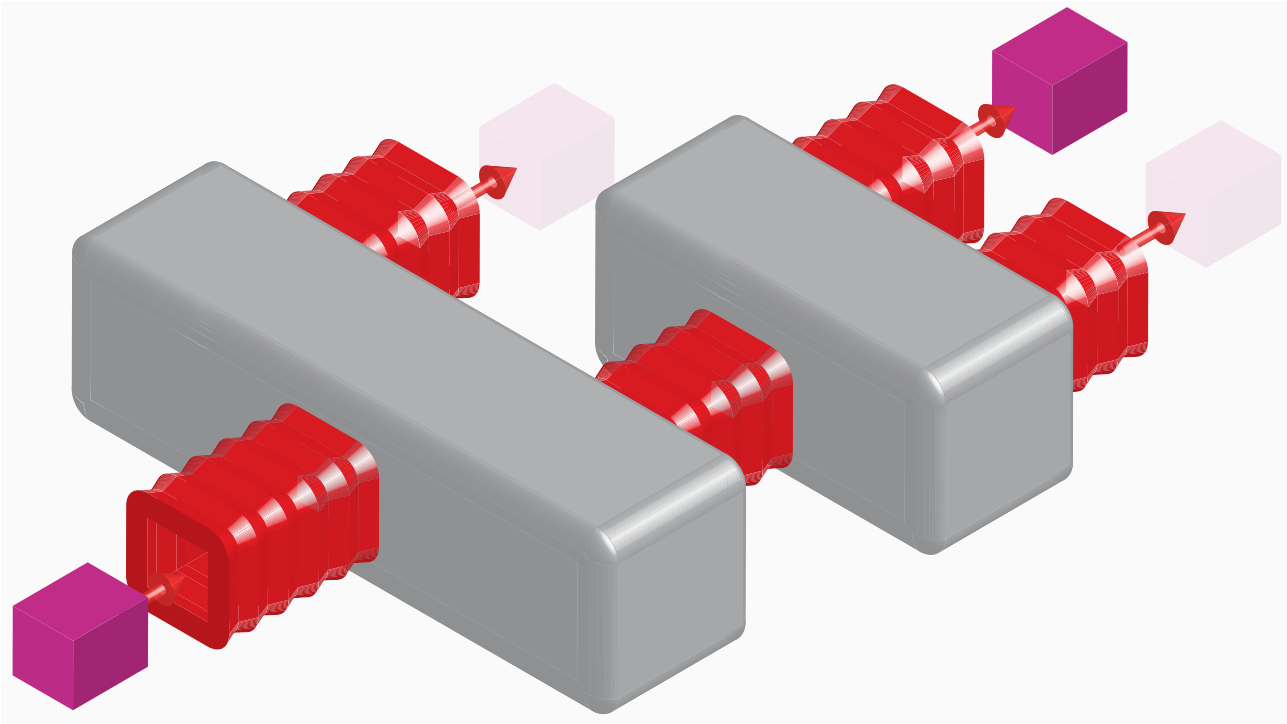


Figure C.13: A metaphore of interaction expressions. This figure represents a metaphore of the expression of Figure C.12

$$operator(\{e_1, e_2, \ldots, e_n\}) = concatenate(map(\{e_1, e_2, \ldots, e_n\}, toString))$$

$$\text{where} \quad toString(e) = \begin{cases} toLowerCase(removeSpaces(e)) & \text{if } e \text{ is a string} \\ \text{`()'} & \text{if } e \text{ is an interaction} \end{cases}$$

Interaction	Operator
`(when (this) then (that))`	'when()then()'
`(label (active) displaying ("Hi!"))`	'label()displaying()'
`((x)+(5))`	'()+()'
`("Hello !")`	'"Hello !"'
`(active)`	'active'

Table C.9: Examples of interaction operators

C.4.4 Interaction operand

An interaction is composed of one operator, and a list of operands. For example, consider the following interaction expression: `(do ((this) or (that)) and then (thisOther))`.. This interaction has 2 operands: `((this) or (that))`, and `(thisOther)`. Interaction operands are also interactions.

More formally, given a list $\{e_1, e_2, \ldots, e_n\}$ of elements of an interaction resulting from the parsing explained in C.4.1, we derive the interaction operands using the function *operand* defined by:

$$operand(\{e_1, e_2, \ldots, e_n\}) = filter(\{e_1, e_2, \ldots, e_n\}, isInteraction))$$

$$\text{where} \quad isInteraction(e) = \begin{cases} true & \text{if } e \text{ is an interaction} \\ false & \text{if } e \text{ is a string} \end{cases}$$

C.4.5 Each interaction is a signal

Each interaction is a signal. The data type of this signal is the datatype-ation of the interaction interface, see section C.3.4 for more information about this operation.

All the data that flows through LIDL system is represented in the form of signals. All signals of a LIDL system are synchronized. At any execution step, each signal of a LIDL system is assigned a value. The structure of signals in a LIDL program is fixed at compile-time. Signals cannot be dynamically created or deleted at run-time.

C.4.6 Remarks and rationales

C.4.6.1 Rationale for the interaction syntax

LIDL interaction syntax allow to have a very simple interaction grammar, yet it enables the creation of any kind of custom operators. Many languages disallow the creation of custom infix or postfix operators, for example.

The languages that allow operator overload have a specific syntax for the creation of custom operators, adding to these languages complexity. For example here is how an infix sum operator can be defined for a data type Box in different languages:

- In C++, the `operator+` method has to be overloaded: `Box operator+(const Box&, const Box&) { ... }`
- In Python, the `__add__` method has to be overloaded: `def __add__(self, other): ...`
- In Groovy, the `plus` method has to be overloaded: `Box plus(Box other) {...}`
- In Haskell, the whole `Num` typeclass has to be instantiated: `instance (Num a,Num b) => Num (Box a b) where Box (a,b) + Box (c,d) = ...`
- In Elm[86], No existing operator an be overloaded, but infix operators can be specified using some special keywords: `infixl 4 box+`

In LIDL, creating the sum interaction for two boxes is the same as creating any other interaction: `interaction ((a:Box in) + (b:Box in)) : Box out is (something)`. LIDL solution is to use parentheses. This allow to not only define prefix, infix and postfix operators for example, but also other arbitrary operators that cannot be described in other languages.

C.5 Base interactions

The set of basic interaction in LIDL is restricted. It turns out that a small number of fundamental interactions are enough to define all possible interactions. These fundamental interactions are the following:

- Function: the only literal, which links to functions specified in a programming language.
- Composition and decomposition: the only way to merge and separate data.
- Function application: the only way to actually perform computations.
-
- Previous: the only way to get an information from the past.
- Identifier: the only aliasing mechanism, allowing referencing same entities in different places of the code.
- Behaviour: the way to specify specific behaviours for objects.

These categories of base interactions are defined by their operators. For more information about operators, see Subsection C.4.3. The following subsections details practical use of these base interactions. The precise semantics of the base interactions are detailled along with the semantics of the rest of the language in Section C.6.

C.5.1 Function

C.5.1.1 Description

Function interactions are simply function literals. They contain the name of a function. This function has to be defined in the target programming language and will be linked to the LIDL program. All LIDL functions are complete, and their implementation must satisfy this rule.

C.5.1.2 Syntax

The function interactions are defined by the use of backticks. They simply contain the name of a function between backticks. Here is the EBNF grammar that define all the possible operators of the function interaction category (There are as many valid function operators in LIDL as there are valid function names in the target programming language):

$$\begin{aligned} \langle \text{functionOperator} \rangle &::= \text{`` '} \langle \text{functionName} \rangle \text{`` '} \\ \langle \text{functionName} \rangle &::= AValidNameForAFunctionInTheTargetLanguage \end{aligned}$$

Table C.10 shows several examples of function operators.

C.5.1.3 Example use

Here is an example of a function interaction in LIDL:

```
1 (`sum`)
```

In order to be able to use this function, it has to be coded in the target programming language. For example, Listing C.4 shows the function sum coded in Javascript, and Listing C.5 shows the equivalent function coded in C.

Interaction	Operator
(‘sin‘)	‘ ` sin ` ’
(‘setParameter‘)	‘ ` setParameter ` ’
(‘concatenateStrings‘)	‘ ` concatenateStrings ` ’
(‘filter‘)	‘ ` filter ` ’

Table C.10: Examples of valid functions operators

```
function sum(arg) {
  if(arg){
    if(arg.a && arg.b) {
      return arg.a + arg.b;
    } else {
      return null;
    }
  } else {
    return null;
  }
}
```

Listing C.4: A version of functionSum in Javascript

C.5.2 Identifier

C.5.2.1 Description

The main point of identifier interactions is to enable the creation of variables that can be used in different parts of the code.

The identifier interactions are the only way to create cross references in LIDL Code. Identifier interactions link expression tree leaves. This changes LIDL programs structure from Trees to DAGs.

C.5.2.2 Syntax

The following grammar describes all operators that are valid identifier operators. The only thing to notice is that any operator starting with a hash character ‘#’ is a valid identifier operator.

$$\begin{aligned} \langle \text{identifierOperator} \rangle &::= \text{‘\#’} \ \langle \text{element} \rangle * \\ \langle \text{element} \rangle &::= \langle \text{string} \rangle \mid \text{‘()’} \\ \langle \text{string} \rangle &::= \textit{Anything except parentheses} \end{aligned}$$

Table C.11 shows some examples of valid identifier interactions and their associated operators.

```
Number sum(Struct_a_Number_b_Number_tcurtS arg) {
  if(arg.activation) {
    if(arg.value.a.activation && arg.value.b.activation) {
      return NumberActive(arg.value.a.value + arg.value.b.value);
    } else {
      return NumberInactive();
    }
  } else {
    return NumberInactive();
  }
}
```

Listing C.5: A version of functionSum in C

Interaction	Operator
(# my Variable)	'#myVariable'
(# when (this) then (that))	'#when()then()'
(#(x)+(5))	'#()+()'
(#"Hello !")	'#"Hello !"'

Table C.11: Examples of valid identifiers operators

C.5.2.3 Examples of use

Basic use In the following example, we reference a global variable called myVariable.

```
((# myVariable) = (5))
...
((# myResult) = ((# myVariable) + (1)) )
```

Step	(5)	(# myVariable)	((# myVariable) + (1))	(# myResult)
1	5	5	6	6
2	5	5	6	6
3	5	5	6	6

Table C.12: Timing diagram of an execution of an identifier interaction

Advanced use In order to have local variables, the semantics of the identifier interaction play together with the semantics of definitions expansion.

C.5.3 Behaviour

C.5.3.1 Description

The behaviour interaction is a way to specify a behaviour associated with an interaction. It is defined by a single operator: `()with behaviour()`. Here is the EBNF grammar that define all the possible operators of the "behaviour" interaction category (This grammar is restricted, there is actually only one behaviour operator):

⟨behaviourOperator⟩ ::= `() with behaviour ()`

For any interface A, we have the following behaviour interaction:

```
interaction ((x:|$\textcolor{c0}{A}$|) with behaviour (y:Activation
↪  out)):|$conjugate (\textcolor{c0}{A})$|
```

The first argument `x` can be seen as the return value. It is an interaction to which the behaviour interaction is equivalent. The second argument `y` is the behaviour which is being activated by the behaviour interaction.

C.5.3.2 Example use

General example Here is the most generic way to use the behaviour interaction. Table C.13 shows an execution of this interaction.

```
((x) with behaviour (y))
```

The signature of the behaviour interaction used here is:

```
interaction ((x:Number in) with behaviour (y:Activation out)):Number out
```

Step	`(x)`	`((x) with behaviour (y))`	`(y)`
0	42	42	*active*
1	π	π	*active*

Table C.13: Timing diagram of an execution of the previous interaction

C.5.4 State

C.5.4.1 Description

LIDL programs are executed step by step, synchronously. Signals values do not propagate from an execution step to the next. The only way to have access to signal values from previous steps is to store value at certain time steps and retrieve them at later time steps.

This interaction is really similar to the `pre` operator in Lustre [57].

C.5.4.2 Syntax

The previous interactions are all defined by a single operator: `previous()`. The only difference between the different versions of the previous interaction is the data types of the operand. Here is the EBNF grammar that define all the possible operators of the "previous" interaction category (This grammar is restricted, there is actually only one previous operator):

⟨statefulOperator⟩ ::= `retrieve () from last time and store () for next time`

```
interaction
  (retrieve(x:|$\textcolor{c0}{A}$|out) from last time and store
↪  (y:|$\textcolor{c0}{A}$|in)for next time):Activation in
```

C.5.4.3 Example use

Simple example Here is the most basic way to use the previous interaction. Table **??** shows an execution of this interaction.

```
interaction
  (previous(x:Number in)):Number out
is
  ((y) with behaviour (retrieve (y) from last time and store (x) for next time))
```

C.5.5 Composition

C.5.5.1 Description

The composition interactions are the way to compose and decompose signals into composed data types and interfaces. Compose and decompose signals into sub signals.

C.5.5.2 Syntax

The composition interactions are defined by the use of braces. They start with an opening brace and finish with a closing brace. In between, they contain elements, which are composed of a label followed

Step	(y)	(retrieve(x) from last time and store (y) for next time)	(x)
1	42	*active*	*inactive*
2	0	*active*	42
3	64	*active*	0
4	12	*active*	64
5	1337	*active*	12
6	24	*inactive*	*inactive*
7	36	*inactive*	*inactive*
8	48	*active*	1337
9	60	*active*	48
10	*inactive*	*inactive*	*inactive*
11	4	*inactive*	*inactive*
12	*inactive*	*active*	60
13	40	*active*	*inactive*
14	2	*active*	40
15	*inactive*	*active*	2

Table C.14: Timing diagram of an execution of the stateful interaction

by a colon and an operand `()`. Here is the EBNF grammar that define all the possible operators of the composition interaction category (There is an infinity of composition operators):

$$\begin{aligned} \langle \text{composition} \rangle &::= \texttt{\{}\ \langle \text{compositionElement} \rangle * \ \texttt{\}} \\ \langle \text{compositionElement} \rangle &::= \langle \text{elementKey} \rangle\ \texttt{:}\ \texttt{()} \\ \langle \text{elementKey} \rangle &::= \mathit{lowerCamelCasedText} \end{aligned}$$

C.5.5.3 Example use

Composition `({ x:(a), y:(b) })`

Table C.15 shows an execution of this interaction.

Step	`({ x:(a), y:(b)})`	`(a)`	`(b)`
1	$\{x:0, y:1\}$	0	1
2	$\{x:2, y:3\}$	2	3
3	*inactive*	*inactive*	*inactive*
4	$\{x:\mathit{inactive}, y:1\}$	*inactive*	1
5	$\{x:\mathit{inactive}, y:\mathit{inactive}\}$	*inactive*	*inactive*

Table C.15: Timing diagram of an execution of a composition interaction

Decomposition Todo

C.5.6 Function Application

C.5.6.1 Description

The function application interaction is the only way to perform computations in LIDL. The idea is to apply a function to a point and get a result.

C.5.6.2 Syntax

The function application interaction is defined by the operator `()in()()=()`. Here is the EBNF grammar that define all the possible operators of the function application interaction category (There is actually only one operator):

⟨functionApplication⟩ ::= `() in () () = ()`

We see from the grammar that the function has four arguments. The first is the return interaction, the second is the function, the third is the point to input to the function, the third is the image of this point returned by the function.

C.5.6.3 Example use

General Example Here is a very general example of use of the function application interaction.

```
((z) in (‘sin‘)(x)=(y))
```

Table C.16 shows an execution of this interaction.

Step	(‘sin‘)	(x)	(y)	(z)	((z) in (‘sin‘)(x) = (y))
1	*sin*	0	0	42	42
2	*sin*	1	$sin(1)$	42	42
3	*sin*	$\frac{\pi}{2}$	1	42	42
4	*sin*	$\frac{\pi}{2}$	1	1337	1337
5	*sin*	π	0	1337	1337
6	*sin*	π	0	5	5
7	*sin*	*inactive*	*inactive*	5	5

Table C.16: Timing diagram of an execution of a function application interaction

Classical use example Here is an example of the function application interaction where the first argument (return value of the whole interaction) is set to the same as the last argument (return value of the function). This is a common use case, the interaction behaves like a normal function application which most programmers are used to.

```
interaction (cos(x:Number in)):Number out is
  ((y) in (‘cos‘)(x)=(y))
```

Table C.17 shows an execution of this interaction.

Step	(‘cos‘)	(x)	(y)	((y) in (‘cos‘)(x)=(y))
1	*cos*	0	1	1
2	*cos*	1	$cos(1)$	$cos(1)$
3	*cos*	$\frac{\pi}{2}$	0	0
4	*cos*	$\frac{\pi}{2}$	0	0
5	*cos*	π	-1	-1
6	*cos*	π	-1	-1
7	*cos*	*inactive*	*inactive*	*inactive*

Table C.17: Timing diagram of an execution of a function application interaction

Reversed use example Here is an example of the function application interaction where the first argument (return value of the whole interaction) is set to the same as the third argument (input of the function). In this use case, the interaction behaves the opposite way of what most programmers are used to. The interaction value is actually the *input* of the function and not its output.

```
1 ((x) in (‘cos‘)(x)=(y))
```

Table C.18 shows an execution of this interaction.

Step	((x) in (‘cos‘)(x)=(y))	(`cos`)	(x)	(y)
1	0	*cos*	0	1
2	1	*cos*	1	$cos(1)$
3	$\frac{\pi}{2}$	*cos*	$\frac{\pi}{2}$	0
4	$\frac{\pi}{2}$	*cos*	$\frac{\pi}{2}$	0
5	π	*cos*	π	-1
6	π	*cos*	π	-1
7	*inactive*	*cos*	*inactive*	*inactive*

Table C.18: Timing diagram of an execution of a function application interaction

Example of use with composition Here is an example which uses composition and function application together in order to create a more powerful interaction that complies to a complex interface (See C.3.3).

```
1 (({input:(x),output:(y)}) in (‘cos‘)(x)=(y))
```

Table C.19 shows an execution of this interaction.

Step	(({ input:(x), output:(y) }) in (‘cos‘)(x) = (y))	({ input:(x), output:(y) })	(‘cos‘)	(x)	(y)
1	$\{input : 0, output : 1\}$	$\{input : 0, output : 1\}$	cos	0	1
2	$\{input : 1, output : cos(1)\}$	$\{input : 1, output : cos(1)\}$	cos	1	$cos(1)$
3	$\{input : inactive, output : inactive\}$	$\{input : inactive, output : inactive\}$	cos	$inactive$	$inactive$

Table C.19: Timing diagram of an execution of a function application interaction

Variable function example Finally, here is an example of the function application interaction where the function argument (f) is not a function literal as explained in C.5.1, but an interaction that refers to another part of the code, and that can vary over time.

```
1 ((y) in (f)(x)=(y))
```

Table C.20 shows an execution of this interaction.

Step	(x)	(f)	(y)	((y) in (f)(x)=(y))
1	0	cos	1	1
2	0	sin	0	0
3	0	cos	$\frac{\pi}{2}$	0

Table C.20: Timing diagram of an execution of a function application interaction

C.6 Semantics

LIDL is made of four different parts which have been detailed in previous sections of this chapter:

- The definitions sublanguage. Detailed in Section **??**.
- The data type sublanguage. Detailed in Section C.2.
- The interface sublanguage. Detailed in Section C.3.
- The interaction sublanguage. Detailed in Section C.4 and Section C.5.

The formal semantics of the definitions, data types and interfaces sublanguages are relatively straightforward and simple. The detailed description of these sublanguages in their respective sections should be enough to understand them fully. The interaction sublanguage, on the other hand, is where most of LIDL power comes from. This section will detail the formal semantics of LIDL interaction language.

In this section, we will express the semantics of LIDL interaction expressions made Here is an example interaction expression:

C.6.1 Definition expansion

C.6.2 Interpretation

An interaction expression can be interpreted

C.6.3 Compilation

An interaction expression can be compiled

The LIDL compilation process

C.6.3.1 Code generation

LIDL programs compilation results in two functions : an initialization function, and a transition function.

The initialization function takes no arguments and returns the initial execution data structure.

The transition function takes an execution data structure and returns a new execution data structure.

The execution data structure contains the following:

- State: Previous values of the interactions for which we need it (the ones which are used within a `previous` interaction)
- Interface: Value of the interface of the main interaction of the program.
- Arguments: Value of the arguments of the main interaction of the program.
- Memory: Optional, previous value of all interactions of the program in order to provide some form of memoization. If input values of an interaction did not change, then its memoized output values can be used instead of computing it again.

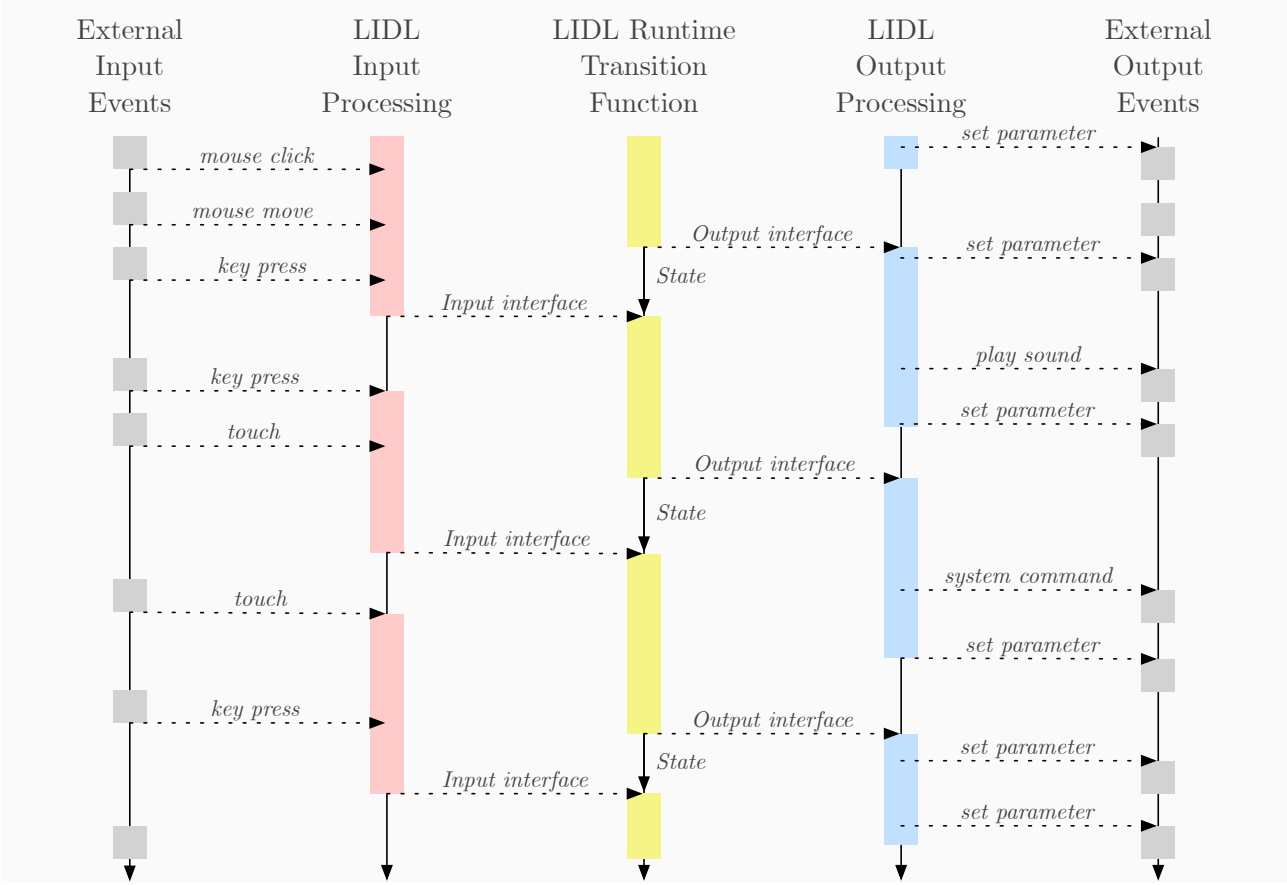


Figure C.14: Sequence diagram of the execution of a LIDL runtime in an event-driven environnement

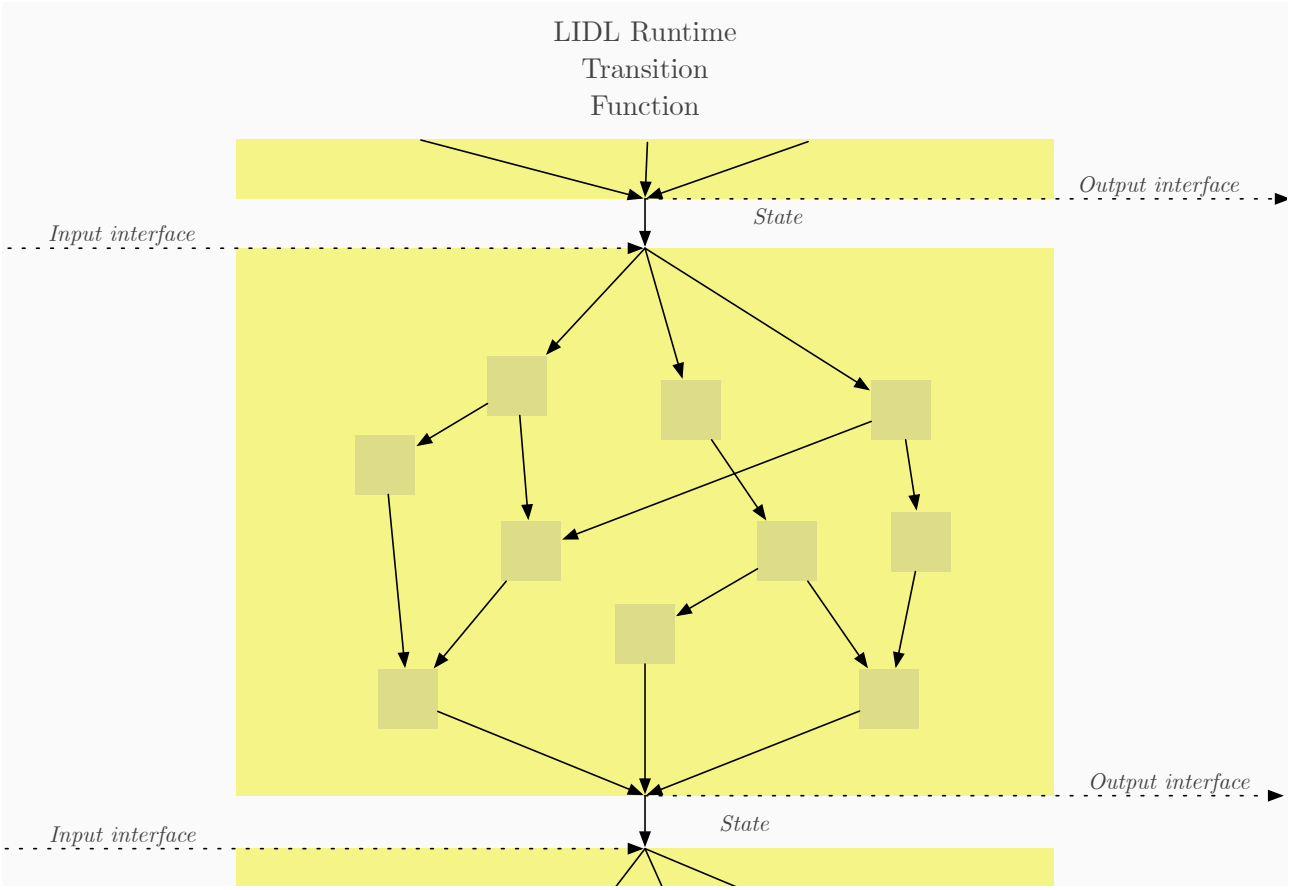


Figure C.15: Sequence diagram of the execution of a LIDL runtime in an event-driven environnement, Focus on the LIDL Runtime